To appear in ESSoS 2018
International Symposium on Engineering Secure Software and Systems

One Leak is Enough to Expose Them All
From a WebRTC IP Leak to Web-based Network Scanning

Mohammadreza Hazhirpasand, Mohammad Ghafari

Software Composition Group, University of Bern, Switzerland
{mhhazhirpasand, ghafari}@inf.unibe.ch

Abstract WebRTC provides browsers and mobile apps with rich real-time communications capabilities, without the need for further software components. Recently, however, it has been shown that WebRTC can be triggered to fingerprint a web visitor, which may compromise the user's privacy. We evaluate the feasibility of exploiting a WebRTC IP leak to scan a user's private network ports and IP addresses from outside their local network. We propose a web-based network scanner that is both browser- and network-independent, and performs nearly as well as system-based scanners. We experiment with various popular mobile and desktop browsers on several platforms and show that adversaries not only can exploit WebRTC to identify the real user identity behind a web request, but also can retrieve sensitive information about the user's network infrastructure. We discuss the potential security and privacy consequences of this issue and present a browser extension that we developed to inform the user about the prospect of suspicious activities.

Keywords: Web-based network scanner, IP leak, browser security and privacy

1 Introduction

Web browsers are amongst the most widely-used software applications to search and explore the Internet. A web browser is not just a container for web pages, but over the last few years it has been increasingly used for building cross-platform and hybrid apps in handheld devices. As a consequence, this common gateway to the Internet has been increasingly targeted by adversaries, and web-based attacks are becoming increasingly common [1].

Although modern browsers incorporate various protection techniques to achieve strong security [2], improvements in the features of these software applications may nevertheless compromise user privacy and introduce new security vulnerabilities [3]. In particular, the advent of HTML5 and CSS3 not only enables the construction of more diverse and powerful web sites and applications, but also brings on more risks especially to personal web privacy. For instance, the Canvas API enables the creation of animations and graphics in the web environment, but also facilitates the tracking of the user's browser and operating system due to different image processing engine and fonts settings [4]. The Geolocation

API provides the physical location of a user, thereby potentially compromising the user's privacy [5]. In general, HTML5 facilitates web browser fingerprinting wherein one collects various pieces of information to distinguish a user from the general flow of those who surf the Internet.

In this paper we focus on the risk posed by WebRTC, a set of HTML5 APIs that enable real-time communication over peer-to-peer connections, *e.g.*, in-browser calling and video conferencing. Previous research has shown that WebRTC leaks a web visitor's real identity even behind a VPN [6]. We build on the state of the art, and examine the extent to which this issue could jeopardize the security and privacy of internal members of a network. We propose a web-based network scanner that exploits the WebRTC IP leak to collect the network information. Unlike existing Javascript scanners that are often imprecise due to several assumptions about the network and browsers, our proposed scanner is both browser- and network-independent. It employs a few heuristics such as predicting a port status based upon the round-trip delay time, and reducing the scan time by using a pop-up window, *etc.*

We conduct various experiments in a private network of 20 active nodes. The network is running at 100 Mbp speed, and a FortiGate Unified Threat Management (UTM) protects the network from outside threats. We assess the IP leak in the latest versions of browsers such as Google Chrome, Mozilla Firefox, and Opera, and in various desktop and mobile operating systems. We also compare the performance of our web-based scanner with that of several system-based scanners.

We found that except on iOS, major browsers like Firefox and Chrome are subject to the WebRTC leakage issue regardless of the underlying operating system. In our experiment, Safari on Mac, and Microsoft Edge on Windows were the only browsers that support WebRTC but do not leak the IP address, and iOS was the only operating system that did not suffer from this issue in any of the tested browsers. Through our scanning approach we identified a great number of active hosts and open ports in a test network. For instance, within about 30 seconds we could scan a complete IP range and discover all active nodes, as well flag those whose `http` ports were open. Once we identified active nodes, within 23 seconds we could scan 7 open ports that disclose various network services and running applications in this private network. We acknowledge that our web-based scanner performs almost as fast and precisely as system-based scanners.

Consequently, WebRTC information leakage does not pose a threat only for a web visitor but also all other members in her network, and we conclude that the privacy risk imposed by WebRTC could be considered high. We developed a browser extension that shields Google Chrome and Mozila FireFox against the WebRTC IP leak without compromising its features and applications.

