

Web Sockets

Kameswari Chebrolu

Outline

- What are WebSockets
- Why arose?
- How implemented?
- Implications on SOP and CORS
- Vulnerabilities
- Defenses
- Real Life Examples
- Summary and References

Web Sockets

- Standardized by Internet Engineering Task Force (IETF) in 2011
- A new protocol that operates at application layer (much like HTTP)
 - Offers a bidirectional, full-duplex communication channel over a single, long-lived TCP connection
 - Browsers can upgrade from HTTP to websockets
- Suitable for real-time applications such as chat applications, online gaming, and financial trading platforms!

Why arose?

- HTTP 1.0: one-to-one mapping between HTTP request and some resource (html page, image etc)
- HTTP 1.1 introduced AJAX, clients request and receive data **asynchronously** (without a page refresh)
 - AJAX still requires **client to initiate** the requests

- Scenario: Live cricket score/commentary
 - Client visits site and sees “latest” content, but content out of date soon after page loads
 - AJAX can be used for “client polling” i.e. regularly sends requests to server
 - May work in some, but not in all scenarios
 - When data does not change often, can generate high volume of requests with no useful data returned!
- Web Sockets can help in this scenario!

Web Socket Implementation

Step1:

- WebSocket connections are normally created using client-side JavaScript
 - a. `var ws = new WebSocket("wss://example.com/chat");`
- Results in HTTP request by client to server to establish the websocket connection

GET /chat HTTP/1.1

Host: example.com

Upgrade: websocket

Connection: Upgrade

Sec-WebSocket-Key: eahmIJKXbCLsvSIum89jqZN=

Sec-WebSocket-Version: 13

Cookie: session=MPc5LQosLb11g9BDKaMexoAG9tDaabL4

- Request includes an **Upgrade** header which indicates that the client wants to switch to the WebSocket protocol
- **Sec-WebSocket-Key** is a randomly generated value
 - Serves as a unique identifier for the client
 - Part of challenge-response mechanism

Step2: Server Response to Upgrade

- If server supports WebSockets and request is ok, responds with HTTP 101 status code (Switching Protocols)
- Also includes a Connection header indicating WebSocket protocol upgrade
- Sec-WebSocket-Accept header contains a value derived from the Sec-WebSocket-Key
 - Server concatenates the value of Sec-WebSocket-Key with a predefined string and then hashes it using SHA-1

HTTP/1.1 101 Switching Protocols

Upgrade: websocket

Connection: Upgrade

Sec-WebSocket-Accept:

q3MQLsoiQm1Q9kMIczhZGbK+yOo:

Step3:

- Once the WebSocket **handshake** is complete, connection is established
 - Both client and server can start sending messages back and forth

Step 4: Sending and Receiving Messages:

- Messages are typically sent as **binary** or **text** in **WebSocket frames**
- A simple message could be sent using JavaScript like `ws.send("Hello");`
- Common for JSON to be used to send structured data within WebSocket messages

- **WebSocket frame**: basic unit of communication
 - Encapsulates data being sent
 - Types of data:
 - **text/binary** data
 - **control frames** for managing connection and
 - **continuation frames** for large messages
 - Key components:
 - Header: Frame begins with a header that tells its **length**, **type**, and whether it is **masked** or not
 - Masking (Optional): payload data is **XORed** with a **masking key** to **obfuscate** contents
 - Payload: actual data being transmitted (binary or text)

Example

- A single-frame **unmasked** text message

0x81 0x05 0x48 0x65 0x6c 0x6c 0x6f

(contains "Hello")

