Security in WebRTC Peer-To-Peer connections and knowing who you're talking to

Marc André Matija
info@casqan.net
Hochschule Rhein-Main
Wiesbaden, Germany

***Abstract*—In the era of real-time web applications, technologies like WebRTC and Peer-to-Peer (P2P) communication have become essential components of modern web infrastructure. WebRTC enables direct, decentralized communication between devices, facilitating real-time audio, video, and data exchange without relying on centralized servers. This paper examines the growing significance of P2P technologies, their inherent advantages over the server-client model, and the unique security challenges they present. Unlike traditional architectures, P2P systems demand trust establishment between endpoints in dynamic, transient environments, necessitating robust mechanisms for encryption, authentication, and integrity. Through an exploration of WebRTC's underlying technologies—such as STUN, TURN, and DTLS-SRTP—and its security framework, this paper highlights how WebRTC ensures secure communication while addressing vulnerabilities like man-in-the-middle attacks and IP address leaks. This study underscores the critical role of security in the adoption and reliability of WebRTC and P2P technologies in shaping the future of real-time communication.**

***Index Terms*—Peer-To-Peer, WebRTC**

I. Introduction

In recent years, the proliferation of real-time applications such as video conferencing, online collaboration tools, multiplayer gaming, and decentralized content-sharing platforms has brought Peer-To-Peer (P2P) communication technologies to the forefront of modern web infrastructure. At the heart of this evolution lies Web Real-Time Communication (WebRTC), a framework that enables seamless audio, video, and data exchange directly between devices without relying on centralized servers for data transmission. WebRTC's integration into browsers and its adoption in applications like Google Meet, Zoom, and Discord have cemented its position as a cornerstone of real-time communication on the web [19]. The importance of P2P technologies stems from their inherent advantages over the traditional server-client model. By enabling direct communication between peers, P2P systems reduce the dependency on centralized servers, thereby lowering costs, improving scalability, and reducing latency [13]. These characteristics make P2P well-suited for scenarios requiring high performance and efficiency, such as low-latency video conferencing and large-scale decentralized networks. Additionally, as edge computing and decentralized ecosystems, including blockchain technologies, continue to grow, P2P communication is poised to play a critical role in shaping the future of the web [20]. However, the adoption of P2P technologies introduces unique security challenges that differ fundamentally from the established server-client architecture. In the server-client model, security largely revolves around protecting a central server, with mechanisms such as TLS ensuring secure communication between clients and the server. In contrast, P2P systems require trust to be established directly between endpoints, often in dynamic and transient network environments. This shift necessitates the use of mechanisms like end-to-end encryption, peer authentication, and data integrity validation to ensure secure interactions without the intermediary role of a central authority. Additionally, P2P architectures must contend with new vectors of attack, such as man-in-the-middle attacks during peer discovery, unauthorized data relay via TURN servers, and exploitation of NAT traversal mechanisms like STUN.

II. What WebRTC does and how it works

WebRTC is a web standard that facilitates a Peer-To-Peer connection between browsers or native clients. In order to do so, a WebRTC connection is required, which involves several steps to enable secure Peer-To-Peer communication for audio, video, and data. The process begins with signaling, where peers exchange Session Description Protocol (SDP) messages and ICE candidates via a signaling server (e.g., WebSocket or SIP). These exchanges negotiate media codecs, encryption keys, and network details [12].

In order to facilitate Peer-To-Peer connections WebRTC makes use of the following technologies:

- **Network Address Translation (NAT):** NAT is a method used in computer networking to map multiple private IP addresses within a local network to a single public IP address. This is typically done by a router or gateway, allowing devices in private networks to communicate with external networks, such as the internet, while conserving the limited pool of IPv4 addresses. NAT plays a crucial role in modern networking due to the IPv4 address scarcity and the widespread adoption of private networks [14].
- **STUN (Session Traversal Utilities for NAT):** Allows a device to discover its public-facing IP and port as seen by the external network. This information is shared with

the other peer during signaling, enabling them to attempt a direct connection [14].

