RTC-Web	E.K. Rescorla
Internet-Draft	RTFM, Inc.
Intended status: Standards Track	October 31, 2011
Expires: May 03, 2012	

Security Considerations for RTC-Web
draft-ietf-rtcweb-security-01

Abstract

The Real-Time Communications on the Web (RTC-Web) working group is tasked with standardizing protocols for real-time communications between Web browsers. The major use cases for RTC-Web 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) RTC-Web 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. This document defines the RTC-Web threat model and defines an architecture which provides security within that threat model.

Legal

THIS DOCUMENT AND THE INFORMATION CONTAINED THEREIN ARE PROVIDED ON AN “AS IS” BASIS AND THE CONTRIBUTOR, THE ORGANIZATION HE/SHE REPRESENTS OR IS SPONSORED BY (IF ANY), THE INTERNET SOCIETY, THE IETF TRUST, AND THE INTERNET ENGINEERING TASK FORCE, DISCLAIM ALL WARRANTIES, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO ANY WARRANTY THAT THE USE OF THE INFORMATION THEREIN WILL NOT INFRINGE ANY RIGHTS OR ANY IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE.

Status of this Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet- Drafts is at http://datatracker.ietf.org/drafts/current/.

Internet-Drafts are draft documents valid for a maximum of six months 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 May 03, 2012.

Copyright Notice

Table of Contents

1. Introduction

The Real-Time Communications on the Web (RTC-Web) working group is tasked with standardizing protocols for real-time communications between Web browsers. The major use cases for RTC-Web 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) RTC-Web communications are directly controlled by some Web server. A simple case is shown below.

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

In the system shown in Figure 1, Alice and Bob both have RTC-Web enabled browsers and they visit some Web server which operates a calling service. Each of their browsers exposes standardized JavaScript calling APIs which are used by the Web server to set up a call between Alice and Bob. While this system is topologically similar to a conventional SIP-based system (with the Web server acting as the signaling service and browsers acting as softphones), control has moved to the central Web server; the browser simply provides API points that are used by the calling service. As with any Web application, the Web server can move logic between the server and JavaScript in the browser, but regardless of where the code is executing, it is ultimately under control of the server.

It should be immediately apparent that this type of system poses new security challenges beyond those of a conventional VoIP system. In particular, it needs to contend with malicious calling services. For example, if the calling service can cause the browser to make a call at any time to any callee of its choice, then this facility can be used to bug a user's computer without their knowledge, simply by placing a call to some recording service. More subtly, if the exposed APIs allow the server to instruct the browser to send arbitrary content, then they can be used to bypass firewalls or mount denial of service attacks. Any successful system will need to be resistant to this and other attacks.

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. The Browser Threat Model

The security requirements for RTC-Web follow directly from the requirement that the browser's job is to protect the user. Huang et al. [huang-w2sp] summarize the core browser security guarantee as:

> Users can safely visit arbitrary web sites and execute scripts provided by those sites.

It is important to realize that this includes sites hosting arbitrary malicious scripts. The motivation for this requirement is simple: it is trivial for attackers to divert users to sites of their choice. For instance, an attacker can purchase display advertisements which direct the user (either automatically or via user clicking) to their site, at which point the browser will execute the attacker's scripts. Thus, it is important that it be safe to view arbitrarily malicious pages. Of course, browsers inevitably have bugs which cause them to fall short of this goal, but any new RTC-Web functionality must be designed with the intent to meet this standard. The remainder of this section provides more background on the existing Web security model.

In this model, then, the browser acts as a TRUSTED COMPUTING BASE (TCB) both from the user's perspective and to some extent from the server's. While HTML and JS provided by the server can cause the browser to execute a variety of actions, those scripts operate in a sandbox that isolates them both from the user's computer and from each other, as detailed below.

Conventionally, we refer to either WEB ATTACKERS, who are able to induce you to visit their sites but do not control the network, and NETWORK ATTACKERS, who are able to control your network. Network attackers correspond to the [RFC3552] "Internet Threat Model". In general, it is desirable to build a system which is secure against both kinds of attackers, but realistically many sites do not run HTTPS [RFC2818] and so our ability to defend against network attackers is necessarily somewhat limited. Most of the rest of this section is devoted to web attackers, with the assumption that protection against network attackers is provided by running HTTPS.

3.1. Access to Local Resources

While the browser has access to local resources such as keying material, files, the camera and the microphone, it strictly limits or forbids web servers from accessing those same resources. For instance, while it is possible to produce an HTML form which will allow file upload, a script cannot do so without user consent and in fact cannot even suggest a specific file (e.g., /etc/passwd); the user must explicitly select the file and consent to its upload. [Note: in many cases browsers are explicitly designed to avoid dialogs with the semantics of "click here to screw yourself", as extensive research shows that users are prone to consent under such circumstances.]