In the remainder of this paper, Section 2 presents some background and attack vectors through which an adversary can exploit WebRTC IP leakage. Section 3 explains our web-based port scanning approach, followed by our experiment and the obtained results in Section 4. Section 5 illustrates a few at-

tacks that we could run against nodes in a private network, and demonstrates a browser extension that we developed for protecting against such attacks. Section 6 presents an overview of the research most relevant to this work, and finally Section 7 concludes the paper.

2 Background

Using WebRTC every connected device, whether it is a computer, tablet, televisions, or smart gadget, can become a communication device without the installation of any third-party plug-ins. WebRTC is free, open source, and easy to use, and as of this writing, most browsers on different platforms from a desktop computer to a mobile device support WebRTC. Its broad adoption by many applications such as Google Hangouts, Facebook Messenger, Whatsapp, and Firefox Hello makes it hard for users to disable this feature [7]. Three HTML5 APIs, namely `getusermedia`, `rtcpeerconnection`, and `rtcdatachannel`, comprise the main part of WebRTC. To initiate a WebRTC application, browsers will request the user to grant permission to use WebRTC, and otherwise, the application would not run. This improves a user's privacy to ensure that their camera or microphone is not accessed from an untrusted web site.

Getting the private IP address of a website visitor used to be an arduous task especially as long as an adversary does not have physical access to the visitor's network, and the visitor uses TOR-like browsers, or is behind a VPN or an HTTP proxy. However, a web session in a WebRTC-powered browser may disclose critical network information without notice to an adversary even when the aforementioned privacy protection mechanisms are in place [8]. In effect, WebRTC needs to find the best path between two nodes. When establishing a peer-to-peer connection between two nodes, the private IP address of the user can be extracted with Javascript, from a Session Description Protocol (SDP) object. Consequently, WebRTC information leakage allows an adversary to silently identify the real IP address of a web visitor. This enables adversaries to scan open ports on her computer, collect information about running services and applications, and exploit a vulnerability in these programs to undertake an attack.

The WebRTC leakage issue is very similar to a recent information leakage discovered in Hotspot Shield, one of the most popular VPN providers with half a billion users.[1] First, they disclose some information regarding the network interfaces of a user. Second, both VPNs and proxies fail to protect the real IP address of the user. Third, the aforementioned information leaks without requiring any user permission. On the other hand, WebRTC as a part of HTML5 runs on nearly all major browsers on different platforms, and its privacy and security issues threaten a significantly large number of users in the globe.

There are at least three main attack vectors through which an adversary can attack a web client, and retrieve information from the victim. A seemingly

[1] `https://nvd.nist.gov/vuln/detail/CVE-2018-6460`

legitimate but *malicious website* can include a malicious script in a web response to the client and wait for the results. In a *man-in-the-middle* attack, an adversary secretly relays and possibly injects a script into the communication between two parties, and collects the scan results. In a *cross-site scripting* attack, lack of rigorous input validation allows an adversary to inject malicious scripts into otherwise benign and trusted websites.

3 Port Scanning

This section explains our approach using WebRTC IP leakage issue to scan private nodes within a network. It is developed in Javascript and is browser-independent.

Once the script starts to execute, it checks whether a target browser supports WebRTC or not. If not, some basic information such as the browser's user agent string, as well as the public IP address will be returned, and the execution process will terminate. Otherwise, if the acquired IP falls within the ranges of private IPv4 address spaces, we proceed with port scanning.

We adopt a scanning heuristic based on timing. Through several experiments in different networks we learned that examining the round-trip delay time (RTT) within a private network could be used to estimate the status of a node in the network. We collected the elapsed time and the response message to various connection requests in a network from the browser's console. We observed that the timing is pretty much the same in various browsers, yet depends on the network latency, which is not known a priori. On the other hand, we observed that within each network the timings fall into three time windows that we can associate with open port, refused connection, and timeout. An open port indicates that the node is active and has a running service on the queried port. When a firewall refuses the connection or the port is closed, a connection refused response is expected. This message implies that the queried node (*i.e.*, the IP address) exists in the network, whereas a timeout response means that there is no such node in the network. Such messages only appear in the browser's console and, for security and privacy reasons, Javascript cannot access them. Nevertheless, we consistently observed that the response to a successful connection (*i.e.*, open port) is received earlier than a refused connection, and always a timeout is received much later than the other two. We therefore propose to cluster the initial scanning results based on their timings to determine a reliable threshold for the status of a request in a private network.

