Preventing WebRTC IP Address Leaks

Guillaume Nibert[1,2][0000−0002−3277−8533], Sébastien Tixeuil[2,3][0000−0002−0948−7172], Baptiste Polvé[1], Nana J. Bakalafoua M'boussi[1], and Xuan Son Nguyen[1]

[1] Snowpack, F-91400 Orsay, France
{guillaume.nibert, baptiste.polve, nana.bakalafoua, xuanson.nguyen}@snowpack.eu
[2] Sorbonne Université, CNRS, LIP6, F-75005 Paris, France
[3] Institut Universitaire de France, F-75005 Paris, France
sebastien.tixeuil@lip6.fr

Abstract. The WebRTC API enables real-time communication of text, video, and audio media streams through a web browser without requiring third-party extensions. However, it was not designed with privacy in mind. We conduct an experiment to analyse privacy leaks associated with WebRTC on Linux, macOS and Windows. Our findings show that despite recent updates to its specification and implementations, sensitive public IP addresses can still leak during audio/video communication, particularly in large non-NAT corporate networks, even when using a VPN, SOCKS or HTTP/S proxy. To address the observed leaks, we develop a simple, easily maintainable, cross-platform, open-source solution that confines the Mozilla Firefox web browser in a docker container. Our tests show that our containerised solution is effective in all situations even with a compromised browser without restricting applications.

Keywords: WebRTC · IP address leaks · docker · confinement · web browser · privacy · virtual private networks.

1 Introduction

Texting, calling and video chatting across the globe has become an essential part of our daily lives. In particular, the WebRTC API popularised real-time communication via web browsers. We show that this API may reveal sensitive information about users without their knowledge. We focus on IP address leakage when using WebRTC-based communication apps on desktop browsers. Leaking public IP addresses can allow DoS attacks, geolocation, ISP and network type (mobile or not) identification. Leaking private IP addresses can reveal information about the local network, including the ISP, whether the agent is on a corporate or home network, and which services are hosted. These addresses can also be used in other attacks, such as recreating cookies [15]. These data are therefore highly sensitive and valuable to attackers.

In this context, our contribution is fourfold:

This version of the contribution has been accepted for publication, after peer review but is not the Version of Record and does not reflect post-acceptance improvements, or any corrections. The Version of Record is available online at: http://dx.doi.org/10.1007/978-3-031-89350-6_22. Use of this Accepted Version is subject to the publisher's Accepted Manuscript terms of use https://www.springernature.com/gp/open-research/policies/accepted-manuscript-terms.

1. To understand when and where IP address leaks occur while using WebRTC technology, we conduct a thorough evaluation on Ubuntu, macOS and Windows. We implement a test bed that contains all necessary elements to enable WebRTC usage. Our first step is to evaluate the main web browsers behaviour. Our findings show that IP address leaks happen in all cases, regardless of the browser.
2. To further understand IP address leaks, we add the ability to use a VPN, HTTP/S or SOCKS proxy to our test bed for privacy purposes. These servers supposedly provide anonymity for the WebRTC-based user application by masking their IP address. Using Mozilla Firefox as a benchmark, we confirm that even with a VPN, IP address leaks can still appear, publicly identifying the user.
3. To protect against IP addresses leakage while using WebRTC technology, we propose a simple user-friendly solution using Docker-based containerisation. This solution was added to our test bed by enabling the ability to use a containerised version of Mozilla Firefox for evaluation purposes. Additionally, we study the resilience of our containerisation proposal against an adversary able to compromise the user's web browser. Our findings show that the combination of containerisation and a VPN is effective in protecting against IP address leaks, even in the case of a compromised browser. This solution is easily maintainable with Docker, thanks to a shared base in Dockerfiles and Compose files across all systems.
4. To evaluate the performance of our Mozilla Firefox containerised solution with respect to its native version, we run open-source browser benchmarks on Ubuntu Linux, macOS, Windows, and show the overhead is limited.

Our test bed, which includes a Firefox script, a Docker containerisation solution, and all data related to our reported results, is available under the GNU General Public License version 3 (GPLv3)[4] *at: https://github.com/snowpackvipn/preventing-webrtc-ip-address-leaks*.

The rest of this paper is organised as followed: Section 2 presents WebRTC and its associated privacy problems. Section 3 reviews previous approaches to the IP address leak problem, and the current privacy related countermeasures provided in the state-of-the-art WebRTC API. Section 4 describes in detail our findings, proposal and evaluation. Section 5 compares the performance of the native and containerised Firefox browser. Finally, Section 6 provides concluding remarks and suggests research avenues for future work.

2 WebRTC IP address leaks

Suppose Alice and Bob want to use the WebRTC API for VoIP peer-to-peer communication. Alice is the initiator of the connection (figure 1). *Steps 1-2*, using the ICE (Interactive Connectivity Establishment) [13] framework, she contacts a STUN (Session Traversal Utilities for NAT) server via her web browser, which returns her public IP address(es). *Step 3*, she provides a list of ICE candidates containing her public and private IP address(es) in the form of SDP (Session

[4] The GNU General Public License v3.0: https://www.gnu.org/licenses/gpl-3.0.html [accessed on 17 September 2024].

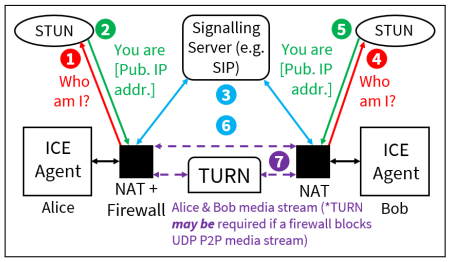


Fig. 1. Establishing real-time Peer-to-Peer media communication behind NATs

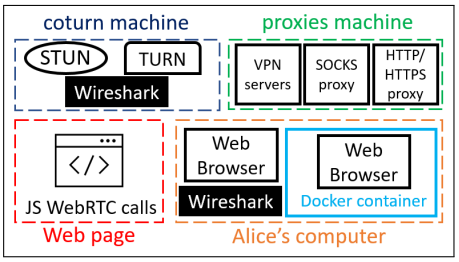


Fig. 2. test bed elements

Description Protocol) objects via a signalling server, which in turn forwards this list to Bob. *Steps 4-5-6*, Bob performs the same steps as Alice. *Step 7*, they establish their connection, then exchange their media stream peer-to-peer. If both participants are unable to use the same transport protocol (TCP/UDP), the media stream transport between Alice and Bob is impossible. A TURN server (Traversal Using Relays around NAT) solves this problem by acting as TCP stream/UDP datagram translator proxy.