Similarly, while Flash SWFs can access the camera and microphone, they explicitly require that the user consent to that access. In addition, some resources simply cannot be accessed from the browser at all. For instance, there is no real way to run specific executables directly from a script (though the user can of course be induced to download executable files and run them).

3.2. Same Origin Policy

Many other resources are accessible but isolated. For instance, while scripts are allowed to make HTTP requests via the XMLHttpRequest() API those requests are not allowed to be made to any server, but rather solely to the same ORIGIN from whence the script came.[I-D.abarth-origin] (although CORS [CORS] and WebSockets [I-D.ietf-hybi-thewebsocketprotocol] provides a escape hatch from this restriction, as described below.) This SAME ORIGIN POLICY (SOP) prevents server A from mounting attacks on server B via the user's browser, which protects both the user (e.g., from misuse of his credentials) and the server (e.g., from DoS attack).

More generally, SOP forces scripts from each site to run in their own, isolated, sandboxes. While there are techniques to allow them to interact, those interactions generally must be mutually consensual (by each site) and are limited to certain channels. For instance, multiple pages/browser panes from the same origin can read each other's JS variables, but pages from the different origins--or even iframes from different origins on the same page--cannot.

3.3. Bypassing SOP: CORS, WebSockets, and consent to communicate

While SOP serves an important security function, it also makes it inconvenient to write certain classes of applications. In particular, mash-ups, in which a script from origin A uses resources from origin B, can only be achieved via a certain amount of hackery. The W3C Cross-Origin Resource Sharing (CORS) spec [CORS] is a response to this demand. In CORS, when a script from origin A executes what would otherwise be a forbidden cross-origin request, the browser instead contacts the target server to determine whether it is willing to allow cross-origin requests from A. If it is so willing, the browser then allows the request. This consent verification process is designed to safely allow cross-origin requests.

While CORS is designed to allow cross-origin HTTP requests, WebSockets [I-D.ietf-hybi-thewebsocketprotocol] allows cross-origin establishment of transparent channels. Once a WebSockets connection has been established from a script to a site, the script can exchange any traffic it likes without being required to frame it as a series of HTTP request/response transactions. As with CORS, a WebSockets transaction starts with a consent verification stage to avoid allowing scripts to simply send arbitrary data to another origin.

While consent verification is conceptually simple--just do a handshake before you start exchanging the real data--experience has shown that designing a correct consent verification system is difficult. In particular, Huang et al. [huang-w2sp] have shown vulnerabilities in the existing Java and Flash consent verification techniques and in a simplified version of the WebSockets handshake. In particular, it is important to be wary of CROSS-PROTOCOL attacks in which the attacking script generates traffic which is acceptable to some non-Web protocol state machine. In order to resist this form of attack, WebSockets incorporates a masking technique intended to randomize the bits on the wire, thus making it more difficult to generate traffic which resembles a given protocol.

4. Security for RTC-Web Applications

4.1. Access to Local Devices

As discussed in Section 1, allowing arbitrary sites to initiate calls violates the core Web security guarantee; without some access restrictions on local devices, any malicious site could simply bug a user. At minimum, then, it MUST NOT be possible for arbitrary sites to initiate calls to arbitrary locations without user consent. This immediately raises the question, however, of what should be the scope of user consent.

For the rest of this discussion we assume that the user is somehow going to grant consent to some entity (e.g., a social networking site) to initiate a call on his behalf. This consent may be limited to a single call or may be a general consent. In order for the user to make an intelligent decision about whether to allow a call (and hence his camera and microphone input to be routed somewhere), he must understand either who is requesting access, where the media is going, or both. So, for instance, one might imagine that at the time access to camera and microphone is requested, the user is shown a dialog that says "site X has requested access to camera and microphone, yes or no" (though note that this type of in-flow interface violates one of the guidelines in Section 3). The user's decision will of course be based on his opinion of Site X. However, as discussed below, this is a complicated concept.

4.1.1. Calling Scenarios and User Expectations

While a large number of possible calling scenarios are possible, the scenarios discussed in this section illustrate many of the difficulties of identifying the relevant scope of consent.

4.1.1.1. Dedicated Calling Services

The first scenario we consider is a dedicated calling service. In this case, the user has a relationship with a calling site and repeatedly makes calls on it. It is likely that rather than having to give permission for each call that the user will want to give the calling service long-term access to the camera and microphone. This is a natural fit for a long-term consent mechanism (e.g., installing an app store "application" to indicate permission for the calling service.) A variant of the dedicated calling service is a gaming site (e.g., a poker site) which hosts a dedicated calling service to allow players to call each other.

With any kind of service where the user may use the same service to talk to many different people, there is a question about whether the user can know who they are talking to. In general, this is difficult as most of the user interface is presented by the calling site. However, communications security mechanisms can be used to give some assurance, as described in Section 4.3.2.