- **TURN (Traversal Using Relays around NAT):** Acts as a relay server when NAT or firewall restrictions make direct connections impossible. While TURN ensures connectivity, it adds latency and bandwidth costs [14].
- **ICE (Interactive Connectivity Establishment):** Manages the process of collecting potential connection candidates (public/private IPs) and testing them to find the most efficient path for communication, whether direct or through a relay [12].
- **DTLS (Datagram Transport Layer Security):** is a protocol designed to provide security for datagram-based applications, such as those using UDP. It ensures privacy, integrity, and authentication by encrypting and authenticating the communication, similar to TLS, but adapted for unreliable transport protocols [5].

To traverse NATs and firewalls, WebRTC uses the ICE framework, leveraging STUN servers to discover public IP addresses and ports. When direct connections are blocked by restrictive networks, TURN servers relay traffic to ensure connectivity. After signaling, a DTLS handshake is performed to exchange cryptographic keys securely. These keys are then used by SRTP to encrypt audio and video streams, ensuring confidentiality and integrity [12].

Once these steps are complete, a direct Peer-To-Peer connection is established, with audio/video transmitted over SRTP and other data over SCTP (Stream Control Transmission Protocol). This process relies on key components like signaling servers, STUN/TURN servers, and browser-supported WebRTC APIs to ensure seamless, secure communication across various network conditions [12].

From a security standpoint, an important part is the Session Description Protocol (SDP) handshake. A critical component of the WebRTC signaling process, enabling two peers to negotiate media parameters and establish a secure connection [16]. While SDP itself is a plain-text protocol and does not provide security features, it plays a pivotal role in enabling the secure setup of WebRTC sessions by facilitating the exchange of cryptographic keys, network information, and supported capabilities. A robust security architecture around SDP ensures the integrity, authenticity, and confidentiality of the WebRTC session.

Key Components of the SDP Handshake in WebRTC

- *Offer/Answer Model: The handshake begins with one peer generating an SDP "offer" that contains information about supported media codecs, network candidates (IP addresses and ports), and cryptographic keys for media encryption. The receiving peer responds with an SDP "answer" that finalizes the agreed-upon parameters [10].*
- *ICE Candidates: The SDP offer/answer exchanges Interactive Connectivity Establishment (ICE) candidates, which include potential connection endpoints. These candidates are necessary for traversing NATs and firewalls using STUN/TURN servers [10].*
- *DTLS Fingerprints: A critical security component in the SDP handshake is the inclusion of DTLS fingerprints—hashes of the public keys used during the subsequent DTLS handshake. These fingerprints ensure that peers can verify the authenticity of the DTLS certificates exchanged later, preventing man-in-the-middle (MITM) attacks during key negotiation [10].*
- *Key Exchange: SDP facilitates the secure exchange of cryptographic keys used by Secure Real-time Transport Protocol (SRTP) for encrypting media streams. WebRTC uses DTLS-SRTP to derive encryption keys during the DTLS handshake, ensuring that media streams remain protected from eavesdropping and tampering [10].*

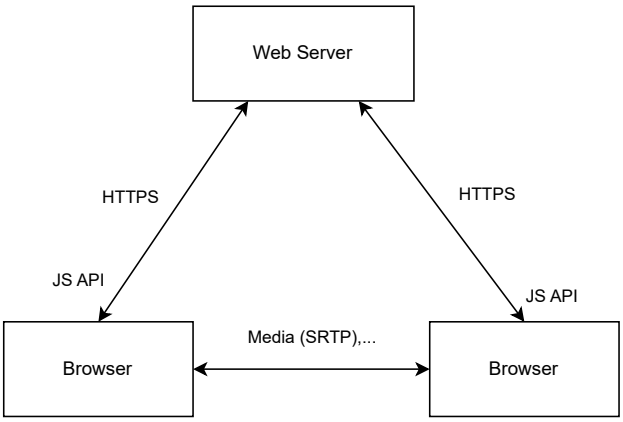

Fig. 1. A Simple WebRTC System [16]

Unlike most conventional real-time systems (e.g., SIP-based [RFC3261] soft phones), WebRTC communications are directly controlled by some Web server, via a JavaScript (JS) API as shown in Figure 1 1.