So, addresses collected by the ICE protocol from the WebRTC communication initiator include *public* IPv4/IPv6 addresses returned by the STUN server, *public* IPv4/IPv6 addresses allocated by a TURN server, and both *public and/or private* IPv4/IPv6 addresses attached to physical and virtual interfaces.

WebRTC IP leaks are induced by its specification. STUN/TURN servers only know the source public IP addresses from the received requests. Yet, additional information can be retrieved by configuring a wildcard DNS record for this server [25]. Irrespective of VPN, one cannot block a STUN/TURN request from the web browser, as this request is made outside the signalling framework [6].

WebRTC leak issue. all or part of the IP addresses' list may be sent to the signalling server, and then to the peer [29]. So, an adversary may deploy their own STUN, TURN, and signalling servers, as well as a malicious web page with JavaScript code calling the WebRTC API (locally run on the client's computer), to gather all addresses of a victim, and send them to their signalling server.

3 Related Work

In 2015, Roesler [25] demonstrates private and public IP address leaks using the WebRTC API. Roesler's JavaScript code contacts a STUN server to get the public IP address of the machine, with private addresses extracted from the SDP offers. As mentioned earlier, it is also possible to encode other information using a wildcard DNS entry associated with a STUN server. For example, if we want to disclose an IP, URIs of the STUN server could be: *stun:**IP-address**.stunserver.com[:port].*

Takasu et al. [27] show a JavaScript code collecting the user's private IP addresses and potentially public IP addresses associated with system's local interfaces.

Englehardt and Narayan [4] explain that collected IP addresses can be those of all local network interfaces. If some of the IP addresses match those provided by the ISP, using a VPN would leak the VPN public address and the ISP provided addresses into the sent SDP offer to the peer. Their study focuses on private IP addresses leaks behind a NAT for tracking purposes. Liu et al. [16] and Revyakina [23] focus on private IP addresses leaks only. Hosoi et al. [11] develop a network scanner using the WebRTC API to extract private and public addresses from an SDP object. They suggest disabling WebRTC or JavaScript, a solution we do not find acceptable as we want to use WebRTC.

Al-Fannah [2] carries out an in-depth study by developing a website using the WebRTC API to recover sensitive IP addresses. The paper demonstrates more protective VPNs and suggests different countermeasures to improve the user's privacy, such as disabling WebRTC, disabling IPv6, using ad-dons, and/or using a more protective browser (Safari). In addition, this study highlights leaks linked to incorrect VPN configurations, particularly when IPv6 is activated in addition to IPv4. Other research has looked into these problems [20,14,1,21]. Ramesh et al. [21] create a tool called *VPNalyzer* which, among other things, identifies VPNs that leak IPv6 addresses. This line of works show that a VPN shall be chosen with care, as these leaks are unrelated to WebRTC.

Coming back to the problems directly related to WebRTC, Reiter and Marsalek [22] present an extensive analysis on WebRTC privacy and security risks. They present code to retrieve private and public IP addresses from local interfaces and a STUN server if the machine is behind a NAT. Public and local IP data can be used by port scanners (*e.g.* `jslanscanner`[5]) to find other services associated with these IPs, geolocate the victim, carry out fingerprinting or inter-protocol attacks. Their recommended solution requires updating the WebRTC specification, which is a long process.

Hazhirpasand and Ghafari [10] determine information about the victims' local network by finding the status of network node ports using round-trip delay time heuristics. They build an extension for both Google Chrome and Mozilla Firefox to analyse requests made on the user's network. If these requests target local network nodes with numerous connections, the user is notified. To avoid a complete leak, they suggest disabling WebRTC or limiting it.

Fakis et al. [6] propose a detector for calls to the WebRTC API in a web page. This detector comes as a browser extension or a gateway. The detector analyses the presence of calls to the WebRTC API in a web page, blocks such calls, notifies the user, and restores the WebRTC objects depending on the user's decision. This blocks STUN requests transmission and identifies malicious sites that hide JavaScript calls to the WebRTC API to discover sensitive IP addresses.

[5] Heyes, G.: Jslanscanner. (Aug 2007). https://code.google.com/archive/p/jslanscanner/ [accessed on 14 June 2024]

WebRTC privacy-friendly specifications. The latest WebRTC RFC 8828 [29] (Jan. 2021) addresses privacy concerns and proposes four ways to manage IP addresses collection in WebRTC. *Mode 1* collects all the machine's interface addresses, and those issued by TURN and STUN servers. *Mode 2* gathers IP addresses attached to the default route interface, and those provided by STUN and TURN servers contacted from the same interface. *Mode 3* accumulates addresses provided by STUN and TURN servers only from the default route, without disclosing local interface IP addresses. *Mode 4* forces the proxy use on the default route[6]. In this way media traffic must pass through the proxy. If the proxy does not support UDP, TCP is used to transport the media stream.

By default (if no user consent is given), mode 2 is used. Otherwise, mode 1 is used. User consent is managed by the API's *getUserMedia*[7] method, which is used to obtain authorisation to access peripherals such as the microphone and camera. In other words, authorising access to the microphone and/or camera means *agreeing* to be in mode 1. On one hand, if a user is hidden behind a VPN and has not given its consent (which leads to mode 2), the signalling server can only retrieve the VPN's virtual interface addresses (VPN private IPv4/IPv6 addresses) and those returned by the STUN/TURN servers (VPN public IPv4/IPv6 addresses). The STUN/TURN servers only know the VPN's public address(es). On the other hand, the user can only exchange text media streams, not carry out audio/video conferences since the WebRTC API does not have authorisation to access the microphone/camera (mode 2).

The IETF draft by Fablet et al. [5] proposes a new scheme against local IP address leakage by replacing them with multicast mDNS addresses. The complementary draft by Uberti et al. [30] adds two operating modes to the WebRTC IP Address Handling Requirements, identical to mode 2 but using mDNS addresses. *Mode 2.1* replaces the IPv4 address associated with the default route local interface, and all IPv6 addresses are associated with the same interface by mDNS addresses unless they are privacy-preserving addresses [19]. *Mode 2.2* replaces all IPv4 and IPv6 addresses associated with the default route local interface by mDNS addresses.

