Research Review of WebRTC Security

Wenqiang Yang

Abstract

This document reviews the security and vulnerability of Web Real-Time Communication (WebRTC) technology. This document first studies the essence of WebRTC, including its API and protocols behind it, then it analyzes the security issues of WebRTC and offers some cases.

Introduction of WebRTC

WebRTC is a web-based peer-to-peer real-time communication technology.

As a web-based technology, WebRTC applications can be developed using HTML and JavaScript, and is supported by all mainstream browsers, including Chrome, Firefox, and Safari. Users can participate in video meeting without installing software manually, and developers can build real time communicating applications with just JavaScript API. It also supports Android and iOS platform, and is used to develop applications for many different devices.

WebRTC offers stable and secure peer-to-peer connection, supporting multiple kinds of data transfer, including audio, video and file transfer.

Due to its convenience and powerfulness, WebRTC is used in various applications, including Google Meet, YouTube live, Slack, Signal, etc.

Architecture of WebRTC

WebRTC can be described with the browser RTC Trapezoid Model[1]. As is shown in Fig. 1, it includes two browsers and two web servers. The client-server path and inter-server path are called signaling path, which transmit signals that are needed for creating, modifying, or terminating sessions[1]. The protocols of signaling path is not specified by WebRTC - they can use existing protocols (SIP or XMPP) or proprietary protocols, over HTTPS or WebSockets[1]. After both browsers have exchanged necessary messages for the peer-to-peer connection, the two browsers are connected directly with the media path, and the peer-to-peer media communication begins.

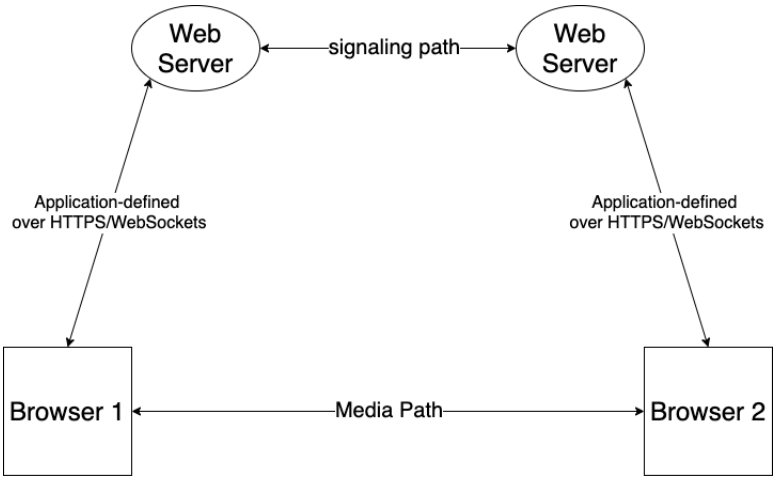

Fig. 1 Web Browser Trapezoid Model

I. Underlying Technology

We'd like to dive a little deeper into the underlying technologies, including the Session Description Protocol, and the NAT traversal process, which helps to understand the details about some security issues like IP leaking issue.

i. Session Description Protocol

Before the direct connection between two users is set up, users need to send a description of their sessions, network setting, etc., and reach an agreement upon them, to find the suitable configuration for the connection. Session Description Protocol (SDP) is the suitable mechanism for this situation.

A standard SDP includes[2]: Session Metadata, Network Description, Stream Description, Security Description, and Qos, Grouping Description. Details can be found in 5th chapter of [3].

ii. NAT

One big barrier to peer-to-peer communication is the widely used technology Network Address Translation (NAT) which is designed to mitigate the scarcity of IPv4. NAT translates the internal IP address and port (local IP and port) to external IP address and port (public IP and port) and vice versa[4]. NAT is also used to conceal privacy of the local network from the external side of NAT.

