Detection of JavaScript Injection Eavesdropping on WebRTC communications

Ahmed Osman, Raouf Abozariba, A. Taufiq Asyhari, Adel Aneiba, M. Ben Farah,
School of Computing and Digital Technology, Birmingham City University, UK
E-mail: {ahmed.osman, raouf.abozariba, taufiq.asyhari, adel.aneiba, mohamed.benfarah}@bcu.ac.uk

***Abstract*—WebRTC is a Google-developed project that allows users to communicate directly. It is an open-source tool supported by all major browsers. Since it does not require additional installation steps and provides ultra-low latency streaming, smart city and social network applications such as WhatsApp, Facebook Messenger, and Snapchat use it as the underlying technology on the client-side both on desktop browsers and mobile apps. While the open-source tool is deemed to be secure and despite years of research and security testing, there are still vulnerabilities in the real-time communication application programming interface (API). We show in this paper how eavesdropping can be enabled by exploiting weaknesses and loopholes found in official WebRTC specifications. We demonstrate through real-world implementation how an eavesdropper can intercept WebRTC video calls by installing a malicious code onto the WebRTC webserver. Furthermore, we identify and discuss several, easy to perform, ways to detect wiretapping. Our evaluation shows that several indicators within webrtc-internals API traces can be used to detect anomalous activities, without the need for network monitoring tools.**

***Index Terms*—WebRTC, XSS, SSH.**

I. Introduction

Web real-time communication is a set of protocols and APIs that enable real-time communication over peer-to-peer connections. In 2012, google released a browser-based WebRTC to allow video conferencing, file transfer, and screen sharing without the need to install external applications or plugins on the browser [1]. Since the inception of WebRTC, adoption in the tech community has increased dramatically. Google, Facebook, and Amazon are among larger technology companies that utilize WebRTC as the underlying technology in their products and services [2]. Furthermore, because this technology is an open-source tool, anyone can leverage it. Many university lectures, business meetings, and social events take place online, largely thanks to WebRTC-enabled browsers technology [3]. Thanks to its offered ultra-low latency streaming, WebRTC enables many applications and use cases in the smart city ecosystem as it does not only facilitate surveillance tasks and video calls, but various machine learning tools, essential for many detection solutions [4]. For example, smart city applications use WebRTC for transmitting videos to cloud servers where ML algorithms perform detection tasks [5]. An increasing number of these applications use simplified versions of WebRTC by developers, which increases the risk of exposure of vulnerable users, leading to the need for further studies and analysis. Confidentiality violations through covert channels and IP leaks were discussed in [6]. A description of end-to-end WebRTC security is given-in [7]. Many mitigation strategies were developed to tackle various threat models but eavesdropping threats remain largely understudied, both from theoretical and practical standpoints.

WebRTC-based applications can run through peer-to-peer communication and without intermedia servers [8]. However, often these applications will require WebRTC servers for many reasons. For example, when using WebRTC for many-to-many communication, the caller needs to know where to find the callee to create a connection, and this is where the signalling server is required. The signalling server sits in the middle of both peers, and it allows them to exchange session description protocol (SDP), a block of text that will enable the peers to establish a connection [9]. After this exchange, peers will know how to find each other and communicate without an intermedia server.

WebRTC's servers can be compromised like any other server through code injection [10]. For example, law enforcement or sophisticated hackers could attack the server or hire an insider in the host company, gaining access to the source code and injecting a malicious file that will allow them to create undetectable connections.

This study shows how one can intercept WebRTC communications. In our solution, the eavesdropper joins the intercepted call as a hidden third party when a WebRTC session is initiated through the webserver. Our attack model allows the eavesdropper to intercept without access to customers' end devices, which are harder to break into [11]. We use a standard unmodified WebRTC platform to implement our interception approach [12]. We exploit a loophole in the WebRTC specification, which allows us to set up a second connection to the eavesdropper (Eve) and duplicate the stream of user 1 (Alice) and user 2 (Bob), before sending it to Eve [13]. This paper's eavesdropping technique is based on unsolicited source code. Such attacks can lead to serious consequences such as loss of personal information and identity theft.

Many of eavesdropping attacks are unobservable through user interfaces (web browser or android applications), offering the opportunity to an attacker to eavesdrop on private data of WebRTC communications without drawing attention. On the other hand, WebRTC used in Machine-to-Machine communications on a large number of devices deployed remotely cannot be physically observed by administrators. Moreover, WebRTC

API does not offer auto-detection of eavesdropping features by default. Detecting the mentioned unauthorized eavesdropping requires access and examination of internal underlying processes and mechanisms of networks and devices.