We applied a clustering heuristic that works as follows. First, we sort the data points (*i.e.*, scanning results) in ascending order, and store the result in a set. Next, we compute the absolute distance between every two consecutive data points, and store them in another set. The top three values in the latter set present the three significant changes in the data. Therefore, the data points in the first set whose indices correspond to the indices of the top three values in the other set indicate the boundaries of the clusters, respectively.

The clustering heuristic is implemented in Javascript and runs only once on the client side to automatically guide the scanning in each network. Alternatively, the timings can be plotted on the server side and the adversary can manually set the proper boundary of each cluster.

We could therefore deduce the timing thresholds based on each cluster. For instance, Table 1 shows the obtained clusters in a full range IP scan for `http` ports within our test network. If the response time in our network is less than 500 ms, we assume the expected port is open. In case of a refused connection, a response is expected in between 700 to 1400 ms. A response timeout usually takes much longer (*i.e.*, on average above 18 seconds).

Table 1. The time span thresholds in ms, computed only for our network

Open port	Refused connection	Timeout
< 500	$700 << 1400$	$18000 <$

Making a connection request is feasible via an Image tag, a WebSocket request or an XMLHttpRequest (XHR). The first method uses an `<img>` tag in order to load a hypothetical picture from a remote location (*i.e.*, a node in the private network). We assume the expected port is open, only if the request fires the `onload` or the `onerror` events in the expected threshold. The second method uses the Websocket API, which enables interaction between a browser and a node in the network with lower overhead than `http`. In this method, we measure the timing as long as the the connection status is on the `connecting` state (*i.e.*, the `readyState` attribute is 0). The third method uses `XHR`, and we check the value of `statusText` attribute. Due to the restriction of the same-origin policy this value is `error` if there is a response from the remote location. Otherwise, In `XHR` a timeout response often takes 21 seconds and therefore we abort the connection if it takes more than the maximum threshold of interest (*i.e.*, 1400 ms).

The time required for port scanning is critical as a web session on a particular website may be short. We decreased the scanning time by scanning in parallel. In particular, we employed the popup window technique to split the number of IPs to scan between the main window and a very small popup window that we open in the rightmost corner of the screen and behind the main window. Such parallelism improves the scanning time up to 44% in total.[2] Moreover, when several clients within the same network are infected, we distribute the scanning amongst these nodes and aggregate the results in the end.

The scanning continues till it completes or as long as the victim is navigating within the infected page. We periodically persist the scanning results to local storage on the victim's browser to minimise data loss in circumstances where the scanning cannot proceed (*e.g.*, when the victim navigates to a different website, or closes the browser). Moreover, we realised when the infected tab is not active

[2] We send a new request every 200 ms.

or when a browser is minimized, the scanner may not perform as expected due to "background timer throttling" policies. In fact, in such circumstances both Chrome and Firefox enforce a timer task to run at most every 1000 ms.[3] We resume an incomplete scan once it is feasible (*e.g.*, when the victim visits the page again), and finally collect the network information.

4 Experiment

In this section, we conducted an experiment to find the IP address of a website visitor, and scanned nodes within the visitor's network. The visitor's network is a test lab designed to evaluate the performance of our port scanner. We simulated a malicious website by injecting a script (*i.e.*, our scanner) into the comment section of a website, so that when a visitor clicks on the "show comments" button, the script will run on the visitor's private network. We collected the private network information of the website's visitors in our experiment, and ran a few attacks against the internal nodes in the network.

We further evaluated the generalizability of our network scanning approach by carrying out another experiment in a wireless network with which we were not familiar in advance.

4.1 Setup

Table 2 presents an overview of 20 active machines in our first experimental network. We manually added Android and iOS devices to this network for experimentation purposes. The whole network is protected by a FortiGate Unified Threat Management (UTM) firewall with a default configuration. Five Windows machines in the network are also protected by Kaspersky products, and the remaining ones are equipped only with the default firewall installed on each machine.