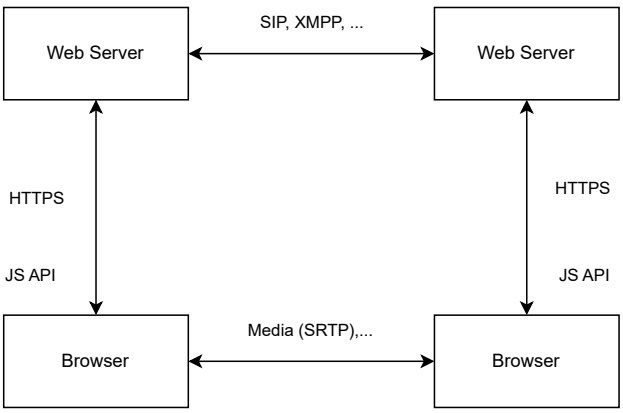

Fig. 2. A Multidomain WebRTC System [16]

A more complicated system might alow for inter-domain calling, as shown in Figure 2. The protocol to be used between the domains is not standardized by WebRTC, requires other form of security and may rely on other standardized Protocols such as the Session initiation Protocol (SIP), or Extensible Messaging and Presence Protocol (XMPP) [16] 2.

III. Authentication in WebRTC

A. Trust Model

The basic assumption of the standardized architecture is that network resources exist in a hierarchy of trust, rooted in the browser, which serves as the user's Trusted Computing Base (TCB) [16]. Any security property which the user wishes to have enforced must be ultimately guaranteed by the browser (or transitively by some property the browser verifies). Conversely, if the browser is compromised, then no security guarantees are possible. Note that there are cases (e.g., Internet kiosks) where the user can't really trust the browser that much. In these cases, the level of security provided is limited by how much they trust the browser. Optimally, we would not rely on trust in any entities other than the browser. However, this is unfortunately not possible if we wish to have a functional system. Other network elements fall into two categories: those which can be authenticated by the browser and thus can be granted permissions to access sensitive resources, and those which cannot be authenticated and thus are untrusted [16].

B. Authenticated Entities

There are two major classes of authenticated entities in the system:

- Calling services: Web sites whose origin we can verify (optimally via HTTPS, but in some cases because we are on a topologically restricted network, such as behind a firewall, and can infer authentication from firewall behavior) [16].
- Other users: WebRTC peers whose origin we can verify cryptographically (optimally via DTLS-SRTP) [16].

Merely being authenticated does not imply that entities are trusted. For example, verification that `https://www.example.org/` is owned by Dr. Evil does not necessarily mean that Dr. Evil can be trusted with access to the camera and microphone [9]. Authentication only provides the user with the opportunity to decide whether they wish to trust Dr. Evil [16]. In some cases, users may choose to grant temporary access to their camera and microphone, such as for a call with Dr. Evil to discuss a ransom payment. However, this trust should be limited to the duration of the call, with the user ensuring that access is not granted outside of this context [16]. The key point is that identification of other network elements is a prerequisite for determining the level of trust, as the policies applied depend on this identification [16].

C. Unauthenticated Entities

Apart from the entities previously mentioned, it is generally not possible to identify other network elements, which means they cannot be trusted. While interaction with these entities is not inherently impossible, it must be assumed that they could behave maliciously. Consequently, it is essential to design systems that remain secure even in the presence of such entities [16].