An eavesdropper detector can be an essential building block of a secure connection. Unfortunately, to date, there has been a limited number of methods to detect eavesdroppers listening on WebRTC communications. Recent studies suggest the use of sniffing tools [14]. But such passive approaches involve monitoring network traffic and inferring outcomes based on those observations. A limitation in monitoring tools such as Tcpdump is that it is difficult to group and apply filters once starting to capture traffic, making it harder to create efficient dynamic eavesdropper detectors. Other notable problems with these methods include the need for middleboxes to sniff traffic, which introduces financial, power, and memory costs, unsuitable for many low power devices such as lightweight IoT terminals and single-board computers, commonly used in smart city applications. Putting ourselves in users' shoes, we propose three low-power browser-based WebRTC eavesdropper detection techniques. The key idea is based on the number of WebRTC connections and the observation that the data-rate of duplicated video is higher than that of unintercepted traffic. Through experiments, we analyze these detection techniques and provide an insight into their advantages and disadvantages.

Our main contributions are (i) developing a testbed in a form of a attack surface to demonstrate eavesdropping on real WebRTC communication involving two peers, enabling extraction of valuable data (ii) and using the observations from experiments to recommend new eavesdropping detection methods, different from the ones proposed in the literature.

The remainder of the paper is organised as follows. Section II details the state of the art of WebRTC. Section III provides a rigorous assessment of the vulnerabilities within WebRTC. Section IV presents the aim of our study while Section V focuses on the steps of the implementation. Section VI details our results. Conclusions and future work are drawn in Section VII.

II. Literature review

In only a few years, WebRTC became one of the most popular video streaming tools, emerging to become a common attack vector for hackers [15]. Many studies were published in the area of WebRTC security and contributed greatly to the API design [10], [16]–[18]. However, most of these studies primarily focus on providing an overview of WebRTC security threats without describing the implantation processes.

[10] introduced a model that allows governments to intercept WebRTC communications between two clients. The model provided in this study makes use of the client application to duplicate invitations sent through the webserver. For example, when Alice sends an invitation to Bob, the client application creates another invitation using RTCPeerConnection [13] and sends it to the law enforcement agencies (LEA). [10] discusses the model's limitations. It explains how users can detect interception using network monitoring tools such as Wireshark. However, the detection method presented in this paper has limitations. For example, while the tools are useful for analysing network packets, they are insufficient for analysing WebRTC packets such as Session Initiation Protocol (SIP). There are specialised analysers that are easier to use and can perform a comparative analysis of multiple calls to determine whether or not a WebRTC call is being eavesdropped on. This paper presents new methods for detecting eavesdropping on WebRTC communication using a specialised SIP analyser that includes the Google Chrome browser as a default. The benefits of our solution compared to the ones presented in [10] are, (1) if the user already has the Google Chrome browser installed, there is no need for them to install a third-party application, (2) Since the tools we use for our detection only focus on WebRTC packets, our solution consumes less CPU, memory, and costs compare to other network monitoring tools. (3) The solution we are using for detection allows further analysis of data gathered in graphs and tables. This does not include a lot of other network monitoring tools.

[16] discusses the threats to WebRTC components. It first describes the powerful WebRTC security protocols such as DTLS-SRTP and HTTPS then it explains how WebRTC is still vulnerable to client-side and server-side attacks. For example, the potential of WebRTC application users and servers can be hackable with cross-site scripting (XSS). The paper then explains the possible outcomes of these attacks, including what data hackers can obtain and actions they can do; while they have unauthorized access to the server.

Interceptions based on Voice over IP (VoIP) were discussed in [18]. In this study, the authors propose various methods of Lawful Interception based on the H.323 standard. As explained in the paper, the benefits of those methods are scalability, large traffic support, and flexibility to various configurations but the quality of experience is significantly degraded. Our method does not impact the QoE which makes it more difficult to detect by users. In this study, we implement the model described in [10] to perform tests and show other simple ways this model can be detected. Furthermore, we use the attacks mentioned in [16] to understand how to take over the client application and install the model described in [10].

III. Background

A. WebRTC

The two main WebRTC communication stages are (1) the signalling stage, which uses servers to connect peers, and (2) the communication stage, which is serverless and allows direct video and voice chat. There are other stages in WebRTC API (e.g., gateways, relays), however, this study focuses on simple WebRTC architecture involving only two users and a signalling server.

The signalling server facilitates the exchange of metadata between end-users [19]. It uses SDP to share the configuration between clients [19]. The signalling server can be set up using server technologies such as WebSocket, Socket.io, and SIP. The general process of exchanging metadata between end-users is shown in figure 1 and explained below.