Table 2. Test lab nodes and their open ports

OS	No of machines	Open ports
Windows	14	80,443,445,3306,110,1433,25,43,7,139,902
Linux	3	80,443,8080,22
MacOS	1	21,22
Android	1	2221
iOS	1	5612

We examined the WebRTC information leakage issue in the latest versions of major browsers in each operating system. In particular, we tested the Google Chrome, Chromium, Mozilla Firefox, Internet Explorer, Microsoft Edge, Safari,

[3] `https://developers.google.com/web/updates/2017/03/background_tabs`

Samsung Internet, and Opera browsers. We performed two tests; in the first test the scan goes from IP address 1 to 254 to identify active nodes in general, and in particular to flag web servers in the victim's private network. In the second test we collect more information about the target network by scanning a broader range of ports only on the active nodes that were identified in the previous test. Generally, scanning the whole range of ports could be time-consuming and users do not spend more than several minutes on a typical web page. Moreover, adversaries usually look for ports that are often associated to particular software systems that they can exploit. Therefore, it is common to scan only a number of ports on a victim's machine. Table 3 presents the list of ports that we scan in each node in the second experiment. We selected these ports randomly from the entire list of open ports in the network.

Table 3. The open ports in the target network

Port	Description
3306	MySQL database system
1433	Microsoft SQL Server database management system (MSSQL) server
8080	Apache Tomcat
902	VMware ESXi
2221	WiFi FTP Server android application
445	Windows shares
80	Hypertext Transfer Protocol (HTTP)

In another experiment, we also ran the same two tests via a browser connected to the wireless network. In both experiments we use our clustering heuristic which automatically determines the status of a network request. We compare our results with three popular IP/port scanning software tools, namely Angry IP Scanner[4], Advanced IP Scanner[5], and Advanced Port Scanner[6]. We set http port, banner grabber, and ping options in the first test. We repeat each test five times to mitigate bias in the measured performance.

There exist a few security tools that employ WebRTC to get the internal IP addresses of a victim's network. For instance, a tool named BeEF (`http://beefproject.com`) has a feature called "Get Internal IP Address", however, our experiment with this tool showed that it suffers from a large number of false positives so that, in the interest of space, we do not discuss it in this paper.

4.2 Result

Table 4 presents a list of popular browsers that were examined for the WebRTC private IP leak. We found that except on iOS, major browsers like Firefox and

[4] `http://angryip.org`
[5] `http://www.advanced-ip-scanner.com`
[6] `https://www.advanced-port-scanner.com`

Chrome are subject to the WebRTC private IP leak regardless of the underlying operating system. In our experiment, Safari on Mac, and Microsoft Edge on Windows were the only browsers that support WebRTC but do not leak the IP address, and iOS was the only operating system that did not suffer from this issue in any of the tested browsers.

Further investigation is needed into the implementation of these browsers to uncover the reason they do not leak the private IP address of a user.

Table 4. The state of WebRTC support in major browsers

Browser	**Version**	**OS**	**WebRCT**	**Private IP Leak**
Google Chrome	63	Windows	Yes	Yes
Mozilla Firefox	57	Windows	Yes	Yes
Internet Explorer	11	Windows	No	No
Microsoft Edge	16	Windows	Yes	No
Opera	49	Windows	Yes	Yes
Mozilla Firefox	54	Android	Yes	Yes
Google Chrome	59	Android	Yes	Yes
Opera Mini	32	Android	No	No
Samsung Browser	6.2	Android	Yes	Yes
Safari	11	Mac	Yes	No
Google Chrome	63	Mac	Yes	Yes
Mozilla Firefox	58	Mac	Yes	Yes
Safari	10	iOS	Yes	No
Google Chrome	64	iOS	Yes	No
Mozilla Firefox	10	iOS	Yes	No
Opera Mini	16	iOS	No	No
Mozilla Firefox	51	Ubuntu	Yes	Yes
Chromium	35	Ubuntu	Yes	Yes
Opera	51	Ubuntu	Yes	Yes
Google Chrome	64	Ubuntu	Yes	Yes

We successfully identified 20 active nodes and flagged 8 whose http ports were open in the first experiment. The average time to scan a complete range in all three methods in the main window of a browser is around 55 seconds. When the scanning is also assigned to a popup window the time significantly decreases to about 31 seconds (*i.e.*, 44% improvement).[7] We realised that the IMG tag approach may make the victim aware of ongoing suspicious activities. For example, in our experiment Firefox shows the IP addresses that are being loaded in the status bar.