IV. Identification in WebRTC

"On the internet, no one knows you are a dog", the punchline of a comic by Peter Steiner back in 1993. More than 20 years later, assurance of user identity remains a challenge, and user impersonations are more frequent [3]. However, identification in Peer-To-Peer systems becomes a big problem, as a peer has to trust, that a connecting peer is who he pretends to be. In the standard HTTPs environment the ssl protocol is used to facilitate a trusted connection from client to server, in which the Server has been previously verified by a Certificate Authority, which is trusted by the client. In a general sense, this means that two peers require a third trusted party to facilitate identification in a secure manner [16]. This part could be a peer, or superpeer that has been previously verified by the opposing party and himself [4] 3.

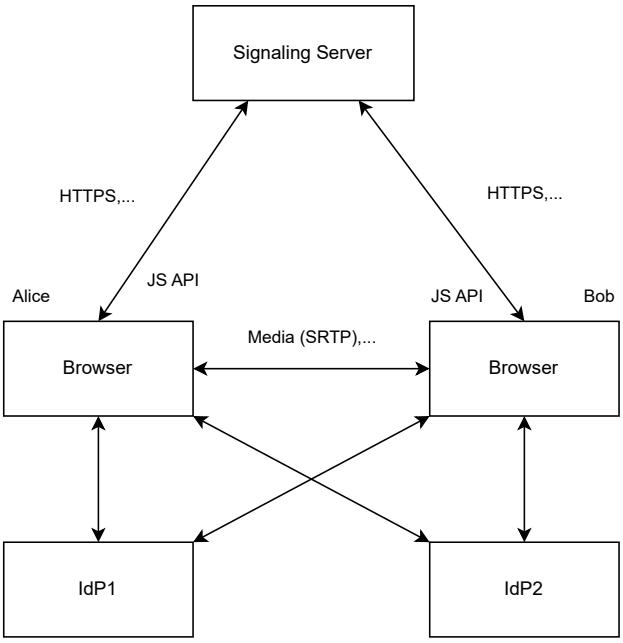

Fig. 3. A Call with IdP-Based Identity [16]

A. IdP

As a standardized way, WebRTC offers and answers (and hence the channels established by RTCPeerConnection objects) can be authenticated by using a web-based Identity Provider (IdP). The idea is that the entity sending an offer or answer acts as the Authenticating Party (AP) and obtains an identity assertion from the IdP which it attaches to the session description. The consumer of the session description (i.e., the RTCPeerConnection on which setRemoteDescription is called) acts as the Relying Party (RP) and verifies the assertion [11].

In order to verify assertions, the IdP domain name and protocol are taken from the domain and protocol fields of the identity assertion [16].

Communication with the IdP is established by the user agent loading the IdP JavaScript from the IdP. The URI for the IdP script is a well-known URI formed from the

"domain" and "protocol" fields, as specified in [16]. The IdP may generate an HTTP redirect to another "https" origin, the browser must treat a redirect to any other scheme as a fatal error. The user agent instantiates an isolated interpreted context, a JavaScript realm that operates in the origin of the loaded JavaScript [16]. Note that a redirect will change the origin of the loaded script The user agent provides an instance of RTCIdentityProviderRegistrar named rtcIdentityProvider in the global scope of the realm. This object is used by the IdP to interact with the user agent [16].

V. Authenticity and Data Integrity

WebRTC at it's core uses User Datagram Protocol a lightweight, connectionless communication protocol within the Internet Protocol (IP) suite [18]. It enables applications to send and receive discrete packets of data, known as datagrams, without requiring a prior connection. While UDP, does include a checksum to verify it's data integrity, it does not include any standard specification for ensuring packages have been received by the opposing party, relying on protocol extensions to facilitate save data transmission [18]. WebRTC enforces end-to-end encryption for all media and data streams by default, using standards like DTLS for signaling and data channels while relying on SRTP for media transmission [15].

