Ensuring endpoint authenticity in WebRTC peer-to-peer communication

Willem De Groef
iMinds-DistriNet, KU Leuven
3001 Leuven, Belgium
willem.degroef@cs.kuleuven.be

Deepak Subramanian
CIDRE research group
CentraleSupélec, France
subudeepak@gmail.com

Martin Johns
SAP SE
Germany
martin.johns@sap.com

Frank Piessens
iMinds-DistriNet, KU Leuven
3001 Leuven, Belgium
frank.piessens@cs.kuleuven.be

Lieven Desmet
iMinds-DistriNet, KU Leuven
3001 Leuven, Belgium
lieven.desmet@cs.kuleuven.be

ABSTRACT

WebRTC is one of the latest additions to the ever growing repository of Web browser technologies, which push the envelope of native Web application capabilities. WebRTC allows real-time peer-to-peer audio and video chat, that runs purely in the browser. Unlike existing video chat solutions, such as Skype, that operate in a closed identity ecosystem, WebRTC was designed to be highly flexible, especially in the domains of signaling and identity federation. This flexibility, however, opens avenues for identity fraud. In this paper, we explore the technical underpinnings of WebRTC's identity management architecture. Based on this analysis, we identify three novel attacks against endpoint authenticity. To answer the identified threats, we propose and discuss defensive strategies, including security improvements for the WebRTC specifications and mitigation techniques for the identity and service providers.

CCS Concepts

•**Security and privacy** → **Web application security;** *Browser security; Distributed systems security; Web protocol security;*

Keywords

Web Application Security; WebRTC; Real-time Communication; Peer-to-Peer Communication; Peer Authentication

1. INTRODUCTION

WebRTC is a new technology stack that enables real-time communication on the Web. Without the need for additional tools or applications, users can now make peer-to-peer connections between each other and stream audio and video, simply with the built-in functionality of their browsers. All this functionality is available to every website, and can be fully controlled via JavaScript APIs.

WebRTC is a joint effort of W3C and IETF, and backed by a large set of industry players. It is expected to be a hugely disruptive technology in the telco sector, and it impacts the whole client-server paradigm we are so used to in the Web.

The overall security mechanisms of WebRTC are well thought out, especially at the network and transport layer [16, 14]. Several protocols and services are inherited from the VoIP and SIP ecosystem, and have already earned their stripes. Fully new however, is the fact that the technology all of a sudden can be controlled from within the Webapplication.

This paper therefore investigates how a malicious application (either due to a malicious Webserver, or due to malicious code injection) can impact the security guarantees of the peer-to-peer WebRTC connection. In particular, we report on the impact of application-level code on the endpoint authenticity guarantees that can be provided by the peer-to-peer connection. In case the endpoint authenticity can get compromised, malicious entities, able to execute JavaScript in the website or malicious website owners, are able to eavesdrop on the peer-to-peer connections between browsers.

Guided by the WebRTC specifications and the early browser implementations, we report in this paper on our security assessment of the current state-of-practice on endpoint authenticity in WebRTC (Section 3). In particular, we describe three novel attack types to compromise endpoint authenticity in Section 4, which allows an attacker to intercept the peer-to-peer connection. Moreover, we propose mitigation techniques and security advices to prevent those attacks from happening in Section 5. These mitigations have been grouped by actor in the WebRTC architecture: security improvements for the WebRTC specifications and the browser implementations, and mitigation techniques for the Identity Providers and website owners.

2. WEBRTC BACKGROUND

In this section we will give a brief overview of the WebRTC technology, and in particular the signaling path and the media path. Next, we will dig into the identity provisioning, as this is another crucial part for this paper. Finally, we will cover the JavaScript APIs that can be used from within the browser to initiate and control WebRTC sessions.

WebRTC stands for a complete stack of protocols and

SAC 2016, April 04-08, 2016, Pisa, Italy

DOI: http://dx.doi.org/10.1145/2851613.2851804

APIs to setup and participate in real-time communication across networks from within a Web browser. Via a WebRTC-enabled browser, it becomes possible to interact, via real-time multi-party audio and video conferencing in a peer-to-peer fashion, with other users. The complete WebRTC technology stack is controllable via JavaScript from within Web applications. Although still very new, most of the underlying technology, including the network and transport layer stack, are reused from existing multimedia- or telecommunication protocols (e.g, the Session Initiation Protocol (SIP) [10] to support multimedia sessions).

2.1 Architecture of WebRTC

The high-level architecture of WebRTC can be split into two different *planes* as shown in Figure 1. The distinction is made based on the kind of data is sent over it. The green layer, or the *media plane* delivers the peer-to-peer real-time streams. The top red layer, or the *signaling plane* delivers all control- and meta-data between the endpoints.

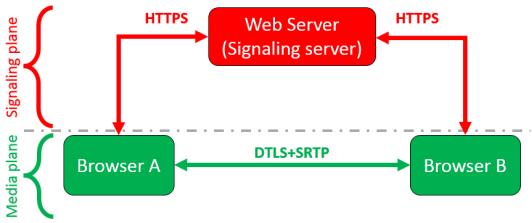

Figure 1: Simple architectural view of WebRTC: the signaling plane via the Signaling Webserver (in red), and the media plane between the communication endpoints (in green).