4.1.1.2. Calling the Site You're On

Another simple scenario is calling the site you're actually visiting. The paradigmatic case here is the "click here to talk to a representative" windows that appear on many shopping sites. In this case, the user's expectation is that they are calling the site they're actually visiting. However, it is unlikely that they want to provide a general consent to such a site; just because I want some information on a car doesn't mean that I want the car manufacturer to be able to activate my microphone whenever they please. Thus, this suggests the need for a second consent mechanism where I only grant consent for the duration of a given call. As described in Section 3.1, great care must be taken in the design of this interface to avoid the users just clicking through. Note also that the user interface chrome must clearly display elements showing that the call is continuing in order to avoid attacks where the calling site just leaves it up indefinitely but shows a Web UI that implies otherwise.

4.1.1.3. Calling to an Ad Target

In both of the previous cases, the user has a direct relationship (though perhaps a transient one) with the target of the call. Moreover, in both cases he is actually visiting the site of the person he is being asked to trust. However, this is not always so. Consider the case where a user is a visiting a content site which hosts an advertisement with an invitation to call for more information. When the user clicks the ad, they are connected with the advertiser or their agent.

The relationships here are far more complicated: the site the user is actually visiting has no direct relationship with the advertiser; they are just hosting ads from an ad network. The user has no relationship with the ad network, but desires one with the advertiser, at least for long enough to learn about their products. At minimum, then, whatever consent dialog is shown needs to allow the user to have some idea of the organization that they are actually calling.

However, because the user also has some relationship with the hosting site, it is also arguable that the hosting site should be allowed to express an opinion (e.g., to be able to allow or forbid a call) since a bad experience with an advertiser reflect negatively on the hosting site [this idea was suggested by Adam Barth]. However, this obviously presents a privacy challenge, as sites which host advertisements often learn very little about whether individual users clicked through to the ads, or even which ads were presented.

4.1.2. Origin-Based Security

As discussed in Section 3.2, the basic unit of Web sandboxing is the origin, and so it is natural to scope consent to origin. Specifically, a script from origin A MUST only be allowed to initiate communications (and hence to access camera and microphone) if the user has specifically authorized access for that origin. It is of course technically possible to have coarser-scoped permissions, but because the Web model is scoped to origin, this creates a difficult mismatch. Arguably, origin is not fine-grained enough. Consider the situation where Alice visits a site and authorizes it to make a single call. If consent is expressed solely in terms of origin, then at any future visit to that site (including one induced via mash-up or ad network), the site can bug Alice's computer, use the computer to place bogus calls, etc. While in principle Alice could grant and then revoke the privilege, in practice privileges accumulate; if we are concerned about this attack, something else is needed. There are a number of potential countermeasures to this sort of issue.

Individual Consent Ask the user for permission for each call.

Callee-oriented Consent Only allow calls to a given user.

Cryptographic Consent Only allow calls to a given set of peer keying material or to a cryptographically established identity.

Unfortunately, none of these approaches is satisfactory for all cases. As discussed above, individual consent puts the user's approval in the UI flow for every call. Not only does this quickly become annoying but it can train the user to simply click "OK", at which point the consent becomes useless. Thus, while it may be necessary to have individual consent in some case, this is not a suitable solution for (for instance) the calling service case. Where necessary, in-flow user interfaces must be carefully designed to avoid the risk of the user blindly clicking through.

The other two options are designed to restrict calls to a given target. Unfortunately, Callee-oriented consent does not work well because a malicious site can claim that the user is calling any user of his choice. One fix for this is to tie calls to a cryptographically established identity. While not suitable for all cases, this approach may be useful for some. If we consider the advertising case described in Section 4.1.1.3, it's not particularly convenient to require the advertiser to instantiate an iframe on the hosting site just to get permission; a more convenient approach is to cryptographically tie the advertiser's certificate to the communication directly. We're still tying permissions to origin here, but to the media origin (and-or destination) rather than to the Web origin.

Another case where media-level cryptographic identity makes sense is when a user really does not trust the calling site. For instance, I might be worried that the calling service will attempt to bug my computer, but I also want to be able to conveniently call my friends. If consent is tied to particular communications endpoints, then my risk is limited. However, this is also not that convenient an interface, since managing individual user permissions can be painful.

While this is primarily a question not for IETF, it should be clear that there is no really good answer. In general, if you cannot trust the site which you have authorized for calling not to bug you then your security situation is not really ideal. It is RECOMMENDED that browsers have explicit (and obvious) indicators that they are in a call in order to mitigate this risk.

4.1.3. Security Properties of the Calling Page

Origin-based security is intended to secure against web attackers. However, we must also consider the case of network attackers. Consider the case where I have granted permission to a calling service by an origin that has the HTTP scheme, e.g., http://calling-service.example.com. If I ever use my computer on an unsecured network (e.g., a hotspot or if my own home wireless network is insecure), and browse any HTTP site, then an attacker can bug my computer. The attack proceeds like this:

1. I connect to http://anything.example.org/. Note that this site is unaffiliated with the calling service.

2. The attacker modifies my HTTP connection to inject an IFRAME (or a redirect) to http://calling-service.example.com

3. The attacker forges the response apparently http://calling-service.example.com/ to inject JS to initiate a call to himself.

