Browser-to-Browser Security Assurances for WebRTC

For several years, browsers have provided assurance that users are talking to a specific, identified website, protected from network-based attackers. With email, messaging, and other applications for which sites act as intermediaries, additional protections are needed to provide end-to-end security. Here, the authors describe how Web Real-Time Communications provide end-to-end security, leveraging both the flexibility of JavaScript and browsers' ability to create constraints through JavaScript APIs.

Richard L. Barnes and Martin Thomson
Mozilla

For many years, a browser's lock icon has indicated a secure connection. Behind this icon is HTTPS — HTTP over the Transport Layer Security (TLS) protocol.[1–3] When a user's browser connects via HTTPS, the icon indicates that he or she is visiting the indicated site, and that the site has authorized anything the browser displays.

This is a reasonable security model in two-party, client-server interactions; it's less satisfactory in more complex interactions such as email or social networking sites, where the server is mediating communications between users. In these cases, users must trust the server not to fake or modify their messages, disclose content to unauthorized third parties, or mine messages for personal information. Although this trust is sometimes warranted, sometimes it isn't, and it's not something users can verify.

Today, calls made with Web Real-Time Communications (WebRTC) are in basically the same situation: because they are server-mediated, users must trust a site to handle their calls in accordance with their wishes. As with email, this is acceptable in many situations, either because the user is actually calling the site (as with customer service applications) or because their calls aren't so sensitive that they don't trust the site with them. However, in other cases, such calls *are* sensitive, and users want to know that they're secure even from the site. In light of recent revelations about intelligence agencies acquiring information from websites, it's worth trying to build an ecosystem in which sites that provide WebRTC calls don't have to be trusted with those calls' security.

To achieve this, we need security mechanisms in the browser that can control the two capabilities sites currently have: authenticating calls and accessing media. In initial WebRTC implementations, the site is responsible both for directing the call to the proper remote party and for verifying that party's identity

1089-7801/14/$31.00 © 2014 IEEE Published by the IEEE Computer Society

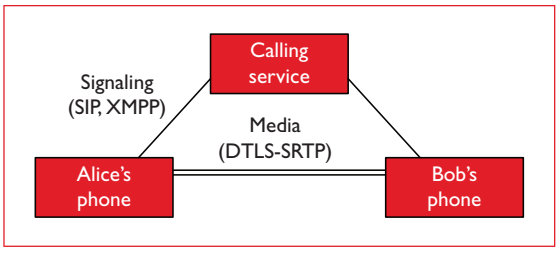

Figure 1. Structure of a real-time session. Alice and Bob set up a call via a calling service, then exchange media directly between the two endpoints.

during call setup (to address man-in-the-middle attacks on media). Controlling this capability requires a system for authenticating users that's independent of the calling site. In addition, such sites can currently access users' audio and video data through WebRTC APIs; controls on this access will need to be realized through these APIs. WebRTC specifications have added mechanisms to address these risks, and these mechanisms are making their way into browsers.

Addressing Threats in the Peer-to-Peer Web

In typical Web applications, security is mainly treated as a point-to-point matter. A user tells his or her browser to connect to https://example.com. The browser establishes a TLS session with a server that claims to be example.com, and uses the credentials the server has provided to verify that it legitimately represents example.com. TLS provides confidentiality and data integrity. The browser provides authentication by verifying a credential, through which some authority (called the *identity provider*, or IdP) asserts a binding between the server's public key and an identity. The authenticated entity is the logical "origin" of the content and code needed to render the page. These resources are fetched either directly from the authenticated entity or from sources it authorizes (as with mashups, content delivery networks, and so on).

In the peer-to-peer Web enabled by WebRTC, the webpages that users visit are only a means to get to the entities they really want to communicate with. This "rendezvous" service is critical for RTC because connectivity can change. It also means that a site can connect you to the wrong person. This is actually a familiar problem from other settings. With email, for example, mail servers let users get messages even with intermittent connectivity, but in almost all email transactions, the server is trusted to deliver the message to the correct user.

Real-time applications are commonly represented as a triangle, with Alice and Bob at two corners, and a calling service (such as Skype or a Session Initiation Protocol, or SIP, proxy) at the other (see Figure 1). Alice and Bob set up a call via a calling service, then exchange media directly between the two endpoints.

This maps directly to the common WebRTC case, where two website users employ WebRTC to talk to one another. The calling service is simply a website, and signaling is carried over HTTPS.