1) Alice and Bob authenticate with the signalling server and wait for a response. If the authentication is successful, a communication process is initiated.
2) Alice creates and sends an SPD offer to the signalling server to communicate with Bob.
3) The signalling server receives Alice's SPD offer and forwards it to Bob.
4) Bob stores the SPD offer from Alice and creates SPD Answer in response to Alice.
5) The signalling server forwards the SPD Answer from Bob to Alice.
6) After completing the previous steps, Alice and Bob are aware of each other's media configuration. However, they are still unsure of how to connect. In this step, they exchange network information of each other (e.g. IP addresses and ports).
7) Using the network information from the previews step, direct communication is opened between Alice and Bob.

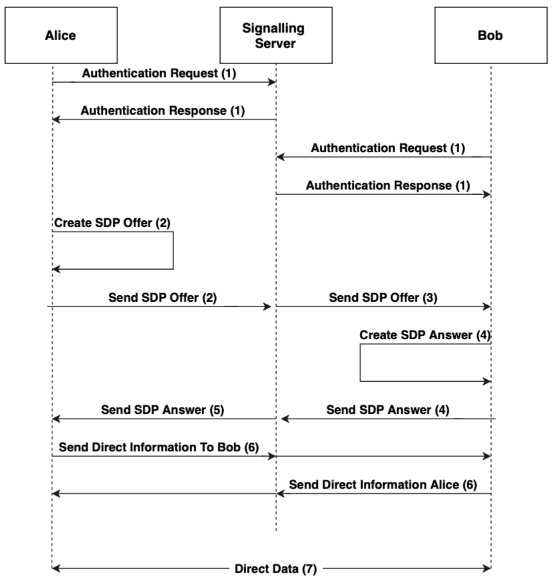

Alice
Signalling Server
Bob
Authentication Request (1)
Authentication Response (1)
Authentication Request (1)
Authentication Response (1)
Create SDP Offer (2)
Send SDP Offer (2)
Send SDP Offer (3)
Create SDP Answer (4)
Send SDP Answer (5)
Send SDP Answer (4)
Send Direct Information To Bob (6)
Send Direct Information Alice (6)
Direct Data (7)

Fig. 1: WebRTC communication steps (simplified)

Communication stage: Following the signalling stage, the two users begin re-validating each other. This process is achieved by re-sending the metadata obtained from previous signalling steps (e.g. ICE metadata) [20]. Unlike browser-based communication where a mediaserver is required, adding delay to the message delivery time, WebRTC is a server-less-based system and allows nodes to communicate directly without a server in the middle [21]. This approach eliminates the need to send your sensitive data to another company server, which may or may not protect it.

B. Cross-Site Scripting Attacks (XSS)

Cross-Site Scripting or XSS is a type of injecting attack in which the attacker exploits vulnerabilities in a website to execute malicious code on a visitor's browser. Attackers use XSS to force web browser interpreters from a data context to a code context [22]. For example, if a page has HTML input form, a hacker can use that form to enter a JavaScript code using the $\langle script \rangle$ tag [23]. If the form does not filter such tags, an attacker will be able to obtain the local and remote stream of the victims. As shown in figure 2, a malicious third-party JavaScript code can set up a hidden WebRTC connection to the Eve, then duplicate the stream of Alice and Bob before sending it to Eve. This type of attack allows the eavesdropper to listen in on Alice and Bob's conversations without their knowledge.

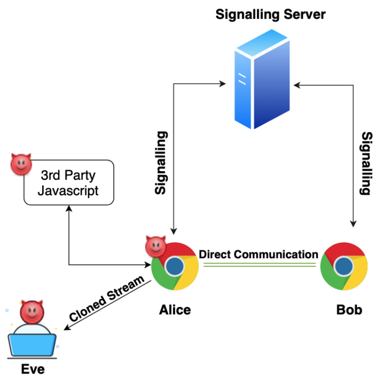


Fig. 2: A practical XSS attack on the server-side.

There are three main types of XSS attack, reflected, Dom-based, and stored [24]. Both reflected and Dom-based attacks are less severe than stored attacks because they require tricking the victim into clicking a vulnerability (e.g., link). However, in-store attack, the hacker injects a malicious code into a webserver through data inputs, such as chat or comment box. Every time a user requests this data, the injected malicious code will execute on the victim's browser. This type of attack is more severe because it allows the hacker to execute malicious code without tricking the victim into clicking a vulnerability [25].

The most common mitigation against XSS attacks is to use a good coding practice that validates and checks inputs. For example, if the input box asks for age, the user should only enter numbers. If they input anything else, the server should refuse to save the data. Another method of defending against XXS attacks is telling the web browser that the input data should only be viewed as data and not be interpreted in any other way (e.g., HTML rendering) [22].

C. SSH brute force attack

A hacker can use Secure Shell (SSH) brute force attack [26] to compromise WebRTC servers and perform actions such as altering the data stream, cloning it, and sending it to an invisible user in a video conference [16]. This method allows the hacker to attempt all the combinations for a password until he or she obtains the administration password for the server [27].