Based on the treatment of UDP, NAT can be divided into[5]:

- Full Cone NAT: all requests from the same internal IP address are mapped to the same external IP address, and any external hosts can send packets to the internal host, by sending to the same external IP.
- Restricted Cone NAT: the same as full cone NAT, except for that external hosts cannot send a packet to the internal hosts, unless that internal host had sent a packet to the same external host before. And port restricted cone behaves similarly.
- Symmetric NAT: requests from the same internal IP address (and port) and to the same destination IP address (and port) are mapped to the same external IP address (and port). Only external host that receive a packet can send a UDP packet back to the internal host, which means that all unsolicited requests are blocked.

Due to some NAT's blocking or restricting unsolicited requests peer-to-peer connection can be hard to establish — so we need to do NAT traversal.

iii. STUN, TURN and ICE

STUN (Session Traversal Utilities for NAT, RFC5389) and TURN (Traversal Using Relays for NAT, RFC5766) are two widely used NAT traversal tools.

STUN is a client-server model - STUN clients are usually on the internal side of NAT, and STUN servers reside in the public Internet. STUN client can send a Binding request to a STUN server[6]. Each time when the request passes through a NAT, the NAT will translate the source IP address (and port) in the packet. When STUN server receives the request, the source IP address becomes the external IP address (and port) created by the nearest NAT. This is called a reflexive transport address. The STUN server copies the received IP address to the XOR-MAPPED-ADDRESS attribute in the response, and sends it back to the STUN client. The XOR-MAPPED-ADDRESS attribute remains intact when transmitting, so the client can receive its reflexive transport address allocated by the outermost NAT with respect to the STUN server.[6]

TURN can be considered as a relay-enabling extension of STUN, and it is used when both endpoints are behind a symmetric NAT (STUN fails for this situation).

WebRTC implements the Interactive Connectivity Establishment (ICE, RFC5245) protocol, with the help of STUN and TURN, to solve the NAT traversal problem.

The basic idea of ICE[7] is to gather a variety of candidate IP addresses and ports and then test these IP addresses pairs' connectivity by trying all possible pairs in a carefully sorted order. The connectivity check process is also implemented by STUN protocol.

ICE gathers candidates, including[7]:

- Host Candidate: a candidate obtained directly from a local interface, both physical interface, like Wi-Fi and Ethernet, and logical interface, like VPN.
- Server Reflexive Candidate: a candidate obtained by using the Binding request to a STUN server or TURN server.
- Relayed Candidate: a candidate obtained by sending a TURN Allocate request to a TURN server.

II. WebRTC APIs

The browser and web applications uses WebRTC APIs to implement certain functions, such as capturing and recording video and audio, to configure the connection, and to build real-time communication.

Using these API, developers can build WebRTC application, and they can also be implemented to fetch some user's data.

There are three important and frequently used APIs:

a. MediaStream

`MediaStream` includes several APIs to get access to media devices (such as microphone and camera), and APIs to manipulate with the media stream and media stream tracks.

To create a stream, `getUserMedia` is one commonly used API which enables the browser to access to user's camera and microphone. It is supported by many browsers, such as Chrome, Firefox, etc., though it performs slightly differently in different browsers. When `getUserMedia` method is called, the browser prompts the user to allow access to the media device (according to `mediaStreamConstraints` argument[8]).

If the user grants the permission, `getUserMedia` returns a `MediaStream` [8]and the media can be rendered on the HTML.

b. RTCPeerConnection

`RTCPeerConnection` enables applications to establish peer-to-peer connection. Each peer instantiates a `RTCPeerConnection` instance respectively, and set up a connection between them. A `RTCPeerConnection` instance is specified with STUN and TURN servers, and when it is instantiated, it calls `getUserMedia()` to add local media[8]. Then users start the signaling process via WebSockets or HTTPs, sending candidate message to peers. When the peer receives the candidate information, it calls `addIceCandidate()` [8]. `RTCPeerConnection` also calls the `createOffer()` and `createAnswer()` method to transmit the SDP message to their peers to exchange stream information, security description, etc.[8]

c. RTCDataChannel

`RTCDataChannel` represents a bidirectional data channel between two peers, it includes APIs for data transmission, it nominates the desired data type in the `dataConstraint` argument.

