Snatch App Messaging Protocol

Table of Contents

SnatchApp Security Concepts (Feb 2020)

This document provides a general overview of the security mechanism implemented by Snatch App. All the core features of the Snatch App application are encrypted: private messages, media sharing, calls. This means that only the user's devices have access to them.

General encryption concepts

Private messages are encrypted by a client-server/server-client scheme.

Text messages and additional parameters like *Reply* are transformed into a simple text line and after that all the data is encrypted.

Attachments (images, documents, video files, etc.) are encrypted before they are uploaded to the *Cloud Storage (Amazon S3)*.

Encryption is implemented based on *Advanced Encryption Standard block cypher* (`aes-256-cbc`). *Advanced Encryption Standard (AES)*, also known by its original name Rijndael, is a specification for the encryption of electronic data established by the U.S. National Institute of Standards and Technology (NIST) in 2001. The algorithm described by AES is a symmetric-key algorithm, meaning the same key is used for both encrypting and decrypting the data.

Data transferred through the Snatch App backend APIs is encrypted by HTTPS protocol.

The *Messaging core* utilizes PostgreSQL DB which is encrypted with *RDS certificates*.

Server side internode communication is encrypted with *self-signed certificates*.

The algorithm for assigning a key

In order to keep user conversations confidential, the entire content of private chats is encrypted.

Snatch App security is based on the mechanism of symmetric cryptography where a pair of digital keys is used to encrypt and decrypt content. Each user owns a pair of

keys, one of them is defined as “Public” and is shared with everyone while the other one, the “Private” key, is kept secret.

Keys are received once (if the device has no key yet) requesting a backend endpoint, authorization is held via a unique `accessToken`. *Keys service* validates user access by requesting the backend with the user’s `accessToken` and `deviceID`. On the launch of the application, each Contact on the Snatch App contacts list is assigned a unique complex *Combined key* that includes the Private key of the current user and the Public key of this particular Contact.

- The generation of keys takes place via the `secp256r1` algorithm. Public and Private keys have different representations. The Public key is generated using a “`base64`”, “`compressed`” scheme; Private keys are created using a “`hex`” digit generation;
- The client app receives both keys in encrypted form;
- A *hash value* is generated from `SnatchID` through `sha256`;
- The *hash value* is converted to `base64`;
- Decryption takes place with the `aes-256-cbc` algorithm.

The Public key is recorded when a user's account is created and saved into the user’s profile. It is sent in each message.

The Private key is saved only on the device.

Both keys are transferred only in encrypted form. Both keys are stored on the server as a pair only in encrypted form.

General message encryption algorithm

1. A *Combined key* is generated based on the user’s Private key and the receiver’s Public key;
2. Data for the encryption is converted into `base64`;
3. The received message is encrypted with `aes-256-cbc`.

When a message is received, the application compares the `receiverPubKey` (Receiver Public Key) and the `pubKey` (Public Key) of the current user. If they match, *XMPP message* of successful delivery is sent and the message gets decrypted.

Group messages

Encryption does not take place for messages in Group chats.

Broadcast messages

In general, encryption for broadcast messages is the same as for basic Private messages.

- The broadcast message that is delivered to a user's private chat is encrypted;
- The version of the broadcast message sent to the broadcast sender will be recorded un-encrypted;

Calls encryption

Snatch App uses a *WebRTC* based solution to provide Calls and Video calls. *WebRTC* is an open-source, web-based application technology, which allows users to send real-time media without the need to install plugins.

QuickBlox follows the same security principles as the *WebRTC protocol.*

Encryption is a mandatory feature of *WebRTC*, and is enforced on all components, including signaling mechanisms. Consequently, all media streams sent over *WebRTC* are securely encrypted via standardised and well-known encryption protocols. Calls and Video calls are Media streams which are encrypted using *Secure Real-time Transport Protocol (SRTP). SRTP* uses *Advanced Encryption Standard (AES)* as the default cipher. This includes two cipher modes: *Segmented Integer Counter Mode* and *f8-mode*. *Segmented Integer Counter Mode* is standard, and is critical for running traffic over an unreliable network with a possible loss of packets. *f8-mode* is used for 3G mobile networks, and is a variation of the output feedback mode designed to be discoverable with an altered initialization function.

Web Real-Time Communication (abbreviated as WebRTC) is a recent trend in web application technology, which promises the ability to enable real-time communication in the browser without the need for plug-ins or other requirements. However, the open-source nature of the technology may have the potential to cause security-related concerns to potential adopters of the technology. This paper will discuss in detail the security of WebRTC, with the aim of demonstrating the comparative security of the technology.

WebRTC is an open-source web-based application technology, which allows users to send real-time media without the need for installing plugins. Using a suitable browser, it can enable a user to call another party simply by browsing to the relevant webpage. Some of the main use cases of this technology include the following:

- Real-time audio and/or video calls;
- Web conferencing;
- Direct data transfers.