In WebRTC, we must consider an additional level of detail. WebRTC relies on JavaScript code from the calling site to create a user interface and control the media flow. When we add these elements, the picture looks similar to a voice-over-IP (VoIP) scenario (see Figure 2), with gateways for signaling and media, but with a twist: these gateways are represented by JavaScript code.

Alice and Bob participate in the WebRTC application by downloading some HTML and JavaScript from the server. The JavaScript creates the WebRTC application by using the WebRTC API to acquire and render media streams and create connections to other peers. In addition to providing the JavaScript, the server usually acts as a signaling server to help the endpoints, Alice and Bob, find each other.

The twist described previously effectively means that adversarial code is running the user's browser, because the server (example.com) is precisely what end-to-end security mechanisms seek to control. In some ways, this is a positive change: the browser is a trusted component in the WebRTC model, so it can enforce constraints on the relay to achieve security properties on the intrabrowser hops.

Let's first consider what level of security we achieve with the mechanisms already present in browsers — namely, HTTPS for signaling interactions and the Datagram Transport Layer Security-Secure Real-Time Transport Protocol (DTLS-SRTP) for media.[4,5] The use of HTTPS on signaling hops, with normal server authentication, addresses network-based attackers between the server and the two browsers. DTLS-SRTP provides a secure channel, and a calling site attests to the browser endpoints' identity.

So, just using the tools in browsers today, we arrive at a fairly high level of security. As long as the user trusts the calling site, all network-based attackers are defeated. WebRTC applications are thus in a similar position to Web-based chat or email applications. In some WebRTC applications,

this is the highest level of security we can expect — for example, in a "click-to-call" customer service application, where the called party is the same entity as the calling site. These cases offer "hop-by-hop" assurances, but the calling site can record or reroute a user's messages or calls.

To achieve end-to-end security without trusting the calling site, we will need new controls on authentication and access to media. For authentication, we reintroduce IdPs. These will play a similar role as they did in the client-server Web context, but we want the authentication to be symmetric, so we'll need two IdPs. To control access to media, limitations must be built into the API that the JavaScript code in the calling site uses to manage media streams.

The model of WebRTC threats that we've discussed thus highlights a few necessary security mechanisms:

- signaling traffic protection (between each browser and the server);
- media traffic protection (between browsers);
- end-to-end authentication of the communicating peers; and
- protection of media streams from the JavaScript code that handles them.

As noted, the first two have been addressed using HTTPS and DTLS-SRTP, respectively. We discuss the latter two in the following sections.

Cryptographic Identity

In considering end-to-end authentication for WebRTC, we immediately run into a problem: unlike the standard public-key infrastructure (PKI) mechanisms that HTTPS uses in client-server transactions, the Web has no universal user credential system, nor is one likely to arise soon. Which technology should WebRTC use? What trust model should it be based on? How can we allow for strong identity as well as anonymity and tracking avoidance?

The current WebRTC security architecture uses some artful indirection to avoid needing a single answer to the first two questions. It addresses the overall identity problem in two steps: first, it verifies that the remote party holds a given key pair; second, it verifies the identity of who holds that key pair. The first problem is solved with the DTLS-SRTP handshake; the browser simply reports that the handshake failed or, if it succeeded, which public key was presented. The second problem requires an identity layer.

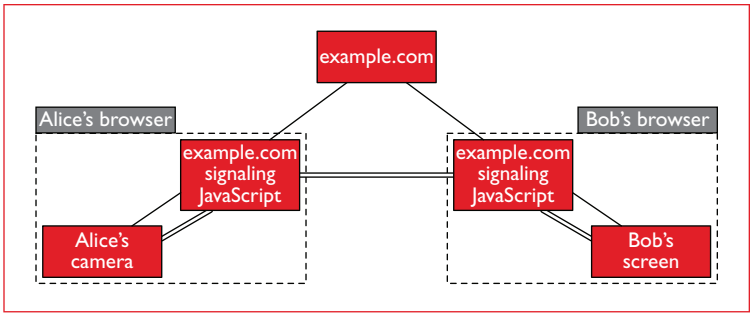

Figure 2. A real-time session mediated by JavaScript. WebRTC relies on JavaScript code from the calling site to create a user interface and control the media flow.

Any identity system has two key parts: a way for IdPs to generate assertions binding a public key to an identity, and a way for relying parties to verify these assertions. In a PKI, for example, an IdP asserts a name/key binding by signing a certificate; a relying party verifies the assertion by verifying that the certificate chains back to the trust anchor.[6]

As befits a Web-based system, the WebRTC identity layer implements specific identity systems with JavaScript. When two parties establish a connection, each one downloads some JavaScript code for generating assertions. This code verifies the user's identity (in a provider-specific way) and generates an identity assertion (in a provider-specific format), binding the user's identity to the public key used in the DTLS-SRTP session. The JavaScript code serves mainly as the interface between the browser and server-based signing services that the IdP operates; obviously, assertions must be authenticated using some secret value specific to the IdP.