Current Implementations. The main desktop browsers [26], namely Chrome, Safari, Edge, Firefox, Opera and Brave[8] implement all or part of the aforementioned WebRTC specifications. WebRTC is disabled in the Tor Browser [24].

Users can force a mode regardless of the authorisations granted through *getUserMedia*. This is easily done with Brave Browser and Opera (from the settings), less so on Firefox (from the hidden settings *about:config*), Chrome and

[6] Note that if several *default routes* are available, the interface chosen is the one with the highest priority (lowest interface metric value).

[7] https://developer.mozilla.org/en-US/docs/Web/API/MediaDevices/getUserMedia [accessed on 13 June 2024]

[8] Brave has been included as it claims to be a privacy-focused browser.

Edge (from an admin console managing policies to be applied to the browser or extensions[9]), and impossible on Safari (options do not exist).

None of the aforementioned studies compared IP address leaks according to the currently defined WebRTC IP address handling modes. They did not explore the collection of public IPv4 addresses attached to multiple local interfaces or investigate the IPv6 case in sufficient depth, including behind an IPv4 or IPv6 VPN tunnel with both IPv4 and IPv6 forwarding capabilities. They also did not consider the ability to hide behind a SOCKS/HTTP(S) proxy. As a result, we need a thorough investigation to understand under which setting the IP address leak occurs.

4 IP address leakage evaluation

4.1 Our test bed architecture

We first set up a system composed of (figure 2):
-*a public STUN server and a public TURN server*, located on the same machine, implemented by `coturn`[10]. Our STUN/TURN servers do not exploit the DNS wildcard entry attack described by Roesler [25].
-*a web page*, which is accessible online or locally, and which is in charge of executing a call to the WebRTC API to generate ICE offers locally in the calling web browser[11]. The list of candidates is generated and displayed locally.
-*a web client*, executed either natively or within a Docker container. Indeed, to improve WebRTC privacy, we propose to isolate the web browser in a container that provides a single interface attaching a private IPv4 address and, additionally for Linux systems, an IPv6 ULA address[12] [9], both of which can be routed to the Internet via NAT rules. Besides, we consider two versions of the web client, one that is honest (running with default settings), and another one that is compromised for WebRTC. Docker was chosen due to its cross-platform advantage. The solution has been implemented on Linux, macOS and Windows. The user runs one script to use the solution.
-*a VPN server, a SOCKS and a HTTP/S proxy*, which may hide the web client.

One experiment on our test bed consists in having the *web client* open the *web page*. As the client does so, a Wireshark traffic analysis is performed both on the client and the machine hosting the STUN and TURN servers to monitor the requests. In fact, when the list of ICE candidates is created, if any of them are considered redundant, they are removed [13].

In our experiments, the Linux, macOS and Windows clients are connected via both Wi-Fi and Ethernet. All clients have public IPv4 and IPv6 addresses

[9] WebRTC Network Limiter, https://chrome.google.com/webstore/detail/webrtc-network-limiter/npeicpdbkakmehahjeeohfdhnlpdklia [accessed on 13 June 2024].
[10] Coturn server: https://github.com/coturn/coturn [accessed on 13 June 2024].
[11] The JS code we produce is based on Reiter and Marsalek work [22].
[12] Docker does not support IPv6 on Mac and Windows: https://docs.docker.com/config/daemon/ipv6/ [accessed on 14 June 2024]

assigned to their respective interfaces. IPv6 addresses are generated in stable privacy addressing mode [8]. Surprisingly, this network configuration is not uncommon: the *eduroam* networks of two university campuses (approximately 1,500 people and 30,000 people respectively) have no NAT.

4.2 Compromised web browser threat model

The adversary is able to exploit any vulnerability in Firefox to force it to operate in the most unfavourable WebRTC mode, or even to modify the WebRTC implementation maliciously (e.g. mode 1 always used regardless of the user's consent). The adversary's capabilities are limited to compromise the browser. The WebRTC client is corrupted and is unaware of it.

Our compromised browser is forced to run only in a mode where WebRTC does not respect privacy: no mDNS protection, even if the user does not give his consent, and link-local/loop-back IP addresses can be candidates.

We could imagine other ways of compromising the browser: a post on the Mozilla Russia forum [3] gives a way of authorising the installation of extensions not signed by Mozilla. All that needs to be done is to create an executable script and have the user download it.

For ease of reading, all the tables mentioned in the following three subsections are grouped together in subsection 4.6.

4.3 Evaluation of current web browsers

Table 1 compares the behaviour of the main web browsers when they visit the experiment web page and make calls to the API, for each WebRTC IP Address Handling mode they implement. The default browser configuration corresponds to a vanilla installation, freshly set up on the computer without any modifications. In this configuration, the user's consent results exclusively in a given WebRTC IP handling mode. For all the web browsers tested except Safari, mode 2.2 is used without consent, but with consent, mode 1 is used. Regarding Safari, whatever the user's consent, mode 2 is used. For the forced configuration case, we consider all possibilities i.e. whether the user has given consent or not. Safari does not allow forced mode.

Similarly to previous IETF works [29,30], we observe that for all web browsers, configured in modes 2.2, 2 or 3, and regardless of the user's possible consent for these modes, the STUN and TURN servers know the public IPv4 and IPv6 addresses from the default route interface with the highest priority. For all three modes, the list of final ICE candidates that may be sent to the signalling server and the peer contain: *at worst (mode 2)*, private IP addresses and/or public IP addresses (from the local interface), public IP addresses (from STUN) and relayed addresses (address allocated by the TURN server that does not identify the user), *by default (mode 2.2)*, mDNS addresses that cannot be resolved by the signalling server and the peer if they are outside the client's network, public

IP addresses (from STUN) and relayed addresses, or *at best (mode 3)*, public IP addresses only (from STUN) and relayed IP addresses.