- A single-frame masked text message

0x81 0x85 0x37 0xfa 0x21 0x3d 0x7f 0x9f 0x4d

0x51 0x58

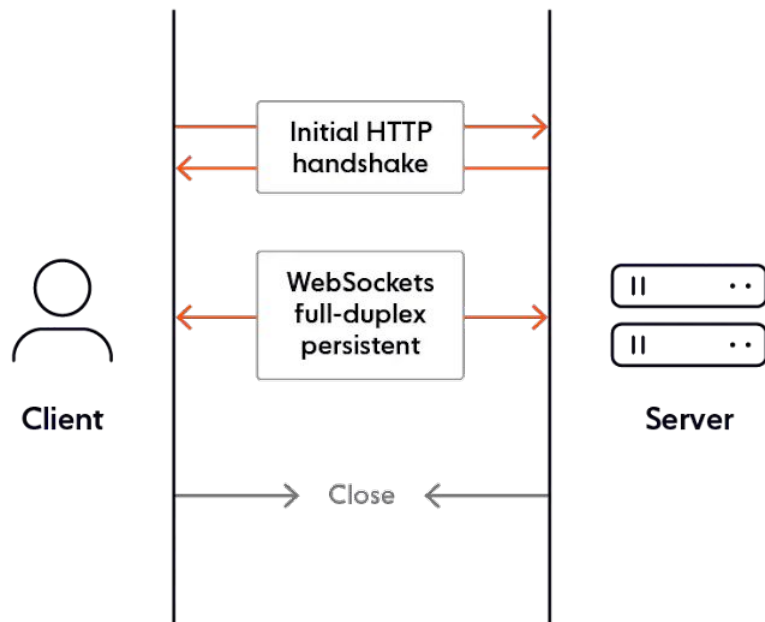
(contains "Hello")

Step5: Closing WebSocket Connection

- Either the client or server can initiate the closure by sending a special close frame
- Upon receiving this frame, the other party also responds with a close frame
- And connection is closed and **resources** freed up!

In the cricket scenario example,

- Once initial page is loaded, no further communication occurs until server has an update
- **Server pushes** the update over websocket connection
- And cycle repeats



Websockets and SOP+CORs

- SOP: Restricts how requesting origin can interact with resources retrieved from another origin
- CORs: Extends SOP to allow access to cross-origin resources
- With web sockets, client-server communication occurs over the WS protocol (i.e., ws:// or wss://)
→ Not same origin, but cross-origin

- However, cross-origin data access restrictions imposed by SOP or for that matter CORS does not apply to WebSockets
 - SOP only pertains to HTTP response data
 - HTTP update response has no data
 - SOP cannot instruct the browser to restrict access to data transmitted via WebSockets.
 - WS do not support custom headers within WS handshake → CORS policy cannot be inserted into an HTTP Upgrade response

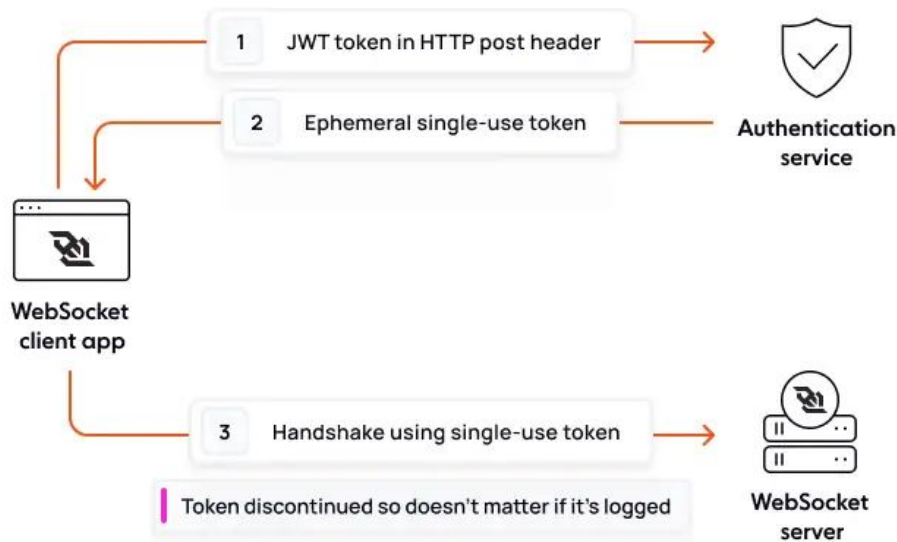
Authentication in Web Sockets

- Specification does not prescribe how to authenticate a WebSocket connection
 - Suggests authenticate the HTTP handshake before establishing the connection
- Often a token is used
 - A JSON Web Token (JWT) or
 - A session token obtained from a previous login process