Security and Vulnerability of WebRTC

Being a peer-to-peer communication technology can be a double-edged sword to cybersecurity. On the one hand, user data are mostly transmitted directly to and from the peer, there is no need to concern about being monitored by the tech company or by government. Also, according to WebRTC protocols, transmission between the peers are forced to use encryption protocols, the data is less likely be exposed. On the other hand, to establish a peer-to-peer connection, users need to provide more information to the application, which may lead to some privacy issues.

Another controversial point for WebRTC is that common users will not be aware of if some website is embedded with or is running WebRTC (or some components or WebRTC). Unlike that the browser will prompt to ask for access of camera, running other WebRTC functions will not notify users.

I. IP Leak Issue

To establish a direct connection with peer user, the ICE gathers many IP addresses and ports as candidates, which is visible to the web application (using simple JavaScript). According to IETF's document concerning WebRTC IP address handling, there mainly three kinds of IP that may be leaked[9]:

1. If the client is behind a NAT, the client's private IP address and port will be collected;
2. If the client is multi-homed, the additional public IP addresses for the client can be learned, for example, if the user is using a "split-tunnel" VPN, WebRTC will not only collect the public IP address of the VPN, but also the ISP public address;
3. If the client is behind a proxy which does not handle all network requests, WebRTC will bypass the proxy, and collect the public IP address.

i. Local IP Address Leak

Some advertising platform and companies employ this vulnerability to build a browser fingerprinting for users without using cookies. Companies and advertising platforms use this to track user behaviors, or to distinguish human users from robots. For example, in 2015, nytimes.com was reported to embed third-party WebRTC to scan visitors' local IP addresses[10].

These innovative uses of WebRTC does raise some concerns, but to many people, it might seem that local IP leakage is of less severity, as the local IP only works for your local network, and will not expose too much privacy information, compared to public IP leakage. However, if this vulnerability is used together with some other known vulnerabilities and bugs, it can lead to huge problems.

Matthew Bryant gave us an example.[11] The main idea of his exploit is to find the router's local IP address and identify its router model, and then hack the router based on the router's own vulnerability.

He first scanned the local network, and tried to find all the alive hosts. Then he made attempts to link every alive hosts to some specific static resources that can identify the host. For example, Linksys WRT56G router have static resources such as `/UILinksys.gif`, `/UI_10.gif`, `/UI_Cisco.gif`, etc. If the attempt succeeded, we knew that the host had these static resources and the router was Linksys WRT54G.

The leakage of local IP address makes it possible to automatically do the scanning and hacking procedures, so the leakage itself may not seem serious, but combining it with other vulnerabilities and bugs can make things different.

ii. Public IP Address Leak behind VPN

Many users use VPN to connect to the Internet to conceal their ISP IP address (to conceal their identification on the Internet), but WebRTC can still collect the ISP IP address. *voidsec* reports that in June 4, 2020, about 16 percent of VPN providers leaks user's IP via WebRTC, and the number has been decreased.[12] Luckily, it seems that it is a problem can be fixed by the VPN providers.