With user consent, Firefox, Chrome, Edge, Opera, and Brave use mode 1. The STUN/TURN servers receive requests from interfaces with a route to the Internet (Wi-Fi and Ethernet). These interfaces attach both a public IPv4 and a public IPv6 addresses, the STUN and TURN servers therefore know in total 4 IP addresses. Private and public addresses attached to all interfaces are also discovered. The signalling server therefore has knowledge of private and public IP addresses, since our client is connected to a network that allocates public IP addresses. As these public addresses are the same as those coming from the STUN server, the risk may be limited. However, we colour them in red on Table 1 because the sequel demonstrates these addresses are a problem. Let us highlight that WebRTC real-time communication involving microphone and/or camera usage (a frequent phenomenon nowadays) implies using mode 1 by default. Since the user uses his/her ISP's connection directly, we colour in light red cells that reveal sensitive public IP addresses to the STUN/TURN servers, and later to the signalling server and the peer. Red and light red colour codes are common to the other tables.

4.4 Evaluation of an *honest* Firefox browser in various configurations (VPN - SOCKS - HTTP/S - Docker)

We set up an OpenVPN UDP VPN server, a Wireguard VPN server, a SOCKS and a HTTP/S proxy on another machine to check for leaks of the user's public IPv4 and IPv6 addresses when all the traffic from the host machine goes through these proxies[13]. Our solution uses Firefox as it is open source, considered by the community as one of the most privacy-protecting browsers, and Mozilla has a transparent policy on its product, development and funding. We have not tested the solution with other browsers as we consider the solution to be network-impacting rather than applicative.

The results are given in Table 2. We colour in green the cells whose IP addresses are those of the VPN server protecting user's identity, and in light green the cells for which no ISP IP addresses could be retrieved. These green colour codes are also common to the Table 3. We notice, alarmingly, that the STUN/-TURN requests consistently bypass the built-in Firefox SOCKS and HTTP/S client, both with and without Docker. With the VPN, by default without containerisation when the user gives consent in use with a VPN, there is a leak of the ***public IP addresses of the ISP***, *private addresses*, and those of the *VPN*. The future signalling server and peer will therefore be able to know this information, and the fact that the user is using a VPN. The STUN/TURN server could also get this information by exploiting the Roesler's DNS wildcard entry attack [25] which is not studied here. However, all computer traffic goes through the VPN (cf. Wireshark data).

[13] For SOCKS and HTTP/S proxies, we use the Firefox default built-in client.

We also obtained neither candidates, nor STUN/TURN requests by forcing mode 4 for all the configurations evaluated. The Mozilla Wiki indicates that some parameters are poorly documented or not even implemented [18]. The associated Wireshark traces are available in the GitHub repository.

Obviously, forcing mode 3 would be an interesting alternative, but doing so hinders quality of service: the user loses full support of IPv6 in the context of a WebRTC communication (see orange cells in Table 2).

4.5 Evaluation of a *compromised* Firefox browser in various configurations (VPN - SOCKS - HTTP/S - Docker)

The results are given in Table 3. We note that in the compromised version, with or without user consent, there is no more mDNS protection. However, without consent, the leaked addresses are those attached to the default route interface.

If all traffic is redirected through a VPN, then we lose quality of service again: IPv6 is no more possible (orange cells). In mode 1 (user consent), there is a leak of both VPN and ***addresses provided by ISP***. The signalling server and the peer are thus aware of this information. STUN/TURN servers also learn it, if they implement the Roesler attack (DNS wildcard entry) [25], otherwise only VPN IPs are leaked.

However, the compromised dockerised web client protects users from any leak of sensitive private IP addresses. If a VPN is also used, they will not reveal any sensitive private address, nor any public address associated with their ISP. As in the previous evaluation (section 4.4), STUN/TURN requests bypass the integrated Firefox proxy clients; configuring a SOCKS or HTTP/S proxy in Firefox settings is ineffective and does not improve the user's privacy, highlighting a critical issue. Similarly, the evaluation in mode 4 does not appear in either of the tables because we did not obtain candidates or STUN/TURN requests for any of the configurations assessed [18].

4.6 Results of the WebRTC leaks evaluation

Our results are summarised in Tables 1, 2, and 3. More precisely, Table 1 considers WebRTC various IP handling modes, Table 2 reviews native and containerised architectures, while Table 3 considers a compromised web browser.

Software information. The various software and versions used in our experiments are available in the Anonymous GitHub repository.

Tables notations. The notation in Tables 1, 2, and 3 are as follows. "X" means that one (or more) client's address(es) was retrieved (depending on the WebRTC IP handling mode). "." means that no address was retrieved. The means used to observe the host addresses retrieved and the addresses discovered by the STUN/TURN servers are in the "Obs." column for Observation. No crosses in Tables 1, 2, and 3 correspond to TURN *relayed addresses* as they do not identify the client. *VPN local, Doc. priv., Doc. ULA, VPN* mean *VPN private IP address, Docker private IPv4 address, Docker ULA IPv6 address, VPN's public IP address*, respectively. *MF* means Mozilla Firefox, *lo* means loopback IP address(es), and *ll* means private link-local address(es).

Table 1. Comparison of leaked IP addresses' types by the main web browsers for different WebRTC IP handling modes.

	Browser		**Mozilla Firefox**					**Chrome - Edge Opera - Brave**				**Safari**
	Configuration		Default		Forced			Default		Forced		Default
	User consent		Yes	No	Yes	No[b]	Both	Yes	No	Both	Both	Both
	Mode		1	2.2	2	2→2.2	3	1	2.2	2	3	2
Observation	**Retrieved address(es)**											
	Source	***Type***										
SDP offers	**Local**[a]	mDNS	.	X	.	X	.	.	X	.	.	.
		Priv. IPv4	X[c]	.	X[e]	.	.	X[c]	.	X[e]	.	X[e]
		Priv. IPv6	X[d]	.	X[d]	.	.	X[d]	.	X[d]	.	X[f]
		Pub. IPv4	X	.	X[e]	.	.	X	.	X[e]	.	X[e]
		Pub. IPv6	X	.	X	.	.	X	.	X	.	X
Wireshark	**STUN**	Pub. IPv4	X	X	X	X	X	X	X	X	X	X
		Pub. IPv6	X	X	X	X	X	X	X	X	X	X
	TURN	Pub. IPv4	X	X	X	X	X	X	X	X	X	X
		Pub. IPv6	X	X	X	X	X	X	X	X	X	X