Note that this attack does not depend on the media being insecure. Because the call is to the attacker, it is also encrypted to him. Moreover, it need not be executed immediately; the attacker can "infect" the origin semi-permanently (e.g., with a web worker or a popunder) and thus be able to bug me long after I have left the infected network. This risk is created by allowing calls at all from a page fetched over HTTP.

Even if calls are only possible from HTTPS sites, if the site embeds active content (e.g., JavaScript) that is fetched over HTTP or from an untrusted site, because that JavaScript is executed in the security context of the page [finer-grained]. Thus, it is also dangerous to allow RTC-Web functionality from HTTPS origins that embed mixed content. Note: this issue is not restricted to PAGES which contain mixed content. If a page from a given origin ever loads mixed content then it is possible for a network attacker to infect the browser's notion of that origin semi-permanently.
[[OPEN ISSUE: What recommendation should IETF make about (a) whether RTCWeb long-term consent should be available over HTTP pages and (b) How to handle origins where the consent is to an HTTPS URL but the page contains active mixed content?]]

4.2. Communications Consent Verification

As discussed in Section 3.3, allowing web applications unrestricted network access via the browser introduces the risk of using the browser as an attack platform against machines which would not otherwise be accessible to the malicious site, for instance because they are topologically restricted (e.g., behind a firewall or NAT). In order to prevent this form of attack as well as cross-protocol attacks it is important to require that the target of traffic explicitly consent to receiving the traffic in question. Until that consent has been verified for a given endpoint, traffic other than the consent handshake MUST NOT be sent to that endpoint.

4.2.1. ICE

Verifying receiver consent requires some sort of explicit handshake, but conveniently we already need one in order to do NAT hole-punching. ICE [RFC5245] includes a handshake designed to verify that the receiving element wishes to receive traffic from the sender. It is important to remember here that the site initiating ICE is presumed malicious; in order for the handshake to be secure the receiving element MUST demonstrate receipt/knowledge of some value not available to the site (thus preventing the site from forging responses). In order to achieve this objective with ICE, the STUN transaction IDs must be generated by the browser and MUST NOT be made available to the initiating script, even via a diagnostic interface. Verifying receiver consent also requires verifying the receiver wants to receive traffic from a particular sender, and at this time; for example a malicious site may simply attempt ICE to known servers that are using ICE for other sessions. ICE provides this verification as well, by using the STUN credentials as a form of per-session shared secret. Those credentials are known to the Web application, but would need to also be known and used by the STUN-receiving element to be useful.
There also needs to be some mechanism for the browser to verify that the target of the traffic continues to wish to receive it. Obviously,

some ICE-based mechanism will work here, but it has been observed that because ICE keepalives are indications, they will not work here, so some other mechanism is needed.

4.2.2. Masking

Once consent is verified, there still is some concern about misinterpretation attacks as described by Huang et al.[huang-w2sp]. As long as communication is limited to UDP, then this risk is probably limited, thus masking is not required for UDP. I.e., once communications consent has been verified, it is most likely safe to allow the implementation to send arbitrary UDP traffic to the chosen destination, provided that the STUN keepalives continue to succeed. In particular, this is true for the data channel if DTLS is used because DTLS (with the anti-chosen plaintext mechanisms required by TLS 1.1) does not allow the attacker to generate predictable ciphertext. However, with TCP the risk of transparent proxies becomes much more severe. If TCP is to be used, then WebSockets style masking MUST be employed.

4.2.3. Backward Compatibility

A requirement to use ICE limits compatibility with legacy non-ICE clients. It seems unsafe to completely remove the requirement for some check. All proposed checks have the common feature that the browser sends some message to the candidate traffic recipient and refuses to send other traffic until that message has been replied to. The message/reply pair must be generated in such a way that an attacker who controls the Web application cannot forge them, generally by having the message contain some secret value that must be incorporated (e.g., echoed, hashed into, etc.). Non-ICE candidates for this role (in cases where the legacy endpoint has a public address) include:

- *STUN checks without using ICE (i.e., the non-RTC-web endpoint sets up a STUN responder.)

- *Use or RTCP as an implicit reachability check.

In the RTCP approach, the RTC-Web endpoint is allowed to send a limited number of RTP packets prior to receiving consent. This allows a short window of attack. In addition, some legacy endpoints do not support RTCP, so this is a much more expensive solution for such endpoints, for which it would likely be easier to implement ICE. For these two reasons, an RTCP-based approach does not seem to address the security issue satisfactorily.
In the STUN approach, the RTC-Web endpoint is able to verify that the recipient is running some kind of STUN endpoint but unless the STUN responder is integrated with the ICE username/password establishment system, the RTC-Web endpoint cannot verify that the recipient consents

to this particular call. This may be an issue if existing STUN servers are operated at addresses that are not able to handle bandwidth-based attacks. Thus, this approach does not seem satisfactory either.