It may seem they are unlikely to ever succeed. However, suppose the site is not configured correctly, and a poor password is used. In that case, many advanced programs like John the Ripperand and Coin & Abel will increase the success rate of brute force attacks [28], [29]. After the hacker obtains the administration password, they can use it to log in to the WebRTC server and edit the application source code in a way that will enable them to duplicate calls without the user noticing.

The most common defence against brute force attacks is to do host level detection such as checking log files, number of incorrect tries and SSH port [30]. New studies, such as [27], use machine learning to detect SSH brute force attacks on the network level. It achieves this by training a model using 24 hours of real-world network data labelled as "brute force attack" and "not a brute force attack". The result of this paper shows that machine learning can successfully detected brute force attacks with a high detection rate and a low false alarm rate.

IV. Methodology

This study aims to demonstrate that, even though WebRTC is a secure project, it still has some weaknesses today. To show this, we first developed a WebRTC application that allows two users on different laptops to communicate. Then, we place a malicious code on the webserver, allowing an eavesdropper to intercept calls made through it. How the malicious code is placed on the webserver is out of the scope of this study. However, in the background section, We discussed how a hacker can use XSS and SSH brute force attacks to install malicious code on a server.

Our application uses WebRTC API components to deliver communications between users. GetUserMedia, PeerConnection, and RTCSessionDescription are some of the WebRTC components we used. The GetUserMedia API is responsible for providing access to the video, audio, or both from the local devices to the peer-to-peer communication. The other two APIs are responsible for sending and receiving media. They also handle SDP negotiation, packet losses, and other network problems.

Our malicious code is JavaScript code-based, and it allows the hacker to receive media from victims while not sending media to them. This allows the hacker to be in the video conference but remains undetected, at least for non-expert users. For example, when someone logs in to the site, the GetUserMedia API accesses their camera and sends it through the PeerConnection API. However, when a hacker logs in as Eve, they will only get streams from the victims without sending them on their own. This process is explained in more detail in the following section.

V. System Implementation

A. Requirements

Functional requirements: The main functional requirement for this system is to allow users to log in using their names. If the login name is not Eve, the GetUserMedia API should access their camera and send it to the other users who are already logged in. However, If the login name is Eve, the GetUserMedia API access should not be granted, and communication should be one-way (e.g. from victims to hackers). Also, any reference to the connection made with Eve should be removed from the victim's dashboard.

Non-Functional Requirements: Our non-functional requirements include: (1) allowing up to 4 users to communicate at the same time, (2) ensuring Eve can intercept all calls made through the webserver, and (3) taking steps to make sure that the overall call acts the same when intercepted and when it is not.

B. Code implementation

Our main JavaScript code is placed on the client-side. We use this code to establish and intercept calls between Alice and Bob. When the home page loads, the user is prompted to enter their login name. If the name is not Eve, a random ID is assigned. However, if it is Eve, a static ID is issued. Next, we grab the local video stream from all users except Eve (Eve is an eavesdropper, so they do not need to share their video). Once we have a successful local video stream, we send a message to connected users, informing them that we have joined the video conference. The following simplified steps take place after this.

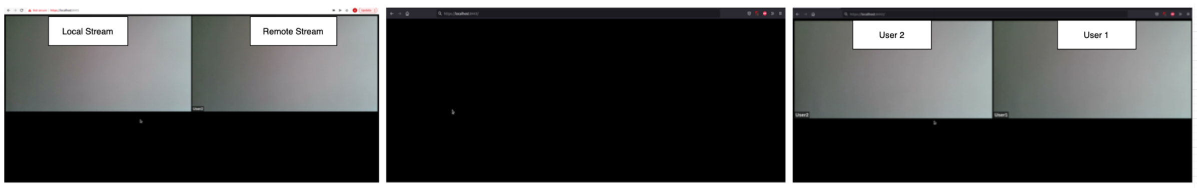
Alice will send an invitation to Bob. However, instead of only sending one invite to bob, our malicious code sends two invites, one to Bob and the other one to Eve. When Bob receives the invitation from Alice, he will create a response to Alice's offer while simultaneously inviting Eve. When Eve receives both of Alice's and Bob's offers, it confirms and sends responses without including her video stream. Following this, the video conference begins, and Eve joins as a hidden user.

C. System components

The core components of the proposed system are the signalling server and webserver.

Signalling server: To initialize a new WebRTC connection, a handshake or signalling process is required. During this phase, nodes exchange connection information to reach each other. During the call setup, Alice, Bob, and Eve share through signalling server the SDP offer, SDP answer, and Interactive Connectivity Establishment (ICE) candidates. These messages are exchanged in plain text and can be sent using any transport protocol. WebSocket was chosen for this study since it is faster than HTTPS [31]. Another reason we pick WebSocket is that it enables us to send data to users without them requesting it. This is important for WebRTC applications because the client needs to know quickly when a change in WebRTC session information occurs. The connecting steps are described in [17].