Table 2. Comparison of leaked addresses' types by a **vanilla** Firefox for different WebRTC IP handling modes and configurations. *Docker only supports IPv6 on Linux.*

	Browser		**MF only MF + SOCKS/HTTP(S)**					**MF + VPNs**					**MF dockerised MF dockerised + SOCKS/HTTP(S)**					**MF dockerised + VPNs**				
	Configuration		Default		Forced			Default		Forced			Default		Forced			Default		Forced		
	User consent		Yes	No	Yes	No[b]	Both	Yes	No	Yes	No[b]	Both	Yes	No	Yes	No[b]	Both	Yes	No	Yes	No[b]	Both
	Mode		1	2.2	2	2.2	3	1	2.2	2	2.2	3	1	2.2	2	2.2	3	1	2.2	2	2.2	3
Obs.	**Retrieved addr.**																					
	Source	***Type***																				
SDP offers	**Local**[a]	mDNS	.	X	.	X	.	.	X	.	X	.	.	X	.	X	.	.	X	.	X	.
		Priv. IPv4	X[c]	.	X[e]	.	.	X[c]	.	VPN local	.	.	Doc. priv.	.	Doc. priv.	.	.	Doc. priv.	.	Doc. priv.	.	.
		Priv. IPv6	X[d]	.	X[d]	.	.	X[d]	.	.[h]	.	.	Doc. ULA	.	Doc. ULA	.	.	Doc. ULA	.	Doc. ULA	.	.
		Pub. IPv4	X	.	X[e]	.	.	X	.	.	.	.	.	.	.	.	.	.	.	.	.	.
		Pub. IPv6	X	.	X	.	.	X	.	.	.	.	.	.	.	.	.	.	.	.	.	.
Wire shark	**STUN**	Pub. IPv4	X	X	X	X	X	VPN	VPN	VPN	VPN	VPN	X	X	X	X	X	VPN	VPN	VPN	VPN	VPN
		Pub. IPv6	X	X	X	X	X	.[g]	.[h]	.[h]	.[h]	.[h]	X	X	X	X	X	VPN	VPN	VPN	VPN	VPN
	TURN	Pub. IPv4	X	X	X	X	X	VPN	VPN	VPN	VPN	VPN	X	X	X	X	X	VPN	VPN	VPN	VPN	VPN
		Pub. IPv6	X	X	X	X	X	.[g]	.[h]	.[h]	.[h]	.[h]	X	X	X	X	X	VPN	VPN	VPN	VPN	VPN

Tables footnotes. The footnotes referenced in Tables 1, 2, and 3 are as follows:

[a] For IP addresses, it is subject to have these addresses' types associated with the local interface(s) used according to the different modes (RFC 8828 [29]).

[b] Even when forcing the use mode 2 on Firefox, the real mode is 2.2 without user consent (mDNS protection of the preferred interface's local addresses).

[c] Firefox filters local IPv4 addresses associated with all interfaces and skips link-local and loop-back addresses [17]. Chromium-based browsers: [28].

[d] A filtering of IPv6 addresses associated with the interface(s) chosen is done. One or more IPv6 addr. preferred by this filtering will be chosen [17,28].

[e] It is either a public or a private IPv4, as mode 2 selects the default route interface and as an interface can only get one IPv4.

[f] Our observation shows that if the default interface does not have a public IPv6 addr. but only a ULA IPv6 addr. (routable to the internet via a NAT), it will be included

Table 3. Comparison of leaked addresses' types by a **compromised** MF for different WebRTC IP handling modes and configurations. *Docker only supports IPv6 on Linux.*

	Browser		**compr. MF compr. MF + SOCKS/HTTP(S)**		**compr. MF + VPNs**		**compr. MF dockerised & compr. MF dockerised + SOCKS-HTTP(S)**		**compromised MF dockerised + VPNs**	
	Configuration		Forced		Forced		Forced		Forced	
	User consent		Yes	No	Yes	No	Yes	No	Yes	No
	Mode		1	2	1	2	1	2	1	2
Obs.	**Retrieved addr.**									
	Source	***Type***								
SDP Offers	**Local**[a]	mDNS	.	.	.	.	.	.	.	.
		Priv. IPv4	X[c]	X[e]	X[c]	VPN local	Doc. priv. + lo	Doc. priv.	Doc. priv. + lo	Doc. priv.
		Priv. IPv6	X[d]	X[d]	X[d]	.[g]	Doc. ULA + lo + ll	Doc. ULA	Doc. ULA + lo + ll	Doc. ULA
		Pub. IPv4	X	X[e]	X	.	.	.	.	.
		Pub. IPv6	X	X	X	.	.	.	.	.
Wire Shark	**STUN**	Pub. IPv4	X	X	VPN	VPN	X	X	VPN	VPN
		Pub. IPv6	X	X	.[g]	.[h]	X	X	VPN	VPN
	TURN	Pub. IPv4	X	X	VPN	VPN	X	X	VPN	VPN
		Pub. IPv6	X	X	.[g]	.[h]	X	X	VPN	VPN

in an ICE candidate (tested with OpenVPN UDP assigning a local ULA addr.). It is very like having a filter similar to the one in Firefox/Chromium.

[g] See [d]; In our tests, Wi-Fi and Ethernet interfaces attach temporary routable public IPv6 addresses. Thus, the VPN local ULA address is eliminated. There are indeed attempts to connect to the TURN and STUN servers via the public addresses (seen on Wireshark) but blocked by the VPN.