If the systems are tightly integrated (i.e., the STUN endpoint responds with responses authenticated with ICE credentials) then this issue does not exist. However, such a design is very close to an ICE-Lite implementation (indeed, arguably is one). An intermediate approach would be to have a STUN extension that indicated that one was responding to RTC-Web checks but not computing integrity checks based on the ICE credentials. This would allow the use of standalone STUN servers without the risk of confusing them with legacy STUN servers. If a non-ICE legacy solution is needed, then this is probably the best choice.

Once initial consent is verified, we also need to verify continuing consent, in order to avoid attacks where two people briefly share an IP (e.g., behind a NAT in an Internet cafe) and the attacker arranges for a large, unstoppable, traffic flow to the network and then leaves. The appropriate technologies here are fairly similar to those for initial consent, though are perhaps weaker since the threats is less severe.

[[OPEN ISSUE: Exactly what should be the requirements here? Proposals include ICE all the time or ICE but with allowing one of these non-ICE things for legacy.]]

4.2.4. IP Location Privacy

Note that as soon as the callee sends their ICE candidates, the callee learns the callee's IP addresses. The callee's server reflexive address reveals a lot of information about the callee's location. In order to avoid tracking, implementations may wish to suppress the start of ICE negotiation until the callee has answered. In addition, either side may wish to hide their location entirely by forcing all traffic through a TURN server.

4.3. Communications Security

Finally, we consider a problem familiar from the SIP world: communications security. For obvious reasons, it MUST be possible for the communicating parties to establish a channel which is secure against both message recovery and message modification. (See [RFC5479] for more details.) This service must be provided for both data and voice/video. Ideally the same security mechanisms would be used for both types of content. Technology for providing this service (for instance, DTLS [RFC4347] and DTLS-SRTP [RFC5763]) is well understood. However, we must examine this technology to the RTC-Web context, where the threat model is somewhat different.

In general, it is important to understand that unlike a conventional SIP proxy, the calling service (i.e., the Web server) controls not only the channel between the communicating endpoints but also the application running on the user's browser. While in principle it is

possible for the browser to cut the calling service out of the loop and directly present trusted information (and perhaps get consent), practice in modern browsers is to avoid this whenever possible. "In-flow" modal dialogs which require the user to consent to specific actions are particularly disfavored as human factors research indicates that unless they are made extremely invasive, users simply agree to them without actually consciously giving consent. [abarth-rtcweb]. Thus, nearly all the UI will necessarily be rendered by the browser but under control of the calling service. This likely includes the peer's identity information, which, after all, is only meaningful in the context of some calling service.
This limitation does not mean that preventing attack by the calling service is completely hopeless. However, we need to distinguish between two classes of attack:

Retrospective compromise of calling service. The calling service is is non-malicious during a call but subsequently is compromised and wishes to attack an older call.

During-call attack by calling service. The calling service is compromised during the call it wishes to attack.

Providing security against the former type of attack is practical using the techniques discussed in Section 4.3.1. However, it is extremely difficult to prevent a trusted but malicious calling service from actively attacking a user's calls, either by mounting a MITM attack or by diverting them entirely. (Note that this attack applies equally to a network attacker if communications to the calling service are not secured.) We discuss some potential approaches and why they are likely to be impractical in Section 4.3.2.

4.3.1. Protecting Against Retrospective Compromise

In a retrospective attack, the calling service was uncompromised during the call, but that an attacker subsequently wants to recover the content of the call. We assume that the attacker has access to the protected media stream as well as having full control of the calling service.
If the calling service has access to the traffic keying material (as in SDES [RFC4568]), then retrospective attack is trivial. This form of attack is particularly serious in the Web context because it is standard practice in Web services to run extensive logging and monitoring. Thus, it is highly likely that if the traffic key is part of any HTTP request it will be logged somewhere and thus subject to subsequent compromise. It is this consideration that makes an automatic, public key-based key exchange mechanism imperative for RTC-Web (this is a good idea for any communications security system) and this mechanism SHOULD provide perfect forward secrecy (PFS). The

signaling channel/calling service can be used to authenticate this mechanism.

In addition, the system MUST NOT provide any APIs to extract either long-term keying material or to directly access any stored traffic keys. Otherwise, an attacker who subsequently compromised the calling service might be able to use those APIs to recover the traffic keys and thus compromise the traffic.

4.3.2. Protecting Against During-Call Attack

Protecting against attacks during a call is a more difficult proposition. Even if the calling service cannot directly access keying material (as recommended in the previous section), it can simply mount a man-in-the-middle attack on the connection, telling Alice that she is calling Bob and Bob that he is calling Alice, while in fact the calling service is acting as a calling bridge and capturing all the traffic.

While in theory it is possible to construct techniques which protect against this form of attack, in practice these techniques all require far too much user intervention to be practical, given the user interface constraints described in [abarth-rtcweb].