The WebRTC app's signaling conveys to the other party an assertion tagged with a domain name for the IdP (see Figure 3). To verify an assertion, the receiving browser downloads the IdP's assertion-verification code (from a well-known URI built from the domain name) and asks it to verify the assertion it received from the signaling layer.

Of course, in call setup, this happens twice — once to authenticate Alice and once to authenticate Bob. The browser mediates all of these IdP actions via the PeerConnection AP, which ensures that the signaling JavaScript has only limited control over which key pair is used for DTLS and, thus, which key pair the IdP will generate an assertion for. In particular, although the JavaScript might be

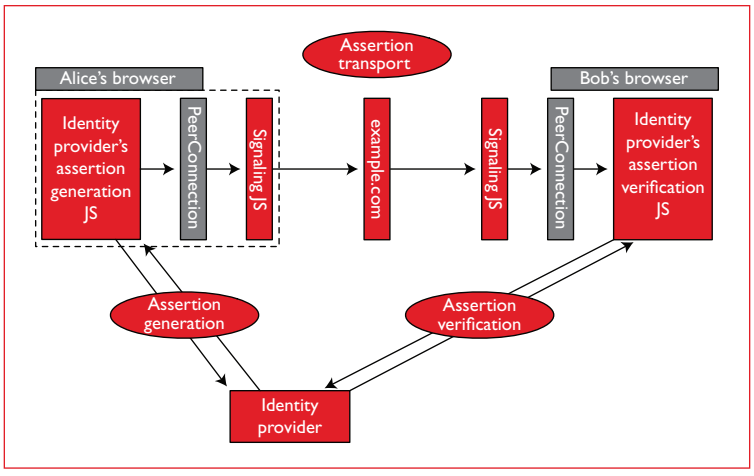

Figure 3. Lifecycle of a WebRTC identity assertion. The WebRTC app's signaling conveys to the other party an assertion tagged with a domain name for the identity provider. To verify an assertion, the receiving browser downloads the IdP's assertion-verification code and asks it to verify the assertion it received from the signaling layer.

able to request a fresh key, it can't impose a key of its choosing or extract the private key. This assures the IdP that the assertions it generates will be used only for WebRTC and not for other purposes.

The abstraction in the WebRTC identity layer gives it considerable flexibility. As noted, it can incorporate a wide range of identity systems, including X.509, OpenID,[7] OAuth,[8] and Persona (https://login.persona.org/about). In principle, any Web server can act as an IdP as long as it provides JavaScript for assertion generation and verification. This level of openness means that WebRTC isn't bound to a single trust model. Whatever your favorite trust model is, you can make an IdP that realizes it. The notion of an IdP maps naturally onto authority-based identity models (with the IdP as the authority). However, the IdP could just as well provide code to realize a more distributed model, one with traceable anonymity, and so on.

One important property of all WebRTC identities is that they are scoped — that is, they all have the form "user@domain," where "domain" identifies the IdP, and "user" is the identity that provider asserts. This constraint rules out using unscoped identifiers such as domain names or telephone numbers within the WebRTC identity system. However, it avoids all of the thorny issues surrounding authority that arise when we use these identifiers (such as which IdP may make assertions about a given phone number). Instead, an IdP can assert any identities it likes, but all within its own scope. (Of course, the ability to use phone numbers and other unscoped identifiers is desirable in many cases; additional mechanisms would need to specify how to authenticate these identifiers, as with the work being done in the IETF's Secure Telephone Identity Revisited working group[9]).

The other positive consequence of using only scoped identities is that the browser need not choose which IdPs are trusted — it just discloses which IdP vouched for a given identity. This effectively reduces the endpoint authentication problem to the same set of problems that websites already face. The browser will verify that the person you're talking to has the identity teller@bank0famerica.com, but it won't necessarily be able to tell you that "bank0famerica.com" doesn't actually represent Bank of America.

One might anticipate applying antiphishing techniques or techniques similar to extended validation certificates here, but some developers working on WebRTC stacks have an even simpler vision. A user's address book is effectively a list of identities that the user has directly verified (for example, Dad is johnsmith@example.com). If the browser can compare the scoped identity from the WebRTC stack with the user's address book, then it can present identity information that the user has already verified.