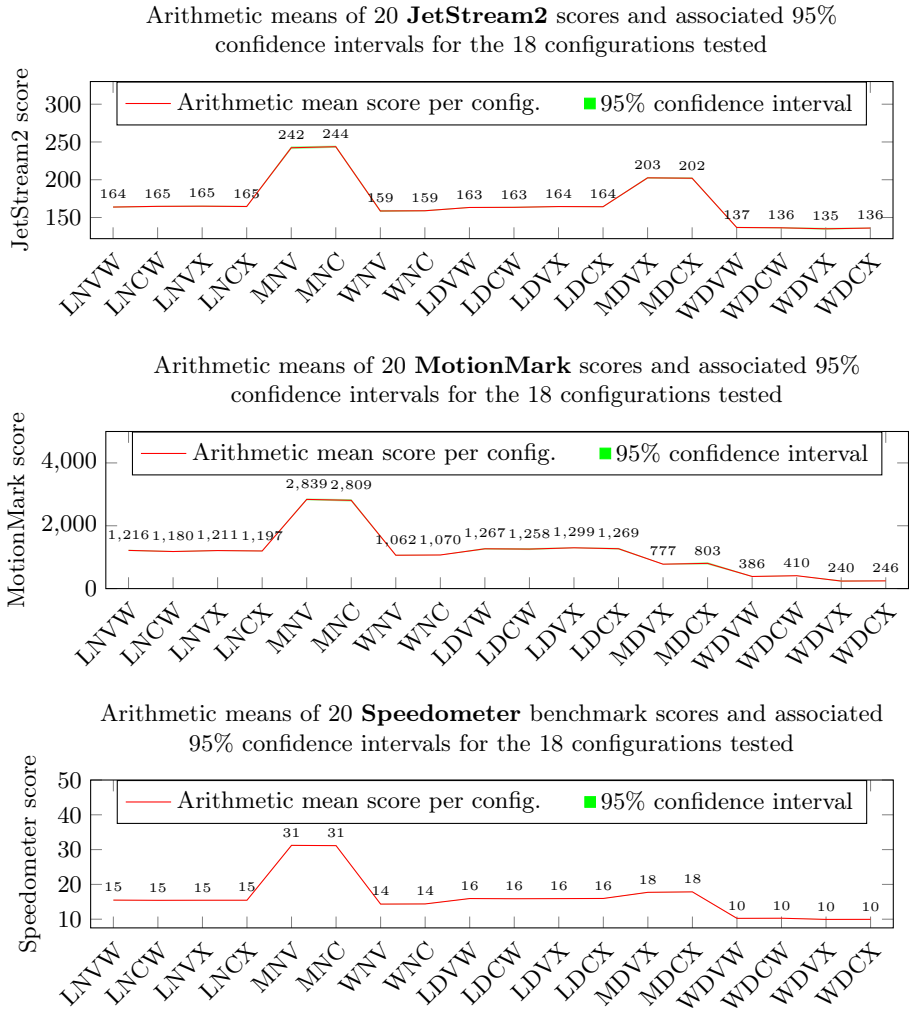
[h] See [d]; It seems that filtering is done on all interfaces before selecting the default interface used in modes 2, 2.2 and 3 (here VPN one). As there is at least one public IPv6 address, the VPN ULA is removed. Firefox then appears to select only remaining IPv4 addresses associated with the VPN interface (since the ULA has been removed). Consequently, no private IPv6 ULA address, routable via the VPN NAT, is present in the ICE candidate list, which prevents any requests to the STUN/TURN servers.

5 Performance impact evaluation

We evaluate our containerised Firefox web browser in terms of performance with respect to its native, non-containerised version. We use three open-source benchmarks[14] suites actively developed and used by the three main web browser engine developers, Apple, Mozilla and Google: JetStream2 v2.2 (a JavaScript/WebAssembly benchmark suite); MotionMark v1.3 (a graphics benchmark); Speedometer v3.0 (a web application responsiveness benchmark). Each benchmark suite is run twenty times in 18 different configurations described in figure 3. For each configuration we calculate the arithmetic mean of the scores obtained from each benchmark and its 95% confidence interval (figure 3).

We can therefore compare the impact of our containerised solution on performance by calculating the ratio of the mean scores of the Firefox native version to the mean scores of the containerised version for the three benchmarks (figure 4). We note that with Linux, the speed differences with the JetStream2 and

[14] Hosted by Apple at: https://browserbench.org/ [accessed on 13 June 2024].



Meaning of groups of 3 and 4 letters representing the configurations tested on the x axis.

1^{st} letter: Linux (L), macOS (M) or Windows (W) host OS
2^{nd} letter: Native (N) or Dockerised (D) execution of Firefox
3^{rd} letter: Vanilla (V) or Compromised (C) version of Firefox
4^{th} letter: X (X) or Wayland (W); *N/A on macOS and Windows Native execution of Firefox*

Fig. 3. Mean scores for the 3 benchmarks for each configuration tested (higher is better)

Speedometer benchmarks are minimal, and in some cases, the native version is slightly slower than the containerised version. This is due to containers being simply namespaces dedicated to the process running inside them [7]. On Windows, the native Firefox browser is respectively around 17%, 168% and 40% faster than the *Wayland* containerised solution on JetStream2, MotionMark, Speedometer,

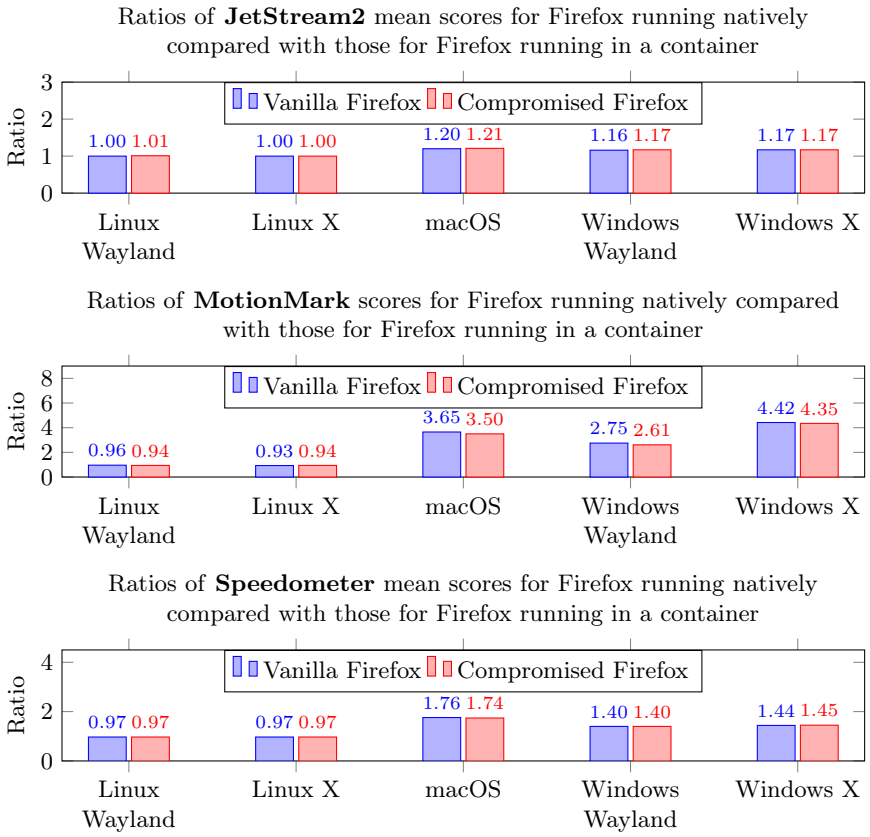


Fig. 4. Ratios of the 3 benchmarks mean scores for Firefox running natively compared with those for Firefox running in a container (lower is better)