Unlike most real-time systems (e.g. SIP), WebRTC communications are directly controlled by a Web server, via a JavaScript API.

The prospect of enabling embedded audio and visual communication in a browser without plugins is exciting. However, this naturally raises concerns over the security of such technology and whether it can be trusted to provide reliable communication for both the end users and any intermediary carriers or third parties.

Overview of WebRTC Architecture

WebRTC enables direct media-rich communication between two peers, using a peer-to-peer (P2P) topology. WebRTC resides within the user's browser and requires no additional software to operate. The actual communication between peers is prefaced by an exchange of metadata, termed "signalling". This process is used to initiate and advertise calls, and facilitates connection establishment between unfamiliar parties. As depicted in Figure 1, this process occurs through an intermediary server:

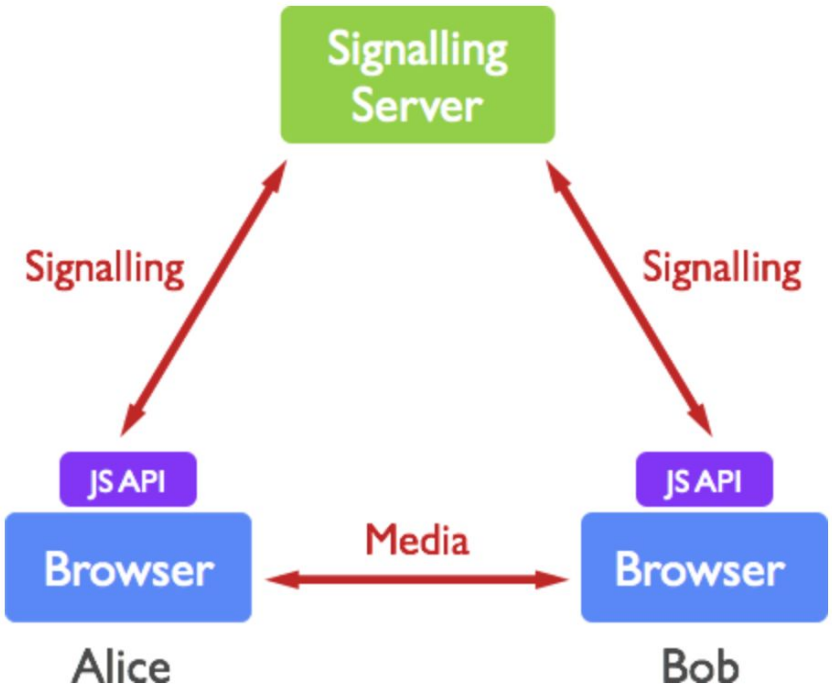

Figure 1. A simple WebRTC Call Topology

A signaling protocol is not specified within WebRTC, allowing developers to implement their own choice of protocol. This allows for a deeper degree of flexibility in adapting a WebRTC app for a specific use case or scenario.

How does WebRTC communication work?

WebRTC relies on three APIs, each of which performs a specific function in order to enable real-time communication within a web application. These APIs will be named and explained briefly. The implementation and technical details of each protocol and technology are outside the scope of this report, however the relevant documentation is readily available online.

getUserMedia

For many years it was necessary to rely on third-party browser plugins such as Flash or Silverlight to capture audio or video from a computer. However, the era of HTML 5 has ushered in direct hardware access to numerous devices, and provides JavaScript APIs which interface with a system's underlying hardware capabilities.

getUserMedia is one such API, enabling a browser to access a user's camera and microphone. Although utilised by WebRTC, this API is actually offered as part of HTML 5.

RTCPeerConnection

RTCPeerConnection is the first of two APIs which are offered specifically as part of the WebRTC specification. An RTCPeerConnection interface represents the actual WebRTC connection, and is relied upon to handle the efficient streaming of data between two peers.

When a caller wants to initiate a connection with a remote party, the browser starts by instantiating an RTCPeerConnection object. This includes a self-generated SDP description to exchange with their peer. The recipient in turn responds with its own SDP description. The SDP descriptions are used as part of the full ICE workflow for NAT traversal.

With the connection now established, RTCPeerConnection enables the sending of real-time audio and video data as a bitstream between browsers.

Ultimately, RTCPeerConnection API is responsible for managing the full life-cycle of each peer-to-peer connection and encapsulates all the connection setup, management, and state within a single easy-to-use interface.

RTCPeerConnection has two specific traits: Direct peer-to-Peer communication between two browsers; Use of UDP/IP. There is no guarantee of packet arrival (as in TCP/IP), but there is much reduced overhead as a result. (By allowing the loss of some data, we can focus upon offering real-time communication.)

RTCDataChannel

The RTCDataChannel is the second main API offered as part of WebRTC, and represents the main communication channel through which the exchange of arbitrary application data occurs between peers. In other words, it is used to transfer data directly from one peer to another.

Although a number of alternative options for communication channels exist (e.g. WebSocket, Server Sent Events), these alternatives were designed for communication with a server rather than a directly-connected peer. RTCDataChannel resembles the popular WebSocket, but instead takes a peer-to-peer format while offering customisable delivery properties of the underlying transport.