[7] Distributing the work amongst more popup windows could improve the speed, but the risk that they will be noticed by the user increases as well.

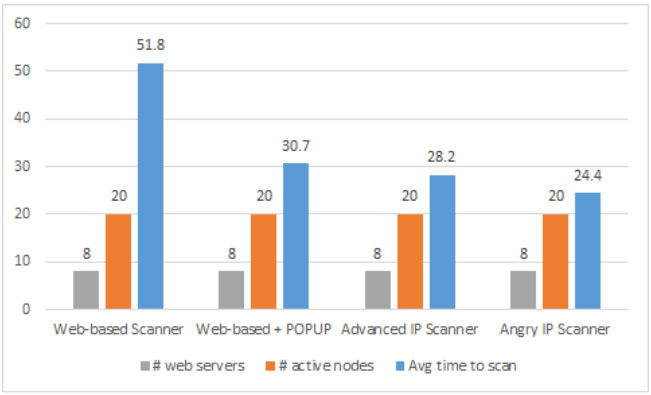

Figure 1. The performance of scanners in the first test, first experiment

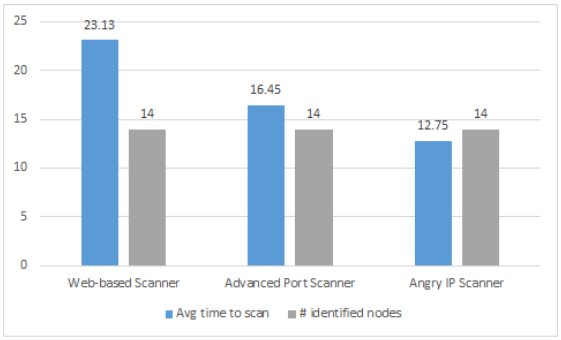

Figure 2. The comparison of scanners in the second test, first experiment

Figure 1 presents a comparison of our web-based scanner with two state-of-the-art network scanner tools. All the scanners are successful at identifying the nodes in the network. System-based scanners perform faster than our web-based scanner but only slightly faster when a popup window is employed. Moreover, system-based scanners have a higher privilege to scan all ports, perform ICMP requests, and use multi-threaded features. In contrast, in this experiment we found that due to security concerns Javascript is not allowed to communicate with all ports. In fact, out of 65535 ports, Javascript was banned from scanning 59 ports. These ports are listed in Table 8 in the Appendix. In addition, the performance of our scanner depends on the system resources that are allocated to the infected tab in a browser, as discussed in Section 3.

Figure 2 illustrates the results of the second test in which we scanned several ports listed in Table 3. The average time for scanning dropped significantly in all methods as we only scanned the active nodes *i.e.*, 20 machines.

Amongst 20 machines in the network, 14 had at least one open port. Table 5 presents the obtained results. The remaining six machines were not identified as none of their open ports were listed in Table 3.

Table 5. The identified open ports w.r.t. the ports listed in Table 3

	Open Ports	**OS**
1	80,443,3306	Windows
2	80,443,445,1433,3306	Windows
3	80,443,445,3306	Windows
4	80,443,445	Windows
5	445	Windows
6	902	Windows
7	80,443,445,3306	Windows
8	80,443,445	Windows
9	80,443,445	Windows
10	445	Windows
11	445	Windows
12	80,443	Linux (Ubuntu)
13	8080	Linux (Arch)
14	2221	Android

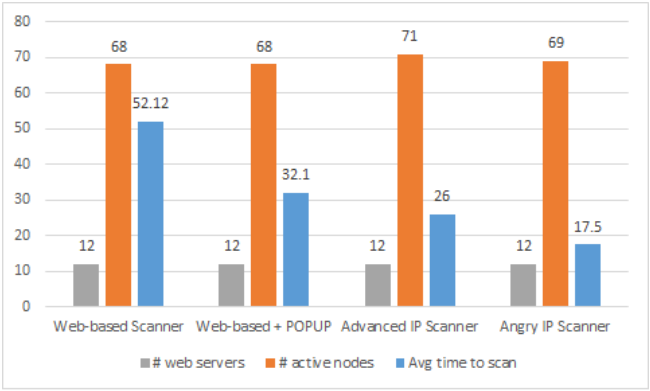

Figure 3. The performance of scanners in the first test, second experiment

In the following, we briefly discuss the results obtained in the second experiment within a wireless network. According to Figure 3, the performance of scanners in the first test is consistent with our findings in the previous experiment, though the Angry IP Scanner has performed slightly better than before. We found the number of identified active nodes varies a bit as few clients con-

nected or disconnected to the network during our experiment. In the second test we examined the status of the ports in Table 3, and amongst 68 active nodes we found 17 nodes with open ports (See figure 4). It is worth mentioning that the scan completion in our web-based scanner took much longer than the second test in the first experiment as there were about five times as many nodes, and we did not employ any parallelism.