and 17%, 342% and 45% faster than the *X* containerised solution. On macOS, the native Firefox browser is respectively around 21%, 258% and 75% faster than the containerised solution on JetStream2, MotionMark and Speedometer. The performance differences on macOS and Windows are attributable to the use of a virtual machine to run Docker and, in particular for the MontionMark graphics benchmark, the inability of the containerised Firefox to access the host GPU.

- Ubuntu 22.04 and Windows 11 Pro 23H2 with a Dell Latitude 5520 (2022), Intel Core i5-1145G7 @ 2.60 GHz, 16 GB RAM
- macOS Sonoma 14.5 with a MacBook Pro 13-inch (2022), M2, 16 GB RAM

6 Conclusion

We analysed the IP address leaks resulting from the use of the WebRTC API in various situations and concluded that none of the most popular web browsers

ensure sufficient privacy on this matter. Critically, we first observe that configuring a SOCKS or HTTP/S proxy in Firefox settings is ineffective and provides absolutely no protection to the user. We also note that large non-NAT corporate networks exist and are not rare situations as shown by the *eduroam* networks of the two university campuses. This situation leads to a leak of public IP addresses even when hidden behind a VPN. Despite not studying the case of a compromised browser, RFC 8828 [29] and its complementary draft [30] shows promise in describing different operation modes. However, using the *getUserMedia* method to make WebRTC more privacy-friendly is not reasonable. User consent definition should be reviewed, and users should be informed of how consent is given. Forcing mode 3 is difficult on Chrome and Edge, making it hard to protect against IP address leaks. Last, without the work of Fakis et. al [6], it is impossible to natively block STUN/TURN requests.

To address these problems, we proposed a simple container-based solution to enable WebRTC usage without compromising privacy in various settings. Our Firefox isolation in a Docker container provides only one network interface and safeguarding the users from IP address leaks and compromises during video/audio communication (mode 1). It leaks less sensitive information: only the private IP addresses of the unique network interface provided by the container, and the public IP addresses provided by its ISP (prefix for IPv6). By hiding behind a VPN, no sensitive information leaks to third-party STUN/TURN servers, signalling and the other peer. Our solution is user-friendly, requiring no extensions or IPv6 disabling at the host level. The performance impact of our containerised solution compared with running Firefox is zero on Ubuntu, and low on other OSes except for the graphics benchmark, with a peak on macOS and Windows.

Future research could focus on improving this aspect on these two systems. Additionally, conducting experiments in more diverse environments, including various network configurations, would enhance assessment of the stability and effectiveness of the solution under real-world conditions. Extending this approach to containerise other web browsers and evaluating its effectiveness could broaden the solution's applicability. There should also be potential for developing a more user-friendly deployment method for the Docker-based solution possibly through a graphical user interface (GUI) for non-technical users. Lastly, creating a more comprehensive guidance for implementing the containerised solution and integrating it with existing privacy tools, such as VPNs or anonymity networks would further improve its usability.

Acknowledgments. This work was supported by the French National Association of Research and Technology under CIFRE No. 2022/0074 and France 2030 cybersecurity acceleration strategy.

Disclosure of Interests. The authors declare no conflicts of interest.

References

1. Abbas, H., Emmanuel, N., Amjad, M.F., Yaqoob, T., Atiquzzaman, M., Iqbal, Z., Shafqat, N., Shahid, W.B., Tanveer, A., Ashfaq, U.: Security Assessment and