The calling site's JavaScript code is in charge of configuring the identity process, in that it chooses whether to use identity and which IdP to use. This latter choice doesn't allow identity spoofing because scoping WebRTC identifiers means that if the site chooses a different IdP than the user, a different identity will be asserted. The choice of whether to use identity only lets the site strip it — in telephone parlance, to block caller ID. Browsers can guard against both of these abuses. Address book integration will help with IdP switching, and the more common authentication is, the more stringent browsers can be in their warnings to users about unauthenticated calls.

WebRTC thus handles end-to-end authentication in layers. First, the DTLS-SRTP layer associates the remote party with a public key. Then, the identity layer maps that public key to a scoped identifier by way of JavaScript-based IdPs. Finally, the browser user interface can map the authenticated identifier to rich identity information from sources such as a user's address book.

Local Access Controls

In WebRTC, the main function of the website that creates the application is rendezvous. When Alice wants to call Bob, the site figures out how to reach Bob right now, in real time. The identity layer protects against the site performing this function incorrectly (connecting Alice to someone other than Bob). That level of assurance protects Alice and Bob from attackers in the network, both passive listeners and active redirection attacks from the website. However, as noted, in the WebRTC model, threats exist inside the browser — the JavaScript code that came from the same server that the identity layer protects against.

For WebRTC apps to work, the JavaScript code from the signaling server must be able to acquire media streams from input devices and the network, and connect them to facilities that render or send them across the network. Using JavaScript for this "wiring" is central to WebRTC's flexibility, but it can also let the JavaScript access potentially sensitive media — for example, record camera or microphone output, or even manipulate a user's voice or image.

JavaScript has legitimate use cases to do these things. Some calling services might want to offer call recording as part of their WebRTC application. Video processing in JavaScript can be used for a range of applications, from ones that insert funny background images to those that create composites for video conferences. However, in all these cases, users must trust the calling site, and they won't receive assurance that the only party with access to the media is the authenticated remote party.

How about the scenario in which the site wants to provide this harder assurance? In the current draft API,[10] the site explicitly relinquishes access to the media. As usual, the site requests access to media streams (cameras, microphones, and so on), but adds a *peerIdentity* option to its request. Effectively, instead of asking, "Could I please have the output of the camera?" the site says, "Could I please send the camera's output to bob@example.com?" This lets the browser display different user interface permissions to the user, indicating Bob instead of the site as the media recipient. If the user grants access, the site receives an "isolated stream," an opaque handle to the media stream that it can use for nothing but wiring — that is, sending media to Bob or displaying media in a video element. If the site tries to access the media stream content, the API will throw an exception.

An astute reader might note that untrusted JavaScript exists on both sides of a WebRTC connection, so assurances from the API would need to be enforced in both browsers. How does Alice know that the JavaScript in Bob's browser isn't recording the call, even if the JavaScript in her browser is precluded from doing so? At one level, Alice wouldn't really benefit from this assurance; Bob is a trusted entity in our threat model, so he could be recording the call in any case, whether with the JavaScript in his browser or otherwise. Nonetheless, it would reduce the burden on Bob to verify the security properties of his JavaScript if Alice's browser could communicate to Bob's browser that certain media streams should be isolated. The IETF RTCWEB and TLS working groups are currently defining such a signal.

A similar problem exists as regards managing the cryptographic keys used to protect WebRTC connections. On one hand, the identity system hides most of the identity process details from the calling site, as discussed. On the other, the calling site might legitimately want to control certain aspects of the authentication process — most importantly, the frequency with which new cryptographic keys are generated for use with real time sessions.

Using the same key for multiple WebRTC sessions can provide a useful form of lightweight identity via key continuity. This authentication mode is also known as the Secure Sockets Shell (SSH) model, or "trust on first use" (TOFU). Conversely, public keys can leak identity information if the same key is used in different contexts. For example, suppose that a browser generated a single key pair per calling site. If a user then used that calling site to place calls to the same endpoint with two different identities, from two different IdPs, then the called party would be able to recognize that the two identities are actually on the same computer (because they both have access to the same private key). In fact, if the two IdPs compared notes, they would be able to come to the same conclusion (as sometimes occurs with cookies today).

The questions of how much control a calling site or IdP should have over the key pairs and how to realize that control in the API are currently under discussion in the W3C WebRTC working group. The group is considering two main types of control: let the site learn the public key that will be used on a given connection, or let it request a new key for a connection.

It isn't acceptable for the site to dictate that a specific key pair be used, because this violates the contract made with IdPs that the assertions they issue will be used only for WebRTC. So, the main question is how to express these two capabilities.