Signaling Plane

The signaling plane consists of one or more signaling servers that mediate and route communication, typically over an HTTPS connection, between two or more endpoints. The second task of a signaling server is to serve the initial client-side application-specific code that interacts with the JavaScript API for WebRTC [4].

Media Plane

The media plane will take care of the peer-to-peer connections between the endpoints, relying on the Datagram Transport Layer Security (*DTLS*) protocol [17]. To support real-time streams, the media plane relies on the SRTP transport protocol on top of DTLS. Data channels require the use of the Stream Control Transmission Protocol (*SCTP*) over DTLS.

Complete Architectural Overview

Due to complex setups of today's network infrastructure, the architectural picture is often far more complex, as shown in Figure 2. Services to obtain mapped public IP addresses from within private networks (e.g., STUN and TURN servers, shown in purple), and to provide identity management (shown in blue), are all part of the complete architecture.

2.2 WebRTC Communication

Although simple on a high level, setting up a WebRTC communication channel between two endpoints, involves a strict sequence of actions and events, which we will briefly cover in this section.

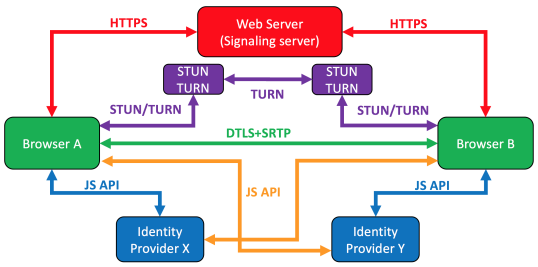

Figure 2: Full architectural view of WebRTC (based on [16]): the HTTPS signaling plane (red), the DTLS/SRTP media path (green), the interaction with STUN/TURN (purple), and the interaction with Identity Providers (blue for the assertion generation and yellow for the assertion verification).

Step 1: Setting up the endpoints

The communication providers in the WebRTC context are called *signaling servers*. These servers are contacted by the endpoints, typically over HTTPS, to download the application-specific code and to authenticate to the Webapplication (if any). The application-specific code has access to the WebRTC JavaScript API, available in any modern browser, and will take care of all end-user interactions and exchange of meta-data via the signaling server.

Step 2: Communication request by endpoint

After a communication request by the end-user, the application-specific code will ask, via a WebRTC API, to set up a `RTCPeerConnection` object. The browser of the end-user will collect all necessary meta-data (e.g., relevant network data & media preferences) and exchange it with the remote party, via the signaling server. The `RTCPeerConnection` is used to manage the peer-to-peer connection during its entire life cycle.

Step 3: Exchange of meta-data via signaling server

The meta-data is sent towards the signaling server via Session Description Protocol (*SDP*) [18] messages and Interactive Connectivity Establishment candidates. The signaling server relays this data to the necessary communicating parties, via an unspecified protocol (e.g., WebSockets or *socket.io*).

The communicating party will send an *SDP offer*, combining both the meta-data and an identity assertion (see Section 2.3), to the remote party via the signaling server. The remote party will respond with an *SDP answer*, containing the meta-data of the remote party, again via the signaling server. At any point in time, meta-data updates can be send via new SDP offer/answer messages.

Step 4: Peer-to-peer communication channel setup

After the endpoints agree on the media preferences, ICE candidates are exchanged to effectively establish the peer-to-peer communication channel. As part of the DTLS connection set up, to secure the communication channel, the certificate fingerprint, which is sent as part of the SDP message, is used to prevent tampering with the communication channel. It can be used to verify the authenticity of the communicating endpoint (this is not the same as verifying its *identity*).

Step 5: Peer-to-peer real-time communication

After establishing the peer-to-peer communication channel, the SRTP/DTLS communication channel is used to route both media streams and the data channel. Additional streams can be added or removed during communication by both endpoints by calling the appropriate WebRTC JavaScript API.

Finally, Listing 1 briefly illustrate how the WebRTC JavaScript APIs can be used to perform some of the steps mentioned before. In particular, the code fragment is part of the client-side application code, set by the signaling server in step 1. It creates a communication path towards the signaling server, and sets up the *RTCPeerConnection* object (step 2). Furthermore, the code adds handlers to make sure that the SDP offers are being sent over to the remote party via the signaling path (step 3), Finally, also media streams (such as coming from the microphone and camera) are added to the peer connection (step 5).

```
1  /* the signaling channel is the application specific
       protocol to send data via the signaling plane. */
2  var signalingChannel = new SignalingChannel();
3  /* setup of the peer connection object */
4  var pc = new RTCPeerConnection({});
5  navigator.getUserMedia(..., function(stream) {
6    /* Media streams are added to the PeerConnection */
7    pc.addStream(stream);
8    /* SDP offers are sent via the signaling path */
9    pc.createOffer(function(offer) {
10     pc.setLocalDescription(offer);
11     signalingChannel.send(offer.sdp);
12   });
13 });
```

Listing 1: This client-side code fragment illustrates some of the basic WebRTC APIs to add streams to a peer-to-peer connection and use handlers to send SDP offers via the signaling plane.