Media Encryption & Communication Security

There are a number of ways in which a real-time communication application may pose security risks. One particularly notable one is the interception of unencrypted media or data during transmission. This can occur between browser-browser or browser-server communication, with an eavesdropping third-party able to see all data sent. Encryption

however, renders it effectively impossible for an eavesdropper to determine the contents of communication streams. Only parties with access to the secret encryption key can decode the communication streams.

Encryption is a mandatory feature of WebRTC, and is enforced on all components, including signaling mechanisms. Consequently, all media streams sent over WebRTC are securely encrypted and enacted through standardised and well-known encryption protocols. The encryption protocol used depends on the channel type; data streams are encrypted using Datagram Transport Layer Security (DTLS) and media streams are encrypted using Secure Real-time Transport Protocol (SRTP).

DTLS: Datagram Transport Layer Security

WebRTC encrypts information (specifically data channels) using Datagram Transport Layer Security (DTLS). All data sent over RTCDataChannel is secured using DTLS.

DTLS is a standardised protocol which is built into all browsers that support WebRTC, and is the one protocol consistently used to encrypt information in web browsers, email, and VoIP platforms. The built-in nature of DTLS also means that no prior setup is required before use. As with other encryption protocols it is designed to prevent eavesdropping and information tampering. DTLS itself is modelled upon the stream-orientated TLS, a protocol which offers full encryption with asymmetric cryptography methods, data authentication, and message authentication. TLS is the de-facto standard for web encryption, utilised for the purposes of such protocols as HTTPS. TLS is designed for the reliable transport mechanism of TCP, but VoIP apps (and games, etc.) typically utilise unreliable datagram transports such as UDP.

As DTLS is a derivative of SSL, all data is known to be as secure as using any standard SSL-based connection. In fact, WebRTC data can be secured via any standard SSL-based connection on the web, allowing WebRTC to offer end-to-end encryption between peers with almost any server arrangement.

DTLS over TURN

The default option for all WebRTC communication is direct P2P communication between two browsers, aided with signalling servers during the setup phase. P2P encryption is relatively easy to envisage and setup, but in the case of failure, WebRTC setup falls back to communication via a TURN server (if available). During TURN communication the media can suffer a loss of quality and increased latency, but it allows an "if all else fails" scenario to permit the WebRTC application to work even under challenging circumstances. We must also consider encrypted communication under TURN's alternative communication structure.

It is known that regardless of communication method, the data sent is encrypted at the end points. A TURN server's purpose is simply the relay of WebRTC data between parties in a call, and it will only parse the UDP layer of a WebRTC packet for routing purposes. Servers will not decode the application data layer in order to route packets, and therefore we know that they do not (and cannot) touch the DTLS encryption. Consequently, the protections put in place through encryption are not compromised during WebRTC communication over TURN, and the server cannot understand or modify information that peers send to each other.

SRTP: Secure Real-time Transport Protocol

Basic RTP does not have any built-in security mechanisms, and thus places no protections of the confidentiality of transmitted data. External mechanisms are instead relied on to provide encryption. In fact, the use of unencrypted RTP is explicitly forbidden by the WebRTC specification.

WebRTC utilises SRTP for the encryption of media streams, rather than DTLS. This is because SRTP is a lighter-weight option than DTLS. The specification requires that any compliant WebRTC implementation support RTP/SAVPF (which is built on top of

RTP/SAVP). However, the actual SRTP key exchange is initially performed end-to-end with DTLS-SRTP, allowing for the detection of any MiTM attacks.

Verifying keys

Snatch App users additionally have the option to verify the keys of the other users with whom they are communicating so that they are able to confirm that an unauthorized third party has not initiated a *man-in-the-middle (MITM)* attack.
This can be done by scanning a QR code or by comparing an 80-digit number.

The QR code contains the encrypted `SnatchID`. When scanning, it will be decrypted so the application will be able to confirm if a conversation is taking place only with the person the user wanted to chat with.

The 80-digit number is a *Combined key* converted to `base64` shown byte-by-byte. The User and the Recipient can compare the keys to make sure there's no *MITM attack*.

SnatchID registration principle

A `SnatchID` is the main user identificator (4-15 symbols value). During registration users are allowed to skip the option of entering a phone number or email address. This gives additional security to Snatch App users as phone number as well as email address access can be easily stolen.
Phone numbers and email addresses can be used later to reset a password if it had been forgotten.

Conclusion

Messages between Snatch App users are protected with the `aes-256-cbc` encryption algorithm so that third parties and Snatch App itself cannot read them and so that the messages can only be decrypted by a Recipient. Every type of Snatch App private

message (including chats, images, videos, voice messages and files) and Snatch App calls are protected by the `aes-256-cbc` encryption algorithm.

Snatch App servers do not have access to the Private keys of users, and Snacth App users have the option to verify keys in order to ensure the integrity of their communication.