4.3.2.1. Key Continuity

One natural approach is to use "key continuity". While a malicious calling service can present any identity it chooses to the user, it cannot produce a private key that maps to a given public key. Thus, it is possible for the browser to note a given user's public key and generate an alarm whenever that user's key changes. SSH [RFC4251] uses a similar technique. (Note that the need to avoid explicit user consent on every call precludes the browser requiring an immediate manual check of the peer's key).

Unfortunately, this sort of key continuity mechanism is far less useful in the RTC-Web context. First, much of the virtue of RTC-Web (and any Web application) is that it is not bound to particular piece of client software. Thus, it will be not only possible but routine for a user to use multiple browsers on different computers which will of course have different keying material (SACRED [RFC3760] notwithstanding.) Thus, users will frequently be alerted to key mismatches which are in fact completely legitimate, with the result that they are trained to simply click through them. As it is known that users routinely will click through far more dire warnings [cranor-wolf], it seems extremely unlikely that any key continuity mechanism will be effective rather than simply annoying.

Moreover, it is trivial to bypass even this kind of mechanism. Recall that unlike the case of SSH, the browser never directly gets the peer's identity from the user. Rather, it is provided by the calling service. Even enabling a mechanism of this type would require an API to allow the calling service to tell the browser "this is a call to user X". All the calling service needs to do to avoid triggering a key continuity warning is to tell the browser that "this is a call to user Y" where Y is close to X. Even if the user actually checks the other side's name (which all available evidence indicates is unlikely), this would require (a) the browser to trusted UI to provide the name and (b) the user to not be fooled by similar appearing names.

4.3.2.2. Short Authentication Strings

ZRTP [RFC6189] uses a "short authentication string" (SAS) which is derived from the key agreement protocol. This SAS is designed to be read over the voice channel and if confirmed by both sides precludes MITM attack. The intention is that the SAS is used once and then key continuity (though a different mechanism from that discussed above) is used thereafter.

Unfortunately, the SAS does not offer a practical solution to the problem of a compromised calling service. "Voice conversion" systems, which modify voice from one speaker to make it sound like another, are an active area of research. These systems are already good enough to fool both automatic recognition systems [farus-conversion] and humans [kain-conversion] in many cases, and are of course likely to improve in future, especially in an environment where the user just wants to get on with the phone call. Thus, even if SAS is effective today, it is likely not to be so for much longer. Moreover, it is possible for an attacker who controls the browser to allow the SAS to succeed and then simulate call failure and reconnect, trusting that the user will not notice that the "no SAS" indicator has been set (which seems likely).

Even were SAS secure if used, it seems exceedingly unlikely that users will actually use it. As discussed above, the browser UI constraints preclude requiring the SAS exchange prior to completing the call and so it must be voluntary; at most the browser will provide some UI indicator that the SAS has not yet been checked. However, it it is well-known that when faced with optional mechanisms such as fingerprints, users simply do not check them [whitten-johnny] Thus, it is highly unlikely that users will ever perform the SAS exchange.

Once uses have checked the SAS once, key continuity is required to avoid them needing to check it on every call. However, this is problematic for reasons indicated in Section 4.3.2.1. In principle it is of course possible to render a different UI element to indicate that calls are using an unauthenticated set of keying material (recall that the attacker can just present a slightly different name so that the attack shows the same UI as a call to a new device or to someone you haven't called before) but as a practical matter, users simply ignore such indicators even in the rather more dire case of mixed content warnings.

4.3.2.3. Recommendations

[[OPEN ISSUE: What are the best UI recommendations to make? Proposal: take the text from [I-D.kaufman-rtcweb-security-ui] Section 2]]

[[OPEN ISSUE: Exactly what combination of media security primitives should be specified and/or mandatory to implement? In particular, should we allow DTLS-SRTP only, or both DTLS-SRTP and SDES. Should we allow RTP for backward compatibility?]]

5. Security Considerations

This entire document is about security.

6. Acknowledgements

Bernard Aboba, Harald Alvestrand, Cullen Jennings, Hadriel Kaplan (S 4.2.1), Matthew Kaufman, Magnus Westerland.

7. References

7.1. Normative References

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

7.2. Informative References

[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.
[RFC3552]	Rescorla, E. and B. Korver, "Guidelines for Writing RFC Text on Security Considerations", BCP 72, RFC 3552, July 2003.
[RFC2818]	Rescorla, E., "HTTP Over TLS", RFC 2818, May 2000.
[RFC5479]	Wing, D., Fries, S., Tschofenig, H. and F. Audet, "Requirements and Analysis of Media Security Management Protocols", RFC 5479, April 2009.
[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.
[RFC4347]	Rescorla, E. and N. Modadugu, "Datagram Transport Layer Security", RFC 4347, April 2006.
[RFC4568]	Andreasen, F., Baugher, M. and D. Wing, "Session Description Protocol (SDP)