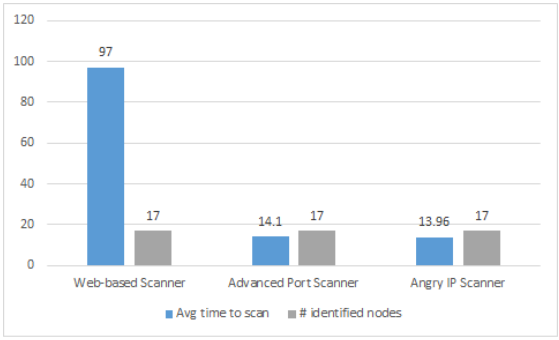

Figure 4. The comparison of scanners in the second test, second experiment

Therefore, the obtained results in our experiments confirm that our approach for network scanning performs reliably and independent of the underlying network.

4.3 Threats to Validity

The firewall in our test lab was set up with its default configuration, and did not prevent port scanning. In fact, perimeter firewalls often control the flow of network traffic entering or leaving an organization, whereas the origin of scanning is an internal node, and this bypasses many existing rules on firewall systems. Moreover, finding the right timing thresholds is a non-trivial task due to the network congestion, in-place firewall systems, and various policies like timing and resource allocation in browsers. Hence, a node whose response time takes longer than the selected thresholds may mistakenly be assumed to be unavailable. We mitigated this threat by scanning only when the infected tab is the active tab in a browser, and employing a clustering technique to predict a port status. Finally, our experiment was limited to IPv4 scanning and the obtained results may not generalise to IPv6.

5 Risks and Countermeasures

Besides serious privacy concerns that a remote network scanning imposes, the collected information facilitates a number of attacks against the nodes in a network. In this section, we briefly mention a few attacks that we conducted, and

thereafter, we present our simple approach, implemented as a browser extension, to protect against such attacks.

5.1 Attacks

We run a Denial of Service (DoS) attack against a web server in the network through the browser of a legitimate client in the network. This is interesting as security devices often underestimate the likelihood that such an attack may happen from inside a network. We use a web worker in Javascript, running in the background without interfering with the user interface, to flood the target web server with many XHR requests. The attack can be turned into a Distributed DoS attack if we can infect more browsers in the network.

Table 6 presents the maximum number of requests that we could send within one minute from various browsers to the target web server, without impacting the user experience and the browser responsiveness. We also measured the page load before and at the time of the attack, and found that it increases by 20% delay when running the attack.

Table 6. Number of requests from each browser during a DoS attack

Browser	Platform	Requests in one minute
Google Chrome	Windows	5820
Mozilla Firefox	Windows	5210
Opera	Windows	5442
Mozilla Firefox	Linux (Ubuntu)	3266
Google Chrome	Linux (Ubuntu)	4085
Mozilla Firefox	Android	4231
Google Chrome	Android	3005
Opera Mini	Android	3845

Next, we run a brute-force attack to mine sensitive files and directories from web servers. We selected 600 common paths, and measured the performance when targeting one, three, and five web servers. We checked the availability of a remote path via JSONP as it is able to establish GET requests without the restriction of the same-origin policy. In particular, JSONP expects a response only in JSON format, otherwise if the path exists it returns a `parsererror` message, and if it does not exist it returns an `error` message. Table 7 presents the obtained results.

Internal web applications often lack spam protection mechanism like Captcha code for authentication. Therefore, we can brute-force the login page in such applications. We speculated on the type of web applications installed on each web server based on the identified paths. In one case where we could identify the presence of a protected directory of image files, we remotely brute forced 1000 common passwords within 80 seconds. In particular, in each request we

Table 7. The path brute-force results in one minute

No of web servers	Examined paths	Identified paths
1	600	12
3	200	15
5	120	18

sent credentials as a POST request to the login page through a hidden `iframe` element within the infected page. As we are aware of a protected image, we load the image using the `IMG` tag. In case the image loads successfully, it means the session is set for the visitor, and the last tried password is correct.