A. DTLS

The DTLS protocol is a cryptographic protocol designed to provide secure communication over unreliable datagram-based transport layers such as UDP. It is defined in RFC 6347 and serves as an adaptation of the widely used Transport Layer Security (TLS) protocol, which secures communication over connection-oriented transport protocols like TCP. By incorporating mechanisms to handle packet loss, reordering, and duplication, DTLS ensures secure, reliable communication in real-time and low-latency scenarios [17]. DTLS provides a lightweight security layer that can be seamlessly integrated into datagram-based communication protocols. Its key features include:

- **Encryption for Confidentiality**: DTLS ensures the confidentiality of transmitted data by encrypting it using algorithms like AES or ChaCha20. Encryption ensures that even if an attacker intercepts the data, they cannot read its contents without the correct decryption key [17].
- **Message Authentication for Integrity**: DTLS uses message authentication codes (MACs), such as HMAC (Hash-based Message Authentication Code), to guarantee the integrity of each transmitted packet. This helps detect any tampering or unauthorized changes to the data during transmission [17].
- **Authentication**: DTLS provides mechanisms to authenticate both the client and the server, ensuring that both parties are who they claim to be. This is typically achieved through certificate-based mutual authentication using Public Key Infrastructure (PKI) [17].
- **Replay Protection**: To prevent attackers from replaying previously captured messages to disrupt communication or cause unauthorized actions, DTLS incorporates sequence numbers in its packets and requires message timestamps to detect and reject replayed packets [17].
- **Session Resumption**: DTLS supports session resumption, allowing parties to re-establish a secure connection without having to go through the full handshake process again. This reduces latency and computational overhead for subsequent sessions [17].
- **Denial of Service (DoS) Resistance**: DTLS includes mechanisms to mitigate the risk of Denial of Service attacks, such as through stateless protocols during the handshake, making it less vulnerable to attacks that involve overwhelming the server with connection requests [17].
- **Connectionless Security**: Unlike TLS, which is designed for connection-oriented protocols like TCP, DTLS operates over connectionless transport protocols such as UDP. This allows for low-latency communication, which is essential for real-time applications [17].

B. SRTP

The Secure Real-time Transport Protocol (SRTP) is a security extension of the Real-time Transport Protocol (RTP), designed to provide encryption, message authentication, and integrity for real-time multimedia communication. Introduced in 2004 and defined in RFC 3711, SRTP ensures the confidentiality and integrity of voice, video, and other real-time data transmitted over potentially insecure networks, such as the internet [2].

SRTP operates as a lightweight, scalable security mechanism that is tightly integrated with RTP. It adds minimal overhead, making it particularly suitable for bandwidth-constrained and latency-sensitive applications like Voice over IP (VoIP), video conferencing, and streaming. The key functionalities of SRTP are as follows:

- **Encryption for Confidentiality:** SRTP encrypts the payload of RTP packets to ensure that the media content remains confidential during transmission. By default, it uses the Advanced Encryption Standard (AES) in Counter Mode (AES-CM) for encryption. The encryption key and an initialization vector (IV) are derived from a master key provided during the session setup [2].
- **Message Authentication for Integrity:** To protect against tampering, SRTP employs message authentication codes (MACs) generated using HMAC-SHA1. This allows the receiving party to verify the integrity of each packet and detect modifications or forgeries [2].
- **Replay Protection:** SRTP includes a sequence number in its packets to prevent replay attacks. Packets received with duplicate sequence numbers are discarded, ensuring that attackers cannot retransmit old packets to disrupt communication [2].
- **Key Management and Derivation:** SRTP does not define its own key exchange mechanism but relies on external protocols like Datagram Transport Layer Security (DTLS) or Session Description Protocol Security

Descriptions (SDES) to establish and share master keys. These keys are then used to derive session keys for encryption and authentication [2].

- **Flexibility and Scalability:** SRTP supports multicast and unicast transmissions, making it highly adaptable to different communication scenarios. It also allows for the selective application of security features, such as enabling encryption only for specific streams [2].