	Security Descriptions for Media Streams", RFC 4568, July 2006.
[RFC4251]	Ylonen, T. and C. Lonvick, "The Secure Shell (SSH) Protocol Architecture", RFC 4251, January 2006.
[RFC3760]	Gustafson, D., Just, M. and M. Nystrom, "Securely Available Credentials (SACRED) - Credential Server Framework", RFC 3760, April 2004.
[RFC6189]	Zimmermann, P., Johnston, A. and J. Callas, "ZRTP: Media Path Key Agreement for Unicast Secure RTP", RFC 6189, April 2011.
[RFC5245]	Rosenberg, J., "Interactive Connectivity Establishment (ICE): A Protocol for Network Address Translator (NAT) Traversal for Offer/Answer Protocols", RFC 5245, April 2010.
[I-D.abarth-origin]	Barth, A, "The Web Origin Concept", Internet-Draft draft-abarth-origin-09, November 2010.
[I-D.ietf-hybi-thewebsocketprotocol]	Fette, I and A Melnikov, "The WebSocket protocol", Internet-Draft draft-ietf-hybi-thewebsocketprotocol-17, September 2011.
[I-D.kaufman-rtcweb-security-ui]	Kaufman, M, "Client Security User Interface Requirements for RTCWEB", Internet-Draft draft-kaufman-rtcweb-security-ui-00, June 2011.
[abarth-rtcweb]	Barth, A., "Prompting the user is security failure", RTC-Web Workshop, .
[whitten-johnny]	Whitten, A. and J.D. Tygar, "Why Johnny Can't Encrypt: A Usability Evaluation of PGP 5.0", Proceedings of the 8th USENIX Security Symposium, 1999, .
[cranor-wolf]	Sunshine, J., Egelman, S., Almuhimedi, H., Atri, N. and L. cranor, "Crying Wolf: An Empirical Study of SSL Warning Effectiveness", Proceedings of the 18th USENIX Security Symposium, 2009, .
[kain-conversion]	Kain, A. and M. Macon, "Design and Evaluation of a Voice Conversion Algorithm based on Spectral Envelope Mapping and Residual Prediction", Proceedings of ICASSP, May 2001, .
[farus-conversion]	Farrus, M., Erro, D. and J. Hernando, "Speaker Recognition Robustness to Voice Conversion", .
[huang-w2sp]	

	Huang, L-S., Chen, E.Y., Barth, A., Rescorla, E. and C. Jackson, "Talking to Yourself for Fun and Profit", W2SP, 2011, .
[finer-grained]	Barth, A. and C. Jackson, "Beware of Finer-Grained Origins", W2SP, 2008, .
[CORS]	van Kesteren, A., "Cross-Origin Resource Sharing", .

Appendix A. A Proposed Security Architecture [No Consensus on This]

This section contains a proposed security architecture, based on the considerations discussed in the main body of this memo. This section is currently the opinion of the author and does not have consensus though some (many?) elements of this proposal do seem to have general consensus.

Appendix A.1. Trust Hierarchy

The basic assumption of this proposal 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.

Appendix A.1.1. 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).

- *Other users: RTC-Web 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 an 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, it's safe to temporarily give him access to the camera and microphone for the purpose of the call. The point here is that we must first identify other elements before we can determine whether 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.

Appendix A.1.2. 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.

Appendix A.2. 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.

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

Appendix A.2.1. 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 [REF] or may be assembled piecemeal by the JS. In either case, it will contain:

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

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 Appendix Appendix A.3.6.4 I believe PeerConnection can be agnostic to them), but for now it's easiest to think of as a BrowserID assertion.

This message is sent to the signaling server, e.g., by XMLHttpRequest [REF] or by WebSockets [I-D.ietf-hybi-thewebsocketprotocol]. 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 verity 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.

Appendix A.2.2. Media Consent Verification

As described in Section 4.2. This proposal specifies that that 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.

Appendix A.2.3. 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 extracted 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.

Appendix A.2.4. 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. If a keepalive fails and no new ICE channels can be established, then the session is terminated.

Appendix A.3. Detailed Technical Description

Appendix A.3.1. Origin and Web Security Issues

The basic unit of permissions for RTC-Web is the origin [I-D.abarth-origin]. 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. Thus, clients MUST treat HTTP and HTTPS origins as different permissions domains and SHOULD NOT permit access to any RTC-Web functionality from scripts fetched over non-secure (HTTP) origins. If an HTTPS origin contains mixed active content (regardless of whether it is present on the specific page attempting to access RTC-Web functionality), any access MUST be treated as if it came from the HTTP origin. For instance, if a https://www.example.com/example.html loads https://www.example.com/example.js and http://www.example.org/jquery.js, any attempt by example.js to access RTCWeb functionality MUST be treated as if it came from http://www.example.com/. Note that many browsers already track mixed content and either forbid it by default or display a warning.

Appendix A.3.2. 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:

- Requests for one-time camera/microphone access.

- Requests for permanent 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. 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 SHOULD display the appropriate UI for those permissions.