By interposing API controls, WebRTC can provide a balance in the capabilities offered to untrusted JavaScript code. On one hand, it provides sufficient capabilities to construct a wide variety of different applications and protect users from tracking; on the other, it can prevent JavaScript from accessing user media or keying material.

Even with these restrictions, though, one more significant gap remains — that between secure media and rendered media. Suppose that the signaling JavaScript can set up a legitimate authenticated session between a user and his or her bank. In the current WebRTC framework, the JavaScript could then ignore the secure media (not rendering it), and play a scam message from some other source. So, if the browser displays any security indicators based on authenticating the session with the bank, it must ensure that these indicators are correctly associated with rendered media.

Many subtleties lurk in this requirement because legitimate reasons exist for a WebRTC application to involve media from different secure sources. For example, a video-conferencing application could display incoming audio and video from several sources, each with different identities, possibly from different IdPs. How should the browser distinguish (or let the user distinguish) such an application from the bank scam, which has media from two different authenticated identities (the bank and the scam source)? Solutions for this problem are currently under active discussion in the WebRTC working group.

As a new platform for RTC, WebRTC offers the opportunity to solve the end-to-end problems that have remained unsolved in email. Historically, establishing good identity end to end has been a major stumbling block. WebRTC implementations' ability to dynamically instantiate different identity systems, together with the fact that several Web-based identity systems already exist, provides some hope that we will be able to overcome this challenge. Although the calling site's JavaScript does create an adversary in the browser, we have the tools in the WebRTC API to prevent these adversaries from interfering with calls.

These mechanisms are being implemented in browsers right now. As they become available, we will see whether an ecosystem of end-to-end secure WebRTC applications develops. What current IdPs can we fold into WebRTC? What new identity systems can we build based on the WebRTC framework? What lessons from securing WebRTC can we extend to other applications? If we can build WebRTC applications that are secure from tampering by intermediaries, we will have succeeded where years of application deployments have failed.

References

1. E. Rescorla, *HTTP Over TLS*, IETF RFC 2818, May 2000; http://tools.ietf.org/html/rfc2818.
2. R. Fielding et al., *Hypertext Transfer Protocol — HTTP/1.1*, IETF RFC 2616, June 1999; www.ietf.org/rfc/rfc2616.txt.
3. T. Dierks and E. Rescorla, *Transport Layer Security (TLS) Protocol Version 1.2*, IETF RFC 5246, Aug. 2008; http://tools.ietf.org/html/rfc5246.
4. J. Fischl et al., *Framework for Establishing a Secure Real-Time Transport Protocol (SRTP) Security Context Using Datagram Transport Layer Security (DTLS)*, IETF RFC 5763, May 2010; http://tools.ietf.org/html/rfc5763.
5. D. McGrew and E. Rescorla, *Datagram Transport Layer Security (DTLS) Extension to Establish Keys for the Secure Real-Time Transport Protocol (SRTP)*, IETF RFC 5764, May 2010; http://tools.ietf.org/html/rfc5764.
6. D. Cooper et al., *Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile*, IETF RFC 5280, May 2008; www.ietf.org/rfc/rfc5280.txt.
7. D. Recordon and D. Reed, "OpenID 2.0: A Platform for User-Centric Identity Management," *Proc. 2nd ACM Workshop Digital Identity Management*, 2006, pp. 11–16.
8. D. Hardt, *The OAuth 2.0 Authorization Framework*, IETF RFC 6749, Oct. 2012; http://tools.ietf.org/html/rfc6749.
9. J. Peterson, "Secure Telephone Identity Problem Statement and Requirements," IETF Internet draft, work in progress, 2014.
10. D.C. Burnett et al., "Media Capture and Streams," W3C editor's draft, 2014; http://dev.w3.org/2011/webrtc/editor/getusermedia.html.

Richard L. Barnes is a security technologist at Mozilla, where he leads the development of public-key infrastructure and other security technologies. Barnes has an MS in mathematics from the University of Virginia. He serves as area director for the Real-Time Applications

and Infrastructure area of the IETF, the standards body responsible for WebRTC's network protocols and security architecture. Contact him at rlb@ipv.sx.

Martin Thomson is a generalist. He works for Mozilla on WebRTC identity and security for Firefox. Thomson received a BS in computer engineering from the University of Wollongong. He's the editor of the IETF specification for HTTP/2 and a contributor to many other projects, including Web Push, Geolocation, and Transport Layer Security. Contact him at martin.thomson@gmail.com.