2.3 Peer Authentication

The main idea about Web-based peer authentication is to allow two end-users to verify each other identity, without relying on a third party, in this case the signaling server. The DTLS handshake between the two endpoints relies on self-signed certificates. Therefore, the certificates themselves can not be used to authenticate the endpoints as there is no explicit chain of trust to verify.

The WebRTC architecture provides a mechanism to allow applications to perform their own authentication and identity verification between endpoints. These interactions are done via JavaScript APIs within the browser itself. Each endpoint can specify an Identity Provider while generating the SDP offer/answer (see Step 3 of Section 2.2). Based on the content of a received SDP message, the endpoint can check with the Identity Provider to verify the received certificate and thus to validate the identity.

This leaves the browsers of the end-users as the relying party and the IdP as the assertion party for the identity of each end-user. This whole process is discussed in much more detail in its corresponding IETF document [16] and by Bergkvist et al. [4].

2.3.1 *Interacting with the Identity Provider*

WebRTC provides a very loose coupling to the Identity Provider, so that different Identity Providers with varying protocols can be used.

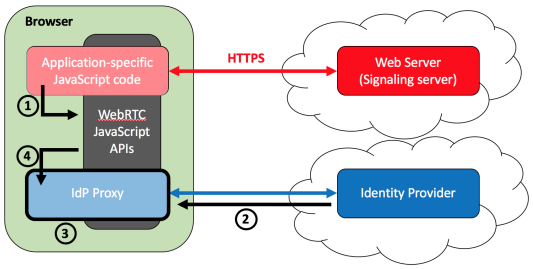

Figure 3: WebRTC integration of the Identity Provider.

Figure 3 provides an architectural overview of the integration of an Identity Provider. In essence, the browser will load a IdP-specific proxy (called *IdP Proxy*) to interact with the Identity Provider, and this proxy implements a very generic interface towards the browser for peer authentication.

Interacting with the Identity Provider happens in 4 steps, as sketched in Figure 3:

1. The signaling server serves application-specific JavaScript code that can set an Identity Provider.
2. Based on the Identity Provider, the browser downloads the IdP proxy code.[1]
3. The IdP proxy code gets executed in an restricted, well-isolated environment (e.g., a hidden iframe or a realm [6]).
4. Communication between the application-specific code and the IdP (proxy) happens over a secure message-passing channel.

WebRTC supports two operations for peer authentication: *identity assertion generation* and *identity assertion verification*. In a first step, an endpoint must generate an identity assertion. After receiving an identity assertion, the other endpoint must verify its authenticity by contacting the IdP. Both steps are explained in further detail in the next sections.

2.3.2 *Identity Generation*

The identity generation process relies on a custom Identity Provider, often specified by the application-specific code coming from the signaling server. The API for this process is `RTCPeerConnection.setIdentityProvider`. It can be expected that the signaling server will take user preferences, on the choice of Identity Provider, into account. In the case no Identity Provider is set, the browser *can* fall back to a default, browser-preferred Identity Provider.

It is clear that prior to the identity generation process, the Identity Provider must have an identity of the end-user. This could be the user having an account on a social network associated with the Identity Provider or e.g., an OpenID provider.

The identity assertion generation works as follows:

1. The IdP receives one or more DTLS-SRTP fingerprints of the `PeerConnection` of the endpoint
2. The IdP generates an identity assertion, which includes the initial fingerprint of the endpoint, based on the available identity of the end-user
3. The identity assertion is attached as part of the SDP offer or answer object before it is sent towards the re-

[1]The IdP proxy code can always be found on a fixed URL, as specified in [16]: `[[scheme]]+ '://' + [[idp domain name]] + '/.well-known/idp-proxy/' + [[protocol]]`.

mote party

Listing 2 illustrates how the identity assertion can be explicitly generated via the WebRTC APIs, and how the client-side JavaScript code can install handlers to be informed about the result of the identity assertion generation. Note that the explicit call to generate an assertion is not necessary, as an identity assertion will automatically be generated if an Identity Provider has been set up for the PeerConnection.

```
1 var pc = new PeerConnection();
2 pc.setIdentityProvider("idp.com");
3 pc.onidentityresult = function(ev) {
4    /* fired when the identity assertion succeeded */
5 };
6 pc.onidpassertionerror = function(ev) {
7    /* fired when the identity assertion has failed */
8 };
9 pc.getIdentityAssertion();
```

Listing 2: Example of the WebRTC APIs to generate an identity assertion as well as to be informed about the result of the identity assertion generation.