Webserver: The webserver frontend comprises JavaScript, HTML, and CSS, while Node.js host the backend with Express. The primary function of the webserver is to enable WebRTC communication. However, it also used to allow Eve to intercept calls made through this server. For example, the source code allows users to communicate without hidden users in the call. When the source code is replaced with a malicious code, calls made through this server are duplicated.



(a) User Interface During Call (b) Eavesdropper Interface Without Calls (c) Hacker Interface With Calls

Fig. 3: Example of system user interface for both interceptor and victims.

Our WebRTC interception technique assumes that this server is unsolicited. If steps are taken to check the source code, an experienced programmer will notice that there is malicious code that is duplicating the calls. To avoid this, we take steps such as placing the malicious code in a hidden folder rather than inside the source code file. The HTML file is then linked to the malicious code file instead of the source code file. This reduces the likelihood of detection, but it can still be detected if the programmer takes a careful look at HTML file script links, as we will show later.

D. *System interface*

We use a node.js signalling server to implement a simple WebRTC application. We started with this application to allow users to perform real-time communication, such as video and audio conferencing. After achieving this, we investigated whether we could hack the calls made through this application without the users knowing. Figure 3a shows the user interface, which includes a local and remote video stream. The local stream is the user's camera, while the remote stream is the person the user is communicating with. As more users join the call, there will be more remote streams. In this study, we only use two peers per connection.

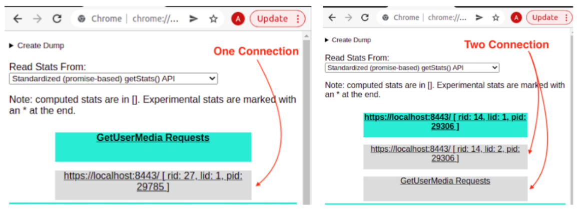
Figures 3b and 3c show the eavesdropper's interface. Figure 3b shows a case where no call is made through the server. In this situation, the eavesdropper interface will be empty, and no streams will be sent from victims. However, as soon as someone makes a call, the eavesdropper receives the stream of that call (Figure 3c). The user interface does not provide indications to the users that their call is being listened to by another user.

VI. Results and Discussion

A. *Number of active connections*

Our malicious code allows us to duplicate every invitation sent through the client application. For example, when Alice sends an invitation to Bob, our malicious code creates another invite and sends it to Eve, thereby creating a hidden connection. An experienced user can detect this leak by looking at how many WebRTC connections are created in the call. There are several ways a user can achieve this. As described in [10], network mentoring tools such as Wireshark can be used to find the hidden connection. However, these tools do not come as default, and they require a user to download it before they can use it. Also, they demand more CPU and memory while monitoring the network.

In this paper, we show easy to perform and accurate methods for detecting a hidden WebRTC connection. We explore webrtc-internals, which is built-in API. It can simply be accessed by opening up a new tab and entering the following protocol and URL (chrome://webrtc-internals). If Alice uses this tool while communicating with Bob and her call is not intercepted, this tool will show her that only one connection has been created, which is from Alice to Bob (figure 4a). However, if her call is intercepted with the malicious code, there will be two connections: one to Bob and the other one to Eve (figure 4b). This method is only useful when the number of connections is known to the user a priory. In situations where there are many peers on the link, with users entering and leaving the call, the number of legitimate users can not be an indication of eavesdropping activity. Furthermore, this tool is only available in the Chrome browser; Firefox has a similar tool [32], but it is not as detailed as webrtc-internals.



▸ Create Dump

Read Stats From:
Standardized (promise-based) getStats() API

Note: computed stats are in []. Experimental stats are marked with an * at the end.

GetUserMedia Requests

https://localhost:8443/ [rid: 27, lid: 1, pid: 29785]

▸ Create Dump

Read Stats From:
Standardized (promise-based) getStats() API

Note: computed stats are in []. Experimental stats are marked with an * at the end.

https://localhost:8443/ [rid: 14, lid: 1, pid: 29306]

https://localhost:8443/ [rid: 14, lid: 2, pid: 29306]

GetUserMedia Requests

(a) Connection without malicious code (b) Connection with malicious code

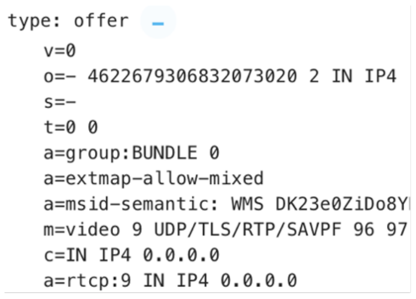
Fig. 4: WebRTC peer connection indicator.