Evaluation of VPNs: A Comprehensive Survey. ACM Computing Surveys **55**(13s), 273:1–273:47 (Jul 2023). https://doi.org/10.1145/3579162
2. Al-Fannah, N.M.: One leak will sink a ship: WebRTC IP address leaks. In: 2017 International Carnahan Conference on Security Technology (ICCST). pp. 1–5. IEEE, Madrid, Spain (Oct 2017). https://doi.org/10.1109/CCST.2017.8167801
3. banbot: How to disable digital signature verification in Firefox add-ons (in Russian). https://forum.mozilla-russia.org/viewtopic.php?id=70326 (Aug 2016), accessed on 2024-07-12.
4. Englehardt, S., Narayanan, A.: Online tracking: A 1-million-site measurement and analysis. In: Proceedings of the 2016 ACM SIGSAC conference on computer and communications security. pp. 1388–1401. Association for Computing Machinery, Vienna, Austria (Oct 2016). https://doi.org/10.1145/2976749.2978313
5. Fablet, Y., Uberti, J., Borst, J.D., Wang, Q.: Using Multicast DNS to protect privacy when exposing ICE candidates. Internet Draft draft-ietf-mmusic-mdns-ice-candidates-03, Internet Engineering Task Force (Dec 2021), https://datatracker.ietf.org/doc/draft-ietf-mmusic-mdns-ice-candidates/03/, accessed on 2024-07-12.
6. Fakis, A., Karopoulos, G., Kambourakis, G.: Neither Denied nor Exposed: Fixing WebRTC Privacy Leaks. Future Internet **12**(5), 92 (May 2020). https://doi.org/10.3390/fi12050092
7. Felter, W., Ferreira, A., Rajamony, R., Rubio, J.: An updated performance comparison of virtual machines and Linux containers. In: 2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS). pp. 171–172 (Mar 2015). https://doi.org/10.1109/ISPASS.2015.7095802
8. Gont, F.: A Method for Generating Semantically Opaque Interface Identifiers with IPv6 Stateless Address Autoconfiguration (SLAAC). Request for Comments RFC 7217, Internet Engineering Task Force (Apr 2014). https://doi.org/10.17487/RFC7217
9. Haberman, B., Hinden, B.: Unique Local IPv6 Unicast Addresses. Request for Comments RFC 4193, Internet Engineering Task Force (Oct 2005)
10. Hazhirpasand, M., Ghafari, M.: One Leak Is Enough to Expose Them All. In: Payer, M., Rashid, A., Such, J.M. (eds.) Engineering Secure Software and Systems. pp. 61–76. Lecture Notes in Computer Science, Springer International Publishing, Cham (2018). https://doi.org/10.1007/978-3-319-94496-8_5
11. Hosoi, R., Saito, T., Ishikawa, T., Miyata, D., Chen, Y.: A Browser Scanner: Collecting Intranet Information. In: 2016 19th International Conference on Network-Based Information Systems (NBiS). pp. 140–145. IEEE, Ostrava, Czech Republic (Sep 2016). https://doi.org/10.1109/NBiS.2016.10
12. Jakobsson, C.: Peer-to-peer communication in web browsers using WebRTC A detailed overview of WebRTC and what security and network concerns exists. Bachelor's thesis, Umeå University (2015), http://urn.kb.se/resolve?urn=urn:nbn:se:umu:diva-108372, accessed on 2024-07-12.
13. Keränen, A., Holmberg, C., Rosenberg, J.: Interactive Connectivity Establishment (ICE): A Protocol for Network Address Translator (NAT) Traversal. Request for Comments RFC 8445, Internet Engineering Task Force (Jul 2018). https://doi.org/10.17487/RFC8445
14. Khan, M.T., DeBlasio, J., Voelker, G.M., Snoeren, A.C., Kanich, C., Vallina-Rodriguez, N.: An Empirical Analysis of the Commercial VPN Ecosystem. In: Proceedings of the Internet Measurement Conference 2018. pp. 443–456. IMC '18, Association for Computing Machinery, New York, NY, USA (Oct 2018). https://doi.org/10.1145/3278532.3278570

15. Laperdrix, P., Bielova, N., Baudry, B., Avoine, G.: Browser Fingerprinting: A Survey. ACM Trans. Web **14**(2), 8:1–8:33 (Apr 2020). https://doi.org/10.1145/3386040
16. Liu, X., Liu, Q., Wang, X., Jia, Z.: Fingerprinting Web Browser for Tracing Anonymous Web Attackers. In: 2016 IEEE First International Conference on Data Science in Cyberspace (DSC). pp. 222–229. IEEE, Changsha, China (Jun 2016). https://doi.org/10.1109/DSC.2016.78
17. Mozilla: addrs.c (Apr 2024), https://hg.mozilla.org/releases/mozilla-release/file/FIREFOX_125_0_3_RELEASE/dom/media/webrtc/transport/third_party/nICEr/src/stun/addrs.c#l66, accessed on 2024-07-12.
18. Mozilla Wiki: Media/WebRTC/Privacy. https://wiki.mozilla.org/Media/WebRTC/Privacy (Jan 2020), accessed on 2024-07-12.
19. Narten, T., Jinmei, T., Thomson, S.: IPv6 Stateless Address Autoconfiguration. Request for Comments RFC 4862, Internet Engineering Task Force (Sep 2007). https://doi.org/10.17487/RFC4862
20. Perta, V.C., Barbera, M., Tyson, G., Haddadi, H., Mei, A.: A glance through the VPN looking glass: IPv6 leakage and DNS hijacking in commercial VPN clients. PoPETs **2015**(1), 77–91 (2015). https://doi.org/10.1515/popets-2015-0006
21. Ramesh, R., Evdokimov, L., Xue, D., Ensafi, R.: VPNalyzer: Systematic Investigation of the VPN Ecosystem. In: Proceedings 2022 Network and Distributed System Security Symposium. Internet Society, San Diego, CA, USA (2022). https://doi.org/10.14722/ndss.2022.24285
22. Reiter, A., Marsalek, A.: WebRTC: your privacy is at risk. In: Proceedings of the Symposium on Applied Computing. pp. 664–669. SAC '17, Association for Computing Machinery, New York, NY, USA (Apr 2017). https://doi.org/10.1145/3019612.3019844
23. Revyakina, E.: Development of a secure video chat based on the WebRTC standard for video conferencing. E3S Web of Conferences **389**, 07017 (2023). https://doi.org/10.1051/e3sconf/202338907017
24. richard: Enable webrtc for base-browser (#41021) · Issues · The Tor Project / Applications / Tor Browser · GitLab (Jun 2022), https://gitlab.torproject.org/tpo/applications/tor-browser/-/issues/41021, accessed on 2024-07-12.
25. Roesler, D.: STUN IP Address requests for WebRTC (Jan 2015), https://github.com/diafygi/webrtc-ips, accessed on 2024-07-12.
26. StatCounter: Desktop browser Market Share Worldwide. https://gs.statcounter.com/browser-market-share/desktop/worldwide (Apr 2024), accessed on 2024-07-12.
27. Takasu, K., Saito, T., Yamada, T., Ishikawa, T.: A Survey of Hardware Features in Modern Browsers: 2015 Edition. In: 2015 9th International Conference on Innovative Mobile and Internet Services in Ubiquitous Computing. pp. 520–524. IEEE, Santa Catarina, Brazil (Jul 2015). https://doi.org/10.1109/IMIS.2015.72
28. The Chromium Project: network.cc (May 2024), https://webrtc.googlesource.com/src.git/+/refs/branch-heads/6478/rtc_base/network.cc#620, accessed on 2024-07-12.
29. Uberti, J.: WebRTC IP Address Handling Requirements. Request for Comments RFC 8828, Internet Engineering Task Force (Jan 2021). https://doi.org/10.17487/RFC8828
30. Uberti, J., Borst, J.D., Wang, Q., Fablet, Y.: WebRTC IP Address Handling Extensions for Multicast DNS. Internet Draft draft-uberti-ip-handling-ex-mdns-00, Internet Engineering Task Force (Nov 2018), https://datatracker.ietf.org/doc/draft-uberti-ip-handling-ex-mdns/00/, accessed on 2024-07-12.