As part of the identity generation, the identity of the end-user will be validated by the Identity Provider. The result of this operation depends on the authentication status of the end-user. The browser can choose to ignore identity assertion unless the function `setIdentityProvider` is explicitly called by the webpage. The format of the identity assertion is a base64 encoded JSON, and is Identity Provider specific.

Authenticated user. In case the end-user is already authenticated to the Identity Provider prior to the WebRTC connection setup, the IdP proxy within the browser will be able to interact with the public server interface of the IdP. For security reasons, the IdP proxy operates from within an isolated context making it impossible to to interact directly with the end-user. This makes it impossible for the IdP to load e.g. an authentication form and therefore has to rely on the pre-established authentication mechanism of the authenticated user (such as cookies, basic HTTP authentication or local-storage).

Non-authenticated user. In case the end-user is not yet authenticated to the Identity Provider at time of setting up the WebRTC connection, the assertion generation will fail, as the IdP proxy is not able to interact with the end-user.

However, by providing an URL (pointing to a authentication form) as part of the failure message, the IdP proxy can hint to the Webapplication to load this URL in a separate window or iframe. By doing so, the user is able to authenticate to the Identity Provider, and after a successful authentication, the authentication form will inform the Webapplication, via a `LOGINDONE` message, that the assertion generation process can be started over with an authenticated user.

2.3.3 *Identity Verification*

As part of the connection setup, the remote endpoint will verify the provided identity assertion. To do so, the Identity Provider of the local endpoint will be loaded, and the `validateAssertion` message will be sent to the IdP Proxy.

Listing 3 shows how the client-side code can also interact with the WebRTC APIs to retrieve the identity of the remote endpoint, and how handlers can be added to be informed about the validity of the remote identity.

```
1  var pc = new RTCPeerConnection();
2  var identity = pc.peerIdentity;
3  if (identity) {
4     alert("Identity of the peer: idp='" + identity.idp
          + "'; assertion='" + identity.name + "'");
5  }
6  else {
7     alert("Identity of the peer has not been verified"
          );
8  }
9
10 pc.onidpvalidationerror = function (ev) {
11    /* fired when associated IdP encounters error
          while validating an identity assertion */
12 };
13
14 pc.onpeeridentity = function(ev) {
15    /* fired when identity assertion from a peer has
          been successfully validated */
16 };
```

Listing 3: Example of the WebRTC APIs to retrieve the remote identity as well as handlers to be informed about the validity of the endpoint.

3. SECURITY ANALYSIS

3.1 Scope of the analysis

As briefly sketched in Section 2, the communication security at the network and transport layer (i.e. HTTPS for the signaling plane and DTLS/SRTP for the media plane) have been well thought out, and they provide a decent and sufficient level of confidentiality and integrity to take the network attacker out of scope. As long as the network attacker does not have access to the TLS and DTLS certificates, the network attacker can not eavesdrop or tamper with the data in transit.

In this paper, we therefore focus on the application layer, and in particularly how malicious JavaScript can undermine the security properties of the peer-to-peer connection. More precisely, the security question that will be answered in this paper is:

> *Does WebRTC provide endpoint authenticity guarantees for the peer-to-peer connection? And if not, what are the necessary prerequisites to achieve assurance of the endpoint's authenticity?*

The security guarantees offered by the DTLS/SRTP connection depend on the verification of the self-signed DTLS certificates. In WebRTC, the fingerprints of these certificates are exchanged via the signaling plane as part of the SDP offers and answers. Moreover, in case an Identity Provider is used, the user identity and the fingerprint are securely bound together as part of the identity assertion, and this identity assertion is also attached to the SDP offers and answers.

As such, to achieve authenticity of the endpoint's browser, it is necessary that the private key used in the DTLS connection is kept confidential in the browser, and that the integrity of DTLS certificate fingerprint is preserved in the transfer between the two browsers. Moreover, as self-signed certificates are being used, it is necessary to bind the end-user's identity to the DTLS certificate fingerprint to be able to authenticate the endpoint's user of the peer-to-peer connection.

The prerequisites for endpoint authenticity are:

1. The confidentiality of the DTLS private keys
2. The integrity of the DTLS certificate fingerprint
3. The integrity of the identity assertion, and its binding to the certificate's fingerprint

The first prerequisite is achieved by default, as none of the WebRTC APIs is exposing the DTLS private keys to the JavaScript context. Under what conditions the other two prerequisites can be achieved will be discussed in more detail in the following sections.

3.2 Threat model

To enable the exchange of the certificate fingerprints and identity assertions between the browsers, the SDP information is exposed to the JavaScript context as a JSON object. Typically, this JSON object is marshalled in the JavaScript context of the sending browser, transmitted via the signaling server and unmarshalled at the JavaScript context of the receiving browser.

As a result, any malicious party operating along the signaling path is able to observe and tamper with the SDP information. In particular, we take into account two potential attacker models: the malicious signaling server, and the malicious third-party JavaScript provider.

Malicious Signaling Server. The signaling server mediates the SDP description and ICE candidates between the endpoints. Malicious server-side code can easily tamper with the JSON objects or their string representations.