Option-1: Access token in query parameter

- Pass credentials (access token) via URL
 - `wss://example.com?token=your_token_here`
- Very easy to implement
- If **TLS** is used, query strings are **encrypted** in transit
- WebSocket URLs aren't really exposed to user
 - Users can't bookmark or copy-and-paste them → minimizes risk of accidental sharing
- However, URL with query parameters likely **logged**
 - If attacker can access logs, attacker can access data

Option-2: One-time token in query parameter



⚠ Tokens sent in the query string will very likely be logged in plain text. Make sure to only send single-use ephemeral tokens with the WebSocket handshake.

- By the time the token is logged on server, it will either have already been used or **expired**
- Better in security than option-1 but need to implement a **custom and stateful authentication**

Option-3: Authenticate over WebSocket Connection

- Send credentials in the first message post-connection over WebSocket
- Server must validate the token before allowing the client to do anything else!
 - If token invalid, server should terminate connection
- Need to define own custom authentication (session handling, timeouts etc)

Option-4: Using Cookie

- WebSocket **handshake** supports **cookies** → can authenticate the request
- However this will not work if **WebSocket server** is hosted on a **different domain** than web app
 - Browser will not send cookies due to SOP

GET /chat HTTP/1.1

Host: example.com

Upgrade: websocket

Connection: Upgrade

Sec-WebSocket-Key: eahmIJKXbCLsvSIum89jqZN=

Sec-WebSocket-Version: 13

Cookie: session=MPc5LQosLb11g9BDKaMexoAG9tDaabL4

Vulnerabilities

- Any web security vulnerability that normally occurs can arise in WebSockets too!
 - User-supplied input can be processed in unsafe ways → SQL injection, XSS etc

XSS

- Consider a chat application based on WebSockets
- User types a message, following sent to Server
 - {"message":"Hello!"}
 - Rendered in user's browser as <td>Hello!</td>
- Assuming no defenses in place, attacker can send {"message":""}
 - Alert popped!

Cross Site WebSocket Hijacking

- Also called cross-origin WebSocket hijacking
- Exploits cross-site request forgery (CSRF) vulnerability on a WebSocket handshake
 - Requires WebSocket handshake to rely solely on HTTP cookies
 - Session handling does not contain any CSRF tokens or other unpredictable values!

- User visits malicious website (A)
- A opens a websocket connection to victim website (B)
 - Browser will send **user cookies** to B and authenticates user (without knowledge of user)
- A can then send arbitrary messages to B and also read responses
 - Why? **SOP does not apply to websockets**
- Unlike regular CSRF, attacker gains **two-way** interaction with the compromised website (B)

Defense

- SOP and CORS don't apply to Websockets → WebSocket server should verify origin of HTTP Upgrade request
 - Reject connections from unexpected or unauthorized origins
 - Origin check should be done on top of authentication and authorization checks (Not a replacement)
- Use the wss:// protocol (WebSockets over TLS) as opposed to ws://

- Treat data received via the WebSocket as **untrusted** in **both directions**
 - Handle data safely to prevent input-based vulnerabilities such as SQL injection and XSS
- Use **CSRF tokens** as part of user's session to avoid attacks like **Cross Site Websocket Hijacking**
 - Attackers cannot forge requests if they cannot guess the token

Real Life Examples

Cross Site WebSocket Hijacking:

<https://hackerone.com/reports/535436>

References

- <https://portswigger.net/web-security/websockets>
- <https://ably.com/topic/websocket-security> and <https://ably.com/blog/websocket-authentication>
- WebSocket Protocol:
<https://datatracker.ietf.org/doc/html/rfc6455>

Summary

- Offer a bi-directional, full duplex communication channel over TCP
- Implemented via HTTP Upgrade, websocket frames
 - SOPS or CORS do not apply to them
- Authentication is challenging, but options exist
- Vulnerable to usual web vulnerabilities
 - Cross Site Websocket Hijacking is a new attack!
- Defense: check origin, use of wss, CSRF tokens