5.2 WebRTC IP Leak Guard

The easiest way to shield against a WebRTC IP leak is to disable it in the browser. Extensions like WebRTC Block and WebRTC Control offer this possibility to a browser, however the risk is that users may not remember to disable WebRTC after each use. There exist a couple of browser extensions like WebRTC Leak Prevent, and WebRTC Network Limiter that allow only the use of public IP in WebRTC. Nevertheless, not all browsers support this feature as it may inhibit the adoption of many useful applications for which WebRTC is proposed.

We have developed a browser extension for Google Chrome and Mozilla Firefox that monitors the network information, and warns about requests that may be related to scanning and attacking internal nodes within a network. In particular, it records the connections destined to local IP addresses, and once the number of such connections exceeds a certain configurable threshold, the extension will notify the user of a suspicious activity. The extension has also a whitelist feature that contains addresses for Intranet applications and services that will not be monitored.

6 Related Work

In this section we review related work that studies WebRTC from a security and privacy standpoint. Recent research has highlighted the role of choosing the right browser and VPN in order to avoid WebRTC leakage [6]. The author found that TorGuard is the least privacy-compromising VPN service, while VyprVPN and ExpressVPN failed to prevent WebRTC IP leaks. Similar to our findings, he found Safari to be the most privacy-preserving browser. In a study of the top one million sites on Alexa [9], the authors found WebRTC being used to discover local IP addresses without user interaction on 715 sites, and mainly for tracking. In a research on web device fingerprinting [8], the authors classified 29 browser-based device fingerprinting techniques in which WebRTC is graded as a medium-level threat. A study of the fingerprintability of browsers found that WebRTC exposes identical device IDs of hardware components like webcam, microphone

and speaker across multiple browsers when visiting a particular website [10]. While the aforementioned studies mostly focused on WebRTC IP leakage for user fingerprinting, Reiter *et al.* [11] used WebRTC for conducting a couple of attacks like DDoS against a remote peer in the network. They also proposed the possibility of targeting internal nodes within a network by using a Javascript network scanner, named jslanscanner.[8] However, our investigation reveals that this scanner is bound to identify routers within a network by probing their well-known default IP addresses and under the assumption that some expected image files are available on these routers. Our experiments showed that this scanner produces a very large number of false positives due to its predefined threshold (*i.e.*, 15 seconds) for open ports, which varies in miscellaneous circumstances discussed earlier in Section 3.

To sum up, previous work mainly mentioned WebRTC information leaks, but their experiments were mostly limited to the choice of VPNs and browsers. Moreover, scanner tools like BeEF that use WebRTC suffer from a large number of false positives. We conjecture that this is due to their failure in adapting the appropriate timing with respect to a target network.

7 Conclusion

We focus on exploiting the WebRTC IP leakage issue for collecting critical information about a private network. In particular, we propose a web-based scanner that leverages this IP leakage to infiltrate a private network, and to discover active nodes and their open ports. The proposed scanner adopts a simple clustering algorithm to bypass the restrictions of previous web-based scanners that need to decide about the network latency a priory.

We compare our approach with state-of-the-art network scanners. Regardless of 59 ports that are banned from being scanned in Javascript, our web-based scanner performs only slightly slower than the system-based scanners. We briefly discuss several security implications of this issue, and introduce a browser extension that we developed for Chrome and Firefox for informing the user about such dubious activities in these browsers.

8 Acknowledgments

We appreciate the valuable feedback from Prof. Oscar Nierstrasz, as well as all parties who kindly allowed us to carry out several tests in their private networks. We gratefully acknowledge the funding of the Swiss National Science Foundations for the project "Agile Software Analysis" (SNF project No. 200020_162352, Jan 1, 2016 - Dec. 30, 2018).[9] We also thank CHOOSE, the Swiss Group for Original and Outside-the-box Software Engineering of the Swiss Informatics Society, for its financial contribution to the presentation of this paper.