In addition, the signaling server deploys the client-side JavaScript code which contains the event handlers to create and process SDP descriptions and ICE candidates. Each of these handlers can be coded to tamper with the fingerprints and/or identity assertions before passing on.

Note that this attacker model also include a special case, in which the signaling server operates benign, but is vulnerable to injection attacks, i.e., Cross-site Scripting (XSS). In such cases, the injected JavaScript code will operate as if it is originating from the signaling server and will have the same permissions.

Malicious 3rd Party JavaScript Provider. Third-party JavaScript that is included on the WebRTC webpage runs in the same security context as the client-side JavaScript code of the signaling server, and therefore has the same permissions.

Malicious third-party JavaScript can wrap or replace the WebRTC APIs as well as functions in the client-side JavaScript code, and by doing so tamper with all parameters that are passed to these functions, or trigger new calls to the underlying WebRTC APIs.

Given the threat model above, we will report in Section 4 how an attacker can violate the prerequisites for endpoint authenticity, and as a consequence can route the real-time streams via an intermediate node under the control of the attacker.

In Section 5, we will propose and discuss mitigation strategies in order to preserve endpoint authenticity. This includes security improvements for the WebRTC specifications (and the browser implementations), and mitigation techniques for the Identity Providers as well as the website owners, offering WebRTC services.

4. OVERVIEW OF ATTACKS AGAINST ENDPOINT AUTHENTICITY

In this section, we discuss the various ways in which the prerequisites for endpoint authenticity can be broken by a malicious signaling server or malicious third-party JavaScript. To remain concise, we will illustrate the attacks with malicious client-side code of the signaling server, but bear in mind that this can equally be substituted by malicious third party JavaScript.

This section covers three different, novel attacks against endpoint authenticity. In Section 4.1, the integrity of the DTLS certificate is compromised in WebRTC setups where no Identity Provider is present. This first scenario is very plausible as at the time of writing (most) browsers do not yet provide wide support for IdP integration.

In a second attack scenario, the Identity Provider is tricked is linking the user's identity with a certificate fingerprint under the control of the attacker Section 4.2.

Finally, in Section 4.3, the identity of the user is replaced by an identity under the control of the attacker without the end-users noticing, due to the lack of UI controls.

4.1 Compromising the integrity of the fingerprint (in the absence of an Identity Provider)

The WebRTC specification does not require the use an Identity Provider within a WebRTC setup. Actually, the default operation of WebRTC instances at this moment is without the involvement of an Identity Provider, as the support for IdP integration in browsers is unfortunately not yet mainstream.

In the absence of an Identity Provider, the endpoint authenticity is boils down to the integrity of the DTLS certificate fingerprint within the SDP object. Concretely, this means that in the absence of an Identity Provider the endpoint authenticity can easily be compromised. Every party on the signaling path is able to manipulate the SDP objects and mangle with fingerprints present in the SDP description.

Example attack. Consider for instance the attack scenario, presented in Listing 4. This is a fragment of the client-side JavaScript code, pushed by a malicious signaling server. In this code example, the *createOffer* function gets replaced by a wrapper function, which replaces the SDP offer by a fake SDP object, retrieved from the attacker website via XHR.

```
1  // pc is an RTCPeerConnection object
2  pc.createOfferOriginal = pc.createOffer;
3  pc.createOffer = function(callback, error){
4     pc.orgCallback = callback;
5     pc.malCallback = function(offer){
6        var newOffer = getAttackerSDPViaXHR();
7        pc.orgCallback(newOffer);
8     };
9    pc.createOfferOriginal(pc.malCallback, error);
10 };
```

Listing 4: Example attack to compromise the certificate fingerprint by replacing the SDP offer with an attacker-controlled version.

The SDP offer is represented via a string, and the fake SDP offer will include a new attacker-controlled fingerprint, as well as other vital parameters (e.g. network configuration) to connect to an attacker-controlled endpoint.

As this first class of attacks compromises the integrity of the DTLS certificate fingerprint, the endpoint authenticity can not be guaranteed in the absence of an Identity Provider, given the presence of a malicious actor on the signaling path.

4.2 Tricking the Identity Provider in compromising the integrity of the identity assertion

In case an Identity Provider is being used, the DTLS certificate fingerprint and the user identity are bound together by the identity assertion, and the validity of this pair can be verified by the remote endpoint. In order to compromise the endpoint's authenticity, it is therefore necessary to present a valid identity assertion, which includes the fingerprint of a DTLS certificate under the control of the attacker.

In this subsection, we try to trick the Identity Provider in generating a valid identity assertion, based on fake input (i.e. a certificate fingerprint under the control of the attacker).

Because of the loose coupling, the identity provisioning in WebRTC is realised by loading a *IdP Proxy* in a realm (or hidden iframe), and use Web Messaging to communicate with the *IdP Proxy*. Both mechanisms are readily available for any JavaScript running in the browser, and as such, the loading and interacting with an Identity Provider can also be reproduced outside the context of a WebRTC connection setup.