B. *SDP offer*

As explained in the background section, every time a new user joins a call, an SDP offer and answer is exchanged between them and other users on the call. In this test, we investigate how a victim might detect eavesdropping activities by looking at how many SDP offers are sent over the network. Our results show that if the victim observes WebRTC-internals tool, they will see how they are sending more SDP offers than they are supposed to. For example, suppose they are just calling one person; instead of sending one offer to that person (fig. 5), they will see how they are sending another one to a hidden user (fig. 6). Again, this detection method is not valid when there are numerous peers on the link with users randomly entering and leaving the call.

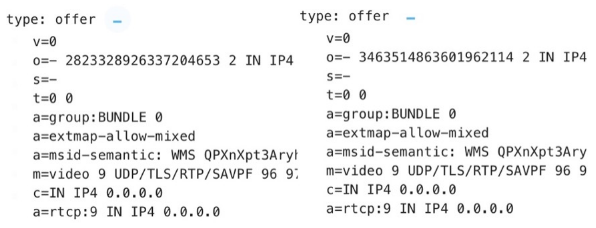
C. *Source code*

According to our results, a user can detect eavesdropping by right-clicking the page and selecting the Inspect option.



```
type: offer
    v=0
    o=- 4622679306832073020 2 IN IP4
    s=-
    t=0 0
    a=group:BUNDLE 0
    a=extmap-allow-mixed
    a=msid-semantic: WMS DK23e0ZiDo8Y
    m=video 9 UDP/TLS/RTP/SAVPF 96 97
    c=IN IP4 0.0.0.0
    a=rtcp:9 IN IP4 0.0.0.0
```

Fig. 5: An example of an SDP offer without malicious code.



```
type: offer
    v=0
    o=- 2823328926337204653 2 IN IP4
    s=-
    t=0 0
    a=group:BUNDLE 0
    a=extmap-allow-mixed
    a=msid-semantic: WMS QPXnXpt3Ary
    m=video 9 UDP/TLS/RTP/SAVPF 96 9
    c=IN IP4 0.0.0.0
    a=rtcp:9 IN IP4 0.0.0.0
```

```
type: offer
    v=0
    o=- 3463514863601962114 2 IN IP4
    s=-
    t=0 0
    a=group:BUNDLE 0
    a=extmap-allow-mixed
    a=msid-semantic: WMS QPXnXpt3Ary
    m=video 9 UDP/TLS/RTP/SAVPF 96 9
    c=IN IP4 0.0.0.0
    a=rtcp:9 IN IP4 0.0.0.0
```

Fig. 6: An example of an SDP offer with malicious code, presenting two simultaneous offers.

This will enable them to see the HTML and JavaScript code that makes up the page, and if experienced, they will be able to differentiate the malicious JavaScript code from the source code JavaScript. For example, they will notice how the `localUID2` variable creates a new user statically every time a user logs in. They will also notice an `if` statement that sends the stream of one user (the victims) but does not send the stream of the other user (the eavesdropper). As we explained in the System components subsection, steps such as separating the malicious and source code can help to reduce the detection of the malicious code. However, the malicious code must be linked to the source code through script links, which can be used to detect the malicious code. Snippet of the malicious code is shown figure 7.

D. Video Bit-rate

An expert user can detect the existence of eavesdropping by looking at how much data is sent. Using the webrtc-internals, we can compare the bitrate with and without the malicious code. According to our results, when two users communicate around 5 minutes without the malicious code, the total bitrate sent was 472,041 Kbits, with an average of 1,627 Kbits per second. During the same period, when the interception was used, this increased to 966,184 Kbits total and 3,233 Kbits per second average. As shown in figure 8, when the malicious code is not used, only one outbound connection is made to the other person, i.e., from victim to User1 (see Fig.8a). However, if the malicious code is used, two outgoing connections are created (cf. Fig.8b): one connection is from victim to User1; and the other connection is to eavesdropper. While other factors such as video quality links can increase and decrease the bitrate, machine learning tools with labeling services such as *VirusTotal*, utilizing historical data, can be used to detect

```
function start(){
localUID = createUID();
localUID2= "398472984729847284"}
........
if(localDisplayName == "Eve"){
localluid = localuuid2
var remove div =
 ↪  document.getElementById("localVid
eocontainer");
remove div.remove();
}
.........
if(localuuid == localuuid2){
} else{
peerConnections
 ↪  (peerluid).pc.addStream(localStream);
}
```