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.

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 Appendix Appendix A.3.3 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. Appendix Appendix A.3.5 provides more on this.

Appendix A.3.3. Communications Consent

Browser client implementations of RTC-Web MUST implement ICE. Server gateway implementations which operate only at public IP addresses may implement ICE-Lite.
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. [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. We will need to define a new mechanism for this. [OPEN ISSUE: what to do here.]

Appendix A.3.4. IP Location Privacy

As mentioned in Section 4.2.4 above, 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 following two API requirements are intended to mitigate this issue:

API Requirement: The API MUST provide a mechanism 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.

API Requirement: The API MUST provide a mechanism for the calling application to indicate that only TURN candidates are to be used. This prevents the peer from learning one's IP address at all.

Appendix A.3.5. Communications Security

Implementations MUST implement DTLS and DTLS-SRTP. 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 and/or plain RTP, DTLS-SRTP MUST be selected.

[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: 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.

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 or not the transmission is cryptographically protected and whether that protection is based on a key that was delivered out-of-band (from a server) or was generated as a result of a pairwise negotiation.

*If the far endpoint was directly verified Appendix Appendix A.3.6 the "security characteristics" MUST include the verified information.

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

*If the transmission is cryptographically protected, the The algorithms in use (For example: "AES-CBC" or "Null Cipher".)

- If the transmission is cryptographically protected, the "security characteristics" MUST indicate whether PFS is provided.

- If the transmission is cryptographically protected via an end-to-end mechanism 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.

Appendix A.3.6. Web-Based Peer Authentication

Appendix A.3.6.1. Generic Concepts

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. This means that the PeerConnection API MUST arrange to talk directly to the identity provider in a way that cannot be impersonated by the calling site. The following sections provide two examples of this.

Appendix A.3.6.2. 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 RTC-Web calls, as shown below.

```
+----------------------+                     +----------------------+
|                      |                     |                      |
|    Alice's Browser   |                     |     Bob's Browser    |
|                      | OFFER ------------> |                      |
|   Calling JS Code    |                     |    Calling JS Code   |
|          ^           |                     |          ^           |
|          |           |                     |          |           |
|          v           |                     |          v           |
|    PeerConnection    |                     |    PeerConnection    |
|       |      ^       |                     |       |      ^       |
| Finger|      |Signed |                     |Signed |      |       |
| print |      |Finger |                     |Finger |      |"Alice"|
|       |      |print  |                     |print  |      |       |
|       v      |       |                     |       v      |       |
|   +--------------+   |                     |   +---------------+  |
|   |  BrowserID   |   |                     |   |  BrowserID    |  |
|   |  Signer      |   |                     |   |  Verifier     |  |
|   +--------------+   |                     |   +---------------+  |
|           ^          |                     |          ^           |
+-----------|----------+                     +----------|-----------+
            |                                           |
            | Get certificate                           |
            v                                           | Check
+----------------------+                                | certificate
|                      |                                |
|       Identity       |/-------------------------------+
|       Provider       |
|                      |
+----------------------+
```

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.

2. The PeerConnection instantiates the BrowserID signer in an invisible IFRAME. The IFRAME is tagged with an origin that indicates that it was generated by the PeerConnection (this prevents ordinary JS from implementing it). The BrowserID signer is provided with Alice's fingerprint. Note that the IFRAME here does not render any UI. It is being used solely to allow the browser to load the BrowserID signer in isolation, especially from the calling site.

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. [Note: there are well-understood Web mechanisms for this that I am excluding here for simplicity.]

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":{
       "identityType":"browserid",
         "assertion": {
         "digest":"<hash of fingerprint and session IDs>",
         "audience": "[TBD]"
         "valid-until": 1308859352261,
        }, // signed using user's key
        "certificate": {
          "email": "rescorla@gmail.com",
          "public-key": "<ekrs-public-key>",
          "valid-until": 1308860561861,
        } // certificate is signed by gmail.com
        }
   }
```

Note that we only expect to sign the fingerprint values and the session IDs, in order to allow the JS or calling service to modify the rest of the SDP, while protecting the identity binding. [OPEN ISSUE: should we sign seq too?]
[TODO: NEed to talk about Audience a bit.]
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 a BrowserID verifier in an IFRAME and provides it the signed assertion.

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.

Appendix A.3.6.3. 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 RTC-Web concepts:

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

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.

Appendix A.3.6.4. Generic Identity Support

I believe it's possible to build a generic interface between the PeerConnection and any identity sub-module so that the PeerConnection just gets pointed to the IDP (which the relying party either trusts or not) and JS from the IDP provides the concrete interfaces. However, I need to work out the details, so I'm not specifying this yet. If it works, the previous two sections will just be examples.

Author's Address

Eric Rescorla Rescorla RTFM, Inc. 2064 Edgewood Drive Palo Alto, CA 94303 USA Phone: +1 650 678 2350 EMail: ekr@rtfm.com