In order to work, this attack assumes that the Identity Provider does not sufficiently verifies the *origin* of the Web Messages that trigger assertion generations. As stated in the WebRTC specifications, the *origin* of Web Messages used in the context of WebRTC is bound to *rtcweb://*, and it is the responsibility of the *IdP Proxy* to restrict its API access to this origin.

Although this attack can easily be mitigated by Identity Providers complying to this requirement, we expect a large quantity of Identity Providers to oversee the impact of this verification, and hence be vulnerable for this attack. Similar vulnerabilities, affecting Webmessaging, have been identified frequently in the past [20, 7]. Unfortunately, at the moment of writing, no Identity Providers are yet available, so we can not assess to what extend this attack is applicable in the wild.

Example attack. Listing 5 illustrates the basic concept of the attack. In a first step, the *IdP Proxy* is loaded. Next, Web Messaging is used to send a sign request to the *IdP Proxy*, and to receive back the identity assertion.

```
window.addEventListener("message",
    myFakeSignatureReceiverHandler);
var ifr = document.createElement("iframe");
ifr.src = "https://vuln-idp.com/.well-known/idp-proxy
    /oauth";
ifr.style.visibility = "hidden";
ifr.style.display = "none";
ifr.id = "frame1";
ifr.onload = function(){
   var ifr = document.getElementById("frame1");
   ifr.contentWindow.postMessage({"type":"SIGN", "id"
       :1, "message":"
       MY_FAKE_FINGERPRINT_AND_OTHER_INFO_TO_SIGN"});
};
```

Listing 5: Example attack showing how to lure an IdP to sign a fake SDP.

4.3 Compromising the integrity of the identity assertion (due to lack of UI controls)

The WebRTC security model stipulates strict requirements about the consent that is required from end-user from giving an origin access to media devices, such as the camera and the microphone. However, this is also the only user consent that is required to use WebRTC. No UI requirements are stipulated for the browsers to inform the end-user about the fact that a WebRTC connection is being set up, or that local network information is pulled via ICE candidates, or that an Identity assertion is generated or verified by the JavaScript code.

Especially the lack of chrome UI to select a preferred identity or Identity Provider, and the lack of granting access to a specific identity to set up a remote WebRTC connection undermines the integrity of the identity assertion used in WebRTC.

Even in case an Identity Provider is used to set up the peer-to-peer connection, and the fingerprint is correctly bound to an identity in the identity assertion, this could still compromise the endpoint's authenticity.

For instance, if the attacker provides a valid identity assertion for an identity and a DTLS certificate fingerprint (both under the control of the attacker), the only way this can currently be detected is by additional custom-made checks in the client-side JavaScript code. After all, the browser will receive a valid identity assertion from the signaling server, and happily verify the assertion with the Identity Provider, used by the attacker.

Since we consider the signaling server (or the third-party JavaScript provider) to be malicious, we can not expect the client-side JavaScript code to perform the additional checks.

Example attack. The JavaScript context can simply change the identity string with any string of its choice. This could be a fake identity signed by a malicious/compromised IdP or a legitimate identity under the control of the attacker. Listing 6 illustrates how the current identity assertions can easily be replaced by identity assertions generated for artifacts under the control of the attacker.

```
// pc is a RTCPeerConnection object
// hjMc is a MessageChannel object
hjMc.port1.onmessage = function(e) {
  newOffer.sdp = changeAllIdentities(e.data,hjMc.
      offer.sdp);
  pc.trueCallback(newOffer);
};
function changeAllIdentities(newIdentity,sdp){
  identityExtraction = base64(newIdentity);
  return sdp.replace(/identity:[A-Z0-9]*\n/g, '
      identity:'+identityExtraction);
};
```

Listing 6: Example attack showing how to modify the identity string to a fake identity.

5. MITIGATION STRATEGIES

In this section we will describe mitigation techniques and security advises for real-life deployments to counter each of the attacks described in Section 4.

Although on a network architecture everything seems properly encrypted, the fact that the meta-data and control messages are exposed to the signaling servers as well as to JavaScript code running in the origin of the signaling website (independent if coming from the signaling server or from any third-party), a malicious signaling server or untrusted third-party JavaScript code can compromise the endpoint authenticity in WebRTC peer-to-peer connections.

In such a security model, only the browser implementation is considered trusted, and the JavaScript code on top is treated as untrusted. In particular, the endpoint authenticity can only be guaranteed if all the the security guarantees about the authenticity of the endpoint are based only on decisions within the browser implementation, irrespective of the JavaScript that triggered the API calls or handled WebRTC events.

This aligns with the following properties:

- Under no circumstances, keying material for the DTLS should be allowed to be added to the browser or extracted from the browser run-time via the JavaScript execution context.
- At all times, the DTLS certificate fingerprint (needed to set up the DTLS connection) needs to be used in combination with the identity assertion to guarantee that actors outside the browser implementations can alter the certificate fingerprint. *[Attack 1]*
- It must be impossible to trigger the IdP to sign custom-crafted data. The IdP proxy can only generate identity assertions based on artifacts, trusted by the browser implementation. *[Attack 2]*
- The end-user must be involved in granting access to a specific identity in setting up a WebRTC connection. *[Attack 3]*

If we can realize these properties, we can use the untrusted signaling path and still achieve endpoint authenticity. In the rest of this section we give, for each stakeholder, an overview of its responsibilities to preserve this property.