[8] `https://code.google.com/archive/p/jslanscanner/`
[9] `http://p3.snf.ch/Project-162352`

Appendix

Table 8. The 59 ports that were banned for scanning via Javascript

Port	Assignment description
0	Reserved
1	TCP Port Service Multiplexer
7	Echo
9	Discard
11	Active Users
13	Daytime
15	Unassigned
17	Quote of the Day
19	Character Generator
20	File Transfer [Default Data]
21	File Transfer Protocol [Control]
22	The Secure Shell (SSH) Protocol
23	Telnet
25	Simple Mail Transfer
37	Time
42	Host Name Server
43	Who Is
53	Domain Name Server
77	any private RJE service
79	Finger
87	any private terminal link
95	SUPDUP
101	NIC Host Name Server
102	ISO-TSAP Class 0
103	Genesis Point-to-Point Trans Net
104	ACRNEMA Digital Imag. & Comm. 300
109	Post Office Protocol - Version 2
110	Post Office Protocol - Version 3
111	SUN Remote Procedure Call
113	Authentication Service
115	Simple File Transfer Protocol
117	UUCP Path Service
119	Network News Transfer Protocol
123	Network Time Protocol
135	DCE endpoint resolution
139	NETBIOS Session Service
143	Internet Message Access Protocol
179	Border Gateway Protocol - BGP
389	Lightweight Directory Access Protocol
465	URL Rendezvous Directory for SSM - Message Submission over TLS protocol
512	remote process execution; authentication performed using passwords and UNIX login names
513	automatic authentication performed based on priviledged port numbers
514	cmd like exec, but automatic authentication is performed as for login server
515	spooler
526	newdate
530	rpc
531	chat
532	readnews
540	uucpd
556	rfs server
563	nntp protocol over TLS/SSL (was snntp)
587	Message Submission
601	Reliable Syslog Service
636	ldap protocol over TLS/SSL (was sldap)
993	IMAP over TLS protocol
995	POP3 over TLS protocol
2049	Network File System - Sun Microsystems
4045	Network Paging Protocol
6000	X Window System

References

1. Zhang, M., Lu, S., Xu, B.: An Anomaly Detection Method Based on Multi-models to Detect Web Attacks. Computational Intelligence and Design ... (dec 2017) 404–409
2. Rogowski, R., Morton, M., Li, F., Monrose, F., Snow, K.Z., Polychronakis, M.: Revisiting Browser Security in the Modern Era: New Data-Only Attacks and Defenses. Proceedings - 2nd IEEE European Symposium on Security and Privacy, EuroS and P 2017 (2017) 366–381
3. Luangmaneerote, S., Zaluska, E., Carr, L.: Inhibiting Browser Fingerprinting and Tracking. Proceedings - 3rd IEEE International Conference on Big Data Security on Cloud, BigDataSecurity 2017, 3rd IEEE International Conference on High Performance and Smart Computing, HPSC 2017 and 2nd IEEE International Conference on Intelligent Data and Securit (2017) 63–68
4. Mowery, K., Shacham, H.: Pixel Perfect : Fingerprinting Canvas in HTML5. Web 2.0 Security & Privacy 20 (W2SP) (2012) 1–12
5. Yoon, S., Jung, J., Kim, H.: Attacks on Web browsers with HTML5. 2015 10th International Conference for Internet Technology and Secured Transactions, ICITST 2015 (2016) 193–197
6. Al-Fannah, N.M.: One Leak Will Sink A Ship: WebRTC IP Address Leaks. arXiv preprint arXiv:1709.05395 (2017) 1–12
7. Cox, J.H., Clark, R., Owen, H.: Leveraging SDN and WebRTC for Rogue Access Point Security. IEEE Transactions on Network and Service Management **14**(3) (2017) 756–770
8. Alaca, F., van Oorschot, P.C.: Device fingerprinting for augmenting web authentication. Proceedings of the 32nd Annual Conference on Computer Security Applications - ACSAC '16 (2016) 289–301
9. Englehardt, S., Narayanan, A.: Online Tracking: A 1-million-site Measurement and Analysis. Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security - CCS'16 (1) (2016) 1388–1401
10. Al-Fannah, N.M., Li, W.: Not All Browsers Are Created Equal: Comparing Web Browser Fingerprintability. (2017) 1–17
11. Reiter, A., Marsalek, A.: WebRTC: your privacy is at risk. Proceedings of the Symposium on Applied Computing - SAC '17 (In press) (2017) 664–669