I tested the code provided by *voidsec* in browser console, the result (shown in Fig. 2) suggested that my public IP behind VPN is leaked.

```
> function findIP(onNewIP) {
    var myPeerConnection = window.RTCPeerConnection || window.mozRTCPeerConnection || window.webkitRTCPeerConnection;
    var pc = new myPeerConnection({iceServers: [{urls: "stun:stun.l.google.com:19302"}]}),
      noop = function() {},
      localIPs = {},
      ipRegex = /([0-9]{1,3}(\.[0-9]{1,3}){3}|[a-f0-9]{1,4}(:[a-f0-9]{1,4}){7})/g,
      key;

    function ipIterate(ip) {
      if (!localIPs[ip]) onNewIP(ip);
      localIPs[ip] = true;
    }

    pc.createDataChannel("");

    pc.createOffer(function(sdp) {
      sdp.sdp.split('\n').forEach(function(line) {
        if (line.indexOf('candidate') < 0) return;
        line.match(ipRegex).forEach(ipIterate);
      });
      pc.setLocalDescription(sdp, noop, noop);
    }, noop);

    pc.onicecandidate = function(ice) {
      if (!ice || !ice.candidate || !ice.candidate.candidate || !ice.candidate.candidate.match(ipRegex)) return;
      ice.candidate.candidate.match(ipRegex).forEach(ipIterate);
    };
  }


  var ul = document.createElement('ul');
  ul.textContent = 'Your IPs are: '
  document.body.appendChild(ul);

  function addIP(ip) {
    console.log('got ip: ', ip);
    var li = document.createElement('li');
    li.textContent = ip;
    ul.appendChild(li);
  }

  findIP(addIP);
< undefined
  got ip:  115.214.54.62                                                                    VM194:37
```

Fig. 2 IP Leaks behind VPN

II. How to Prevent These WebRTC abuses

1. Users should at least know if some website is running a WebRTC application or a WebRTC function component like ICE. Most browser does not notify the user of running a WebRTC. For Chrome, users can visit chrome://webrtc-internals to check.
2. Download some WebRTC limit extension, for example WebRTC Network Limiter for Chrome, and limit WebRTC behavior according to the user's need.
3. Use a secure VPN, and verify if it can leak your public IP via WebRTC.

Reference

[1] Alvestrand, H. "Overview: Real Time Protocols for Browser-based Applications", November, 2017, https://tools.ietf.org/html/draft-ietf-rtcweb-overview-19.

[2] Nandakumar, S., Jennings, C., "Annotated Example SDP for WebRTC draft-ietf-rtcweb-sdp-07", May, 2020, https://tools.ietf.org/html/draft-ietf-rtcweb-sdp-12.

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

[4] Srisuresh, P. and M. Holdrege, "IP Network Address Translator (NAT) Terminology and Considerations", RFC 2663, DOI 10.17487/RFC2663, August 1999, https://www.rfc-editor.org/info/rfc2663.

[5] Rosenberg, J., Weinberger, J., Huitema, C., and R. Mahy, "STUN - Simple Traversal of User Datagram Protocol (UDP) Through Network Address Translators (NATs)", RFC 3489, DOI 10.17487/RFC3489, March 2003, https://www.rfc-editor.org/info/rfc3489.

[6] Rosenberg, J., Mahy, R., Matthews, P., and D. Wing, "Session Traversal Utilities for NAT (STUN)", RFC 5389, DOI 10.17487/RFC5389, October 2008, https://www.rfc-editor.org/info/rfc5389.

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

[8] "Real time communication with WebRTC", *Google Codelabs*, updated June, 2018 https://codelabs.developers.google.com/codelabs/webrtc-web (Accessed: 21 September 2020)

[9] Uberti, J., "WebRTC IP Address Handling Requirements", *IETF*, July, 2019, https://tools.ietf.org/html/draft-ietf-rtcweb-ip-handling-12

[10] incloud, "WebRTC being used now by embedded 3rd party on http://nytimes.com to report visitors' local IP addresses", *Twitter*, July 2015, https://twitter.com/incloud/status/619624021123010560 (Accessed: 21 September 2020)

[11] Matthew Bryant, "sonar.js – A Framework for Scanning and Exploiting Internal Hosts With a Webpage", *the hacker blog*, August 2015, https://thehackerblog.com/sonar-a-framework-for-scanning-and-exploiting-internal-hosts-with-a-webpage/ (Accessed: 21 September 2020)

[12] voidsec, "VPN Leak", *VOIDSEC*, March 2018, https://voidsec.com/vpn-leak/ (Accessed: 21 September 2020)