While SRTP is robust, its reliance on external protocols for key management introduces potential vulnerabilities. For example, insecure signaling can lead to compromised key exchange. Additionally, misconfigured SRTP implementations may fail to enforce essential security features, such as replay protection or encryption [2].

Together, DTLS and SRTP form the backbone of WebRTC's security, enabling safe, real-time communication over potentially insecure networks.

VI. Risks and Concerns

A. WebRTC Leaks

Although WebRTC is standardized and widely adopted, its security is not without vulnerabilities, as WebRTC's inherent requirement for establishing a Peer-to-Peer connection necessitates knowledge of a peer's IP address. This can potentially lead to the leakage of a user's location, even when using VPNs [6]. The information at risk of being exposed includes the user's

- *Public IPv6 address this is the IPv6 address of the platform and is typically assigned by the ISP of the client [6].*
- *Public Temporary IPv6 address: this address is assigned by the network to which the client platform is attached [6].*
- *Unique local address (ULA) assigned by LAN: this IPv6 address is assigned by the network to which the client platform is attached, and is the approximate IPv6 counterpart of the Private IPv4 address assigned by LAN [6].*
- *Private IP address assigned by the VPN server: this private (IPv4 or IPv6, depending on the VPN configuration) address is assigned by the VPN server [6].*
- *Private IPv4 address assigned by LAN: this address is assigned by the network to which the client platform is attached [6].*

While not all leaks carry the same level of risk, for instance, a public IPv6 address leak is more dangerous than a unique local address leak, they remain significant security concerns that VPNs aim to mitigate. To address this, VPN providers have implemented WebRTC address leak detectors, which alert users if their private IP address is exposed [6]. Notable VPN providers offering this service include major players like ExpressVPN and Surfshark. Potential solutions to this issue include:

- Disabling WebRTC entirely inside the Browser or connecting client [6].
- Disabling IPv6 to reduce the amount of information leaked, if a leak occurs [6].
- Relying on Relay Server to transmit data to a central location before continuing on to the connected peer, which however, does need to be implemented by the Developers of the application using WebRTC, which defeats the purpose of not having a central server in Peer-To-Peer to begin with [6].

However, all these are mere workarounds and WebRTC without VPNs has to inherently leak IP addresses as they are required to connect to another peer [6].

B. Man-In-The-Middle during signaling

WebRTC's peer by default require End-To-End Encryption [16], however during the signaling process, it is susceptible to Man in the middle attacks [4].

In an MITM attack, an attacker intercepts and potentially manipulates the signaling messages exchanged between peers. This can lead to several security issues:

- **Key Substitution and Impersonation:** During signaling, peers exchange cryptographic fingerprints (e.g., DTLS-SRTP fingerprints) to verify each other's identity during the connection setup. An attacker could intercept and replace these fingerprints with their own, tricking each peer into establishing a connection with the attacker instead of the intended peer. This enables the attacker to decrypt or manipulate the media streams [7].
- **Session Hijacking:** An attacker could modify SDP parameters during the signaling exchange, such as altering ICE candidates to redirect traffic through malicious servers or inserting their own IP and port as a candidate. This would allow the attacker to take over the communication session [7].
- **Eavesdropping:** If signaling is not encrypted, an attacker can passively intercept SDP messages, which include media codecs, ICE candidates, and DTLS fingerprints. Although the actual WebRTC streams are encrypted, metadata leakage during signaling could provide attackers with useful information for further attacks [7].

In the eventuality that a malicious party succeeds in setting up a Man-In-The-Middle attack, there is typically not an easy solution to discover or fight against it. This is because the attack has no warning, and communication is allowed to proceed as normal. If one is not expecting such an attack, the attack will likely continue unnoticed [1].

However, by monitoring the media path regularly for no suspicious relays, we can take one small step towards mitigating against Man-In-The-Middle attacks. This should be coupled with encrypted signalling, as mentioned above [1].