Fig. 7: Snippet of the malicious code

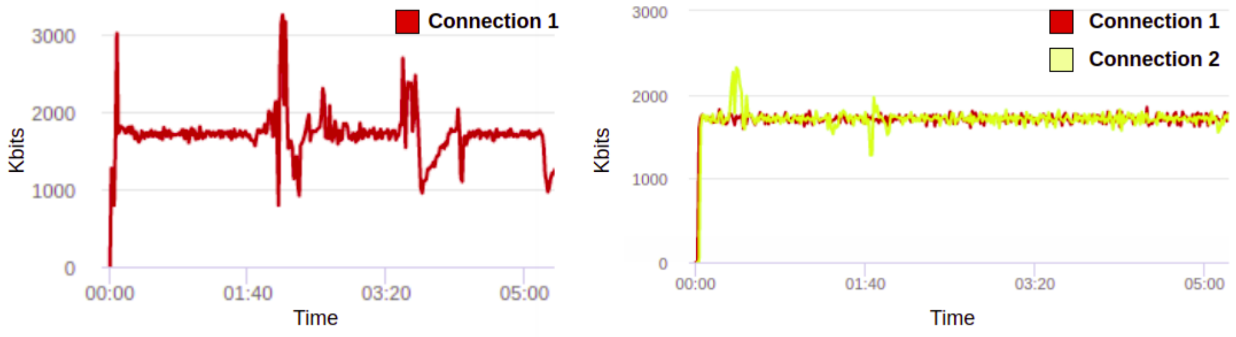
this eavesdropping attack. The impact of classification changes over time which introduces what is known as *label shift* can be handled by delaying labeling until a steady ground-truth is present, minimizing performance degradation.

VII. Conclusion

This research aimed to demonstrate that while WebRTC offers many features when it comes to privacy and security, it is not immune to direct attacks on its components and vulnerabilities in its underlying host applications. To show these vulnerabilities, we created a real-world prototype application that allows a hacker or government agency to intercept WebRTC calls while remaining undetected by the victim, provided he or she does not use any network WebRTC monitoring tools.

Another aim of this paper was to show WebRTC users how they can detect when their calls are being eavesdropped on. We tested various detection methods against the system implemented in this paper. Our results show that the proposed system can be detected if the intercepted user is experienced with monitoring tools. For example, if they know how to use the WebRTC tools included in Google Chrome, they will see how they are connected to more connections than they are supposed to. Furthermore, they can use these tools to investigate SDP offers, allowing them to detect when they are sending more SDP offers than they are supposed to. We also demonstrated in this study how Chrome DevTools such as inspect can be used to detect eavesdropping by analyzing the HTML and JavaScript that comprise the page which contains the malicious code. Leveraging these indicators, one can create effective auto-detection tools.

This work can be extended in many directions. For example, we plan to use machine learning to detect when WebRTC calls are intercepted. Currently, we are using multiple tools to do the detection. This is time-consuming and nonuser-friendly. Our



(a) **Bitrate Without Malicious Code**

(b) **Bitrate With Malicious Code**

Fig. 8: Example of Bitrate with and without Malicious Code.

future detection solution includes aggregating data from all these tools and using machine learning to determine whether the call is intercepted. Other future work can also include actions. For example, suppose the ML detects that a call is being intercepted. In that case, it can be programmed to remove the hidden connection, blocking the hacker's IP address, among other actions.

REFERENCES

[1] C. Alexandru, "Impact of webrtc (p2p in the browser)," *Internet Economics VIII*, vol. 39, 2014.

[2] "WebRTC — The technology that powers Google Meet/Hangout, Facebook Messenger and Discord," https://medium.com/swlh/webrtc-the-technology-that-powers-google-meet-hangout-facebook-messenger-and-discord-cb926973d786, accessed Dec. 20, 2021.

[3] P. M. D. Faye, A. D. Gueye, and C. Lishou, "Virtual classroom solution with webrtc in a collaborative context in mathematics learning situation," in *Innovation and interdisciplinary solutions for underserved areas*. Springer, 2017, pp. 66–77.

[4] A. Hussain, B. Barua, A. Osman, R. Abozariba, and A. T. Asyhari, "Low latency and non-intrusive accurate object detection in forests," in *2021 IEEE Symposium Series on Computational Intelligence (SSCI)*, 2021, pp. 1–6.

[5] A. Osman, R. Abozariba, A. T. Asyhari, A. Aneiba, A. Hussain, B. Barua, and M. Azeem, "Real-time object detection with automatic switching between single-board computers and the cloud," in *2021 IEEE Symposium Series on Computational Intelligence (SSCI)*, 2021, pp. 1–6.

[6] D. McMeekan, *Securing WebRTC*. Western Illinois University, 2021.

[7] R. L. Barnes and M. Thomson, "Browser-to-browser security assurances for webrtc," *IEEE Internet Computing*, vol. 18, no. 6, pp. 11–17, 2014.