5.1 Browser Vendors & Specifications

Firstly, the WebRTC specifications should reconsider the optional use of an Identity Provider. A possible scenario could be derived where the signaling server also operates an Identity Provider, as this would prevent third-party JavaScript from compromising the integrity of the DTLS certificate fingerprint. This would directly counter the attacks described in Section 4.1.

Secondly, the browser should prevent that the IdP Proxy can be loaded as part of a Webapplication. By making the loading exclusive to the WebRTC identity provisioning mechanism, Identity Providers can not be tricked by application-level code to generate identity assertions for the end-user under attack, as was done in Section 4.2.

Thirdly, the specifications should state the browser should provide the necessary UI chrome to enable users to select an appropriate identity of their favorite Identity Providers, and, even more important, enable them to only grant access to remote identities of their choice to set up a peer connection.

In case one wants to free the end-user from being involved in giving user consent, the end-user should at least be informed by the browser about the identity of the peer-to-peer connection. This would at least make attacks as shown in Section 4.3 detectable by end-users.

5.2 Identity Providers

To prevent attacks as presented in Section 4.2, the Identity Provider must make sure that the *IdP Proxy* can only be invoked directly from the browser implementation, and not from the application-level JavaScript code. To do so, the Web Messaging code in the *IdP Proxy* needs to verify that all incoming messages originate from origins with the *rtcweb://* scheme, a scheme that is exclusively used for WebRTC purposes.

This is in line with the more general recommendations for the use of Web Messaging, for which the API specification states that the developers "*SHOULD check the origin attribute to ensure that messages are only accepted from domains that they expect to receive messages from*" [8].

5.3 Website Owners

In this subsection, we consider a benign website owner (also known as the signaling server) that wants to protects its users against the attacks illustrated in Section 4, carried out by third-party script providers or via script injection.

Firstly, the website owner needs to ensure that in all cases an Identity Provider is used. If no external Identity Provider is needed, the website owner can deploy his own *IdP Proxy*, that for instance piggy backs on the session mechanism for the website. By enabling an Identity Provider in all scenarios, the attack from Section 4.1 can be mitigated.

Secondly, the website owner needs to protect its application against all forms of code injection (either via XSS or via untrusted third party JavaScript). Even if the website is operated by a benign stakeholder, code injection attacks can trigger all three attacks.

As stated in [16], there should be no third-party JavaScript allowed on the same page that hosts the application-specific code. This can be achieved by running the WebRTC application in a separate origin or subdomain. Moreover, techniques such as the Content Security Policy (CSP) [21] can be used to prevent third-party resources from being loaded in the website. In cases where third-party scripts are required, one can strictly limit the loaded scripts with CSP, and deploy JavaScript encapsulation, freezing and sandboxing techniques to protect the application code and WebRTC APIs, which might not be trivial [13, 12, 1, 9].

Thirdly, the website owner could take advantage of the fact that all meta-data and control messages are routed via the signaling server. The website owner should verify that the fingerprints/identies passing by exactly match the fingerprints/identities expected to be used in the application. This can help to detect and mitigate attacks from Section 4.2 and Section 4.3.

6. RELATED WORK

In [16] and [15], Rescorla describes WebRTC's underlying security architecture and discusses in depth relevant aspects in the areas of network security and identity management. In [2], Barnes et al. state that the calling site might not (always) be trusted, and they identify a crucial role for the browser as trusted element.

Several researchers on WebRTC security specifically focuses on the complex problem of identity provisioning in WebRTC based applications [5, 3, 11]. Beltran et al. study a variety of trust models for two-party communication, and investigate the use of existing Identity Provider solutions

such as BrowserID, OAuth 2.0 and OpenID Connect in the context of WebRTC [3]. Li et al. investigate identity provisioning in more complex deployments, in which multiple calling sites and Identity Providers are involved, and suggests the use of a Web-of-Trust identity model to overcome the limitations of hierarchical identity systems [11].

Finally, the Tin Can project at Mozilla uses Persona to authenticate [19]. As part of the prototype, end-users are informed about the identities of remote parties, and can grant access to individual remote identities to set up a call. This solution overcomes the UI security challenges, described in Section 4.3.

7. CONCLUSION

Unlike closed identity ecosystems, as provided by commercial video chat offerings, open systems such as WebRTC are potentially prone to identity fraud, due to WebRTC's emphasis on identity federation. In this paper, we explored the technical underpinnings of WebRTC's identity management mechanism. Based on our analysis, we identified three novel attacks against endpoint authenticity. Each of our attacks allows a malicious party, which is able to execute JavaScript in the context of the hosting Webapplication, to fully manipulate the identity assertions that are attached to a WebRTC data channel, thus, enabling the spoofing of the communication endpoint. We proposed a set of mitigation strategies, spread over several target groups including Webapplication providers, browser vendors, and the specification committees.