C. Exploitation of vulnerable TURN Servers

While TURN servers are essential for ensuring connectivity, they can become a liability if misconfigured or improperly secured. Vulnerable TURN servers have been exploited for bandwidth abuse, data interception, and unauthorized use,

which can lead to significant security and operational risks [7].

The following examples illustrate real-world cases where misconfigured or improperly secured TURN servers were exploited, leading to significant security and operational risks.

- **Bandwidth Drain Attacks on Misconfigured TURN Servers:** In 2020, researchers highlighted cases where TURN servers operated without requiring authentication credentials. Attackers discovered these servers via automated scans and used them as relays for high-volume traffic, including video streaming and file transfers. This resulted in significant financial losses for the server operators due to excessive bandwidth usage. In some cases, smaller companies faced operational outages due to unexpected bandwidth exhaustion [8].
- **Abuse in Botnets:** Vulnerable TURN servers have been exploited in botnet operations to facilitate communication between infected devices. Attackers used TURN to bypass firewalls and NAT restrictions, creating resilient command-and-control (C&C) infrastructures. These servers also provided anonymity, making it difficult to trace botnet operations back to their origin [4].
- **Intercepting Sensitive Media Streams:** In cases where TURN servers did not enforce proper encryption, attackers exploited vulnerabilities to intercept and eavesdrop on real-time communications. Such attacks were particularly effective when TURN servers were hosted on public clouds with shared infrastructure, increasing the attack surface [1].
- **Abusing Slack's TURN servers to gain access to internal services:** Slack's TURN servers were found to be misconfigured, allowing researchers to exploit them to gain unauthorized access to internal services. These servers, essential for WebRTC communication, lacked sufficient access restrictions, enabling their use as relays to connect to internal systems that should have been isolated from external networks[8].

  The researchers demonstrated that the authentication mechanism for the TURN servers was weak, as the credentials were static and easily obtainable through client configurations. By using these credentials, they could authenticate and redirect relayed traffic to internal endpoints, effectively bypassing network restrictions. This misconfiguration allowed access to sensitive internal applications and services, exposing Slack's infrastructure to potential exploitation. Furthermore, the TURN servers were not properly isolated from the internal network, increasing the risk of unauthorized access to critical resources [8].

To mitigate the risks associated with misconfigured TURN servers and prevent potential exploitation, several best practices and security measures can be implemented.

- **Enforce Authentication**: Always require user authentication (e.g., long-term credentials or OAuth tokens) to access the server, and implement short expiration times for tokens to minimize misuse [7].
- **Restrict Access**: Limit access to the TURN server by configuring firewalls to allow traffic only from known IP ranges and implementing rate-limiting to prevent abuse [7].
- **Encrypt Traffic**: Ensure that all TURN traffic is encrypted using protocols such as DTLS or TLS, and avoid using plaintext TURN configurations [7].
- **Monitor Server Usage**: Regularly audit server logs to detect unusual patterns or unauthorized access attempts, and set alerts for excessive bandwidth usage [7].
- **Secure Deployment**: Deploy TURN servers in trusted environments and avoid shared or insecure cloud hosting setups [7].

VII. Conclusion

WebRTC and Peer-to-Peer communication have redefined real-time interactions in the modern web, offering unparalleled performance benefits by reducing reliance on centralized servers. However, the adoption of these technologies comes with significant security challenges, particularly in dynamic and distributed environments. This paper has explored the key mechanisms enabling WebRTC's functionality and the security risks posed by its architecture, including signaling vulnerabilities, IP leaks, and trust establishment.

While technologies such as DTLS-SRTP and Identity Providers (IdPs) provide robust solutions for encryption and authentication, there remain challenges, such as mitigating man-in-the-middle attacks during signaling and addressing the trade-offs introduced by TURN servers. Moving forward, securing WebRTC communications will require continued advancements in protocol design and increased awareness of the specific risks associated with P2P architectures. By addressing these challenges, WebRTC and similar technologies can fulfill their potential as the backbone of decentralized, real-time communication for the modern web.