[8] C. Vogt, M. J. Werner, and T. C. Schmidt, "Leveraging webrtc for p2p content distribution in web browsers," in *2013 21st IEEE International Conference on Network Protocols (ICNP)*. IEEE, 2013, pp. 1–2.

[9] "SDP: Session Description Protocol," https://datatracker.ietf.org/doc/html/rfc4566, accessed Jan. 10, 2022.

[10] A. Wagner and R. Puzis, "Lawful interception in webrtc peer-to-peer communication," in *International Symposium on Cyber Security Cryptography and Machine Learning*. Springer, 2021, pp. 153–170.

[11] "Is hacking getting harder?" https://www.sciencefocus.com/science/is-hacking-getting-harder/, accessed Feb. 28, 2022.

[12] S. Dutton *et al.*, "Getting started with webrtc," *HTML5 Rocks*, vol. 23, 2012.

[13] "RTCPeerConnection ," https://developer.mozilla.org/en-US/docs/Web/API/RTCPeerConnection, accessed Feb. 28, 2022.

[14] W. De Groef, D. Subramanian, M. Johns, F. Piessens, and L. Desmet, "Ensuring endpoint authenticity in webrtc peer-to-peer communication," in *Proceedings of the 31st Annual ACM Symposium on Applied Computing*, 2016, pp. 2103–2110.

[15] C. Alexandru, "Impact of webrtc (p2p in the browser)," *Internet Economics VIII*, vol. 39, 2014.

[16] B. Feher, L. Sidi, A. Shabtai, and R. Puzis, "The security of webrtc," *arXiv preprint arXiv:1601.00184*, 2016.

[17] A. Reiter and A. Marsalek, "Webrtc: your privacy is at risk," in *Proceedings of the Symposium on Applied Computing*, 2017, pp. 664–669.

[18] A. Milanovic, S. Srbljic, I. Raznjevic, D. Sladden, I. MatoSevic, and D. Skrobo, "Methods for lawful interception in ip telephony networks based on h. 323," in *The IEEE Region 8 EUROCON 2003. Computer as a Tool.*, vol. 1. IEEE, 2003, pp. 198–202.

[19] H. Bhardwaj, A. Lunthi, H. Bhat, K. S. Rawat, and A. Chabbra, "Real time information and communication center based on webrtc," 2020.

[20] B. Sredojev, D. Samardzija, and D. Posarac, "Webrtc technology overview and signaling solution design and implementation," in *2015 38th international convention on information and communication technology, electronics and microelectronics (MIPRO)*. IEEE, 2015, pp. 1006–1009.

[21] A. Andujar and C. Medina-López, "Exploring new ways of etandem and telecollaboration through the webrtc protocol: Students' engagement and perceptions." *International Journal of Emerging Technologies in Learning*, vol. 14, no. 5, 2019.

[22] M. Johns, "Code-injection vulnerabilities in web applications—exemplified at cross-site scripting," 2011.

[23] "The Script element," https://developer.mozilla.org/en-US/docs/Web/HTML/Element/script, accessed Jan. 10, 2022.

[24] Z. Jingyu, H. Hongchao, H. Shumin, and L. Huanruo, "A xss attack detection method based on subsequence matching algorithm," in *2021 IEEE International Conference on Artificial Intelligence and Industrial Design (AIID)*. IEEE, 2021, pp. 83–86.

[25] J. Kaur and U. Garg, "A detailed survey on recent xss web-attacks machine learning detection techniques," in *2021 2nd Global Conference for Advancement in Technology (GCAT)*. IEEE, 2021, pp. 1–6.

[26] "The Secure Shell (SSH) Transport Layer Protocol," https://www.ietf.org/rfc/rfc4253.txt, accessed Jan. 20, 2022.

[27] M. M. Najafabadi, T. M. Khoshgoftaar, C. Kemp, N. Seliya, and R. Zuech, "Machine learning for detecting brute force attacks at the network level," in *2014 IEEE International Conference on Bioinformatics and Bioengineering*. IEEE, 2014, pp. 379–385.

[28] "John the Ripper password cracker," https://www.openwall.com/john/, accessed Jan. 20, 2022.

[29] "Password cracking using Cain & Abel," https://resources.infosecinstitute.com/topic/password-cracking-using-cain-abel/, accessed Jan. 20, 2022.

[30] J. Owens and J. Matthews, "A study of passwords and methods used in brute-force ssh attacks," in *USENIX Workshop on Large-Scale Exploits and Emergent Threats (LEET)*, 2008.

[31] V. Pimentel and B. G. Nickerson, "Communicating and displaying real-time data with websocket," *IEEE Internet Computing*, vol. 16, no. 4, pp. 45–53, 2012.

[32] "New Tool for Debugging WebRTC," https://blog.mozilla.org/webrtc/new-tool-debugging-webrtc/, accessed Feb. 28, 2022.