8. ACKNOWLEDGEMENTS

We thank the anonymous reviewers for their valuable feedback, and the EU-FP7 STREWS project consortium for leading and contributing to the WebRTC security assessment (reported in [14]).

This research is partially funded by the Research Fund KU Leuven, the Federal Science Policy, the iMinds-ICON project MECOVI, and the EU-funded FP7 project STREWS. Willem De Groef holds a Ph.D. fellowship from the Agency for Innovation by Science and Technology in Flanders (IWT).

9. REFERENCES

[1] Pieter Agten, Steven Van Acker, Yoran Brondsema, Phu H Phung, Lieven Desmet, and Frank Piessens. JSand: Complete client-side sandboxing of third-party JavaScript without browser modifications. In *Proceedings of the 28th Annual Computer Security Applications Conference*, pages 1–10, 2012.

[2] R.L. Barnes and M. Thomson. Browser-to-browser security assurances for webrtc. *Internet Computing, IEEE*, 18(6):11–17, Nov 2014.

[3] V. Beltran, E. Bertin, and N. Crespi. User identity for webrtc services: A matter of trust. *Internet Computing, IEEE*, 18(6):18–25, Nov 2014.

[4] Adam Bergkvist, Daniel C. Burnett, Cullen Jennings, and Anant Narayanan. WebRTC 1.0: Real-Time Communication Between Browsers. *W3C Editor's Draft*, 2015.

[5] Lieven Desmet and Martin Johns. Real-time communications security on the web. *Internet Computing, IEEE*, 18(6):8–10, Nov 2014.

[6] ECMA. Draft ECMAScript 2015 Language Specification (RC4). [online], http://people.mozilla.org/~jorendorff/es6-draft.html, April 2015.

[7] Steve Hanna, Eui Chul, Richard Shin, Devdatta Akhawe, Arman Boehm, Prateek Saxena, and Dawn Song. The emperor's new apis: On the (in) secure usage of new client-side primitives. In *Web 2.0 Security and Privacy (W2SP 2010)*, 2010.

[8] Ian Hickson. HTML5 Web Messaging. *W3C Recommendation*, 2015. http://www.w3.org/TR/webmessaging/.

[9] Lon Ingram and Michael Walfish. Treehouse: Javascript sandboxes to help web developers help themselves. In *Proceedings of the USENIX annual technical conference*, 2012.

[10] A. Johnston, S. Donovan, R. Sparks, C. Cunningham, and K. Summers. Session Initiation Protocol (SIP) Basic Call Flow Examples. RFC 3665 (Best Current Practice), December 2003.

[11] Li Li, Wu Chou, Zhihong Qiu, and Tao Cai. Who is calling which page on the web? *Internet Computing, IEEE*, 18(6):26–33, Nov 2014.

[12] Jonas Magazinius, Phu H. Phung, and David Sands. Safe Wrappers and Sane Policies for Self Protecting JavaScript. In *Information Security Technology for Applications - 15th Nordic Conference on Secure IT Systems, NordSec 2010*, volume 7127 of *LNCS*, pages 239–255. Springer, 2010.

[13] Phu H. Phung, David Sands, and Andrey Chudnov. Lightweight self-protecting JavaScript. In *Proceedings of the 4th International Symposium on Information, Computer, and Communications Security*, ASIACCS '09, pages 47–60, New York, NY, USA, 2009. ACM.

[14] STREWS project consortium. D1.2 Case Study: Security Assessment of WebRTC. Technical report, 2014. https://www.strews.eu/images/webrtc.pdf.

[15] E. Rescorla. Security Considerations for WebRTC. Internet-Draft draft-ietf-rtcweb-security-08, Internet Engineering Task Force, February 2015. Work in progress.

[16] E. Rescorla. WebRTC Security Architecture. Internet-Draft draft-ietf-rtcweb-security-arch-11, Internet Engineering Task Force, March 2015. Work in progress.

[17] E. Rescorla and N. Modadugu. Datagram Transport Layer Security Version 1.2. RFC 6347 (Proposed Standard), January 2012.

[18] J. Rosenberg and H. Schulzrinne. An Offer/Answer Model with Session Description Protocol (SDP). RFC 3264 (Proposed Standard), June 2002. Updated by RFC 6157.

[19] Ryan Seys. Mozilla Tin Can, 2015.

[20] Sooel Son and Vitaly Shmatikov. The Postman Always Rings Twice: Attacking and Defending postMessage in HTML5 Websites. In *Network and Distributed System Security Symposium (NDSS'13)*, 2013.

[21] Mike West, Adam Barth, and Dan Veditz. Content Security Policy Level 2. W3C Working Draft, W3C, July 2015. Work in progress. http://www.w3.org/TR/2015/CR-CSP2-20150721/.