References

[1] *A Study of WebRTC Security*. 2015. URL: https://datatracker.ietf.org/doc/html/rfc3264 (visited on 01/02/2025).

[2] M. Baughera et al. *The Secure Real-time Transport Protocol (SRTP)*. URL: https://www.rfc-editor.org/rfc/rfc3711.html (visited on 12/08/2024).

[3] Victoria Beltran, Emmanuel Bertin, and Noël Crespi. *User Identity for WebRTC Services: A Matter of Trust*.

[4] Dennis Boldt and Sebastian Ebers. "Security Mechanisms for Signaling in WebRTC-based Peer-to-Peer Networks". In: 2017.

[5] E. Rescorla D. McGrews. *Datagram Transport Layer Security (DTLS) Extension to Establish Keys for the Secure Real-time Transport Protocol (SRTP)*. 2010. URL: https://datatracker.ietf.org/doc/html/rfc5764 (visited on 01/03/2025).

[6] Nasser Mohammed Al-Fannah. "One Leak Will Sink A Ship: WebRTC IP Address Leaks". In.

[7] Ben Feher et al. "The Security of WebRTC". In: (Jan. 2016).

[8] Sandro Gauci. 2020. URL: https://www.enablesecurity.com/blog/slack-WebRTC-turn-compromise-and-bug-bounty/ (visited on 01/02/2025).

[9] M. Hansen et al. *Privacy Considerations for Internet Protocols*. 2013. URL: https://datatracker.ietf.org/doc/html/rfc6973 (visited on 01/02/2025).

[10] Columbia U. J. Rosenberg. *An Offer/Answer Model with the Session Description Protocol (SDP)*. 2002. URL: https://datatracker.ietf.org/doc/html/rfc3264 (visited on 01/02/2025).

[11] Cullen Jennings and Martin Thomson. *Identity for WebRTC 1.0*. URL: https://www.w3.org/TR/WebRTC-identity/ (visited on 12/08/2024).
[12] Cullen Jennings et al. *WebRTC: Real-Time Communication in Browsers*. URL: https://www.w3.org/TR/2024/REC-WebRTC-20241008/ (visited on 12/08/2024).
[13] Haitham Mahmoud and Raouf Abozariba. "A systematic review on WebRTC for potential applications and challenges beyond audio video streaming". In: *Springer Link* 11042-024-20448-9 (2024).
[14] M. Holdrege P. Srisuresh. *IP Network Address Translator (NAT) Terminology and Considerations*. 1999. (Visited on 01/02/2025).
[15] E. Rescorla. *RFC 8826 Security Considerations for WebRTC*. URL: https://www.rfc-editor.org/rfc/rfc8826.html (visited on 12/08/2024).
[16] E. Rescorla. *RFC 8827 WebRTC Security Architecture*. URL: https://www.rfc-editor.org/rfc/rfc8827.html (visited on 12/08/2024).
[17] E. Rescorla and N. Modadugu. *RFC 6347 Datagram Transport Layer Security Version 1.2*. URL: https://www.rfc-editor.org/rfc/rfc6347.html (visited on 12/08/2024).
[18] *RFC 768 User Datagram Protocol*. URL: https://www.rfc-editor.org/rfc/rfc768.html (visited on 12/08/2024).
[19] Felix Richter. *Zoom's Post-Pandemic Growth Slows to a Crawl*. URL: https://www.statista.com/chart/21906/zoom-revenue/.
[20] Berat Yilmaz, Ertugrul Barak, and Suat Ozdemir. "Improving WebRTC Security via Blockchain Based Smart Contracts". In: *2020 International Symposium on Networks, Computers and Communications (ISNCC)*. 2020, pp. 1–6. DOI: 10.1109/ISNCC49221.2020.9297333.