Network Security - Assignment2

Efstratios Demertzoglou — TH20580
Vasilis Dalas — TH20093

November 2025

1 Task 1 – WebSocket Signaling Proxy

1.1 Task 1.1: Proxy Implementation and Deliverables

Goal

The goal of Task 1.1 was to implement a WebSocket signaling proxy between the browser-based WebRTC clients and the original signaling server. Instead of connecting directly to the signaling server at `ws://localhost:8080`, the WebRTC clients connect to the proxy at `ws://localhost:8081`. The proxy transparently forwards all WebSocket messages in both directions and logs the full signaling exchange (SDP offers/answers and ICE candidates).

Architecture Overview

The resulting architecture is:

- **WebRTC clients**: browser tabs for `client-a` and `client-b`, configured to use `ws://localhost:8081` as their signaling endpoint.
- **WebSocket proxy** (`ws-proxy`): a local **Node.js** process implemented in `proxy.js`, listening on `ws://localhost:8081`. For each incoming client connection, it opens a corresponding connection to the real signaling server and forwards messages:
  - Client → Proxy → Server
  - Server → Proxy → Client

  All traffic is logged in a human-readable format into `proxy.log`.
- **Real signaling server**: unchanged WebRTC signaling server listening on `ws://localhost:8080`.

Conceptually, the proxy acts as a transparent man-in-the-middle at the signaling layer: it does *not* parse or modify messages, but it is able to inspect and log them.

Running the Proxy

The steps to run the setup for Task 1.1 were:

1. Start the original signaling server on port `8080`.
2. Start the WebSocket proxy (in directory `NS_A2/ws-proxy`, for example):

```
$ cd NS_A2/ws-proxy
$ npm install          # first time only, to install the 'ws' dependency
$ node proxy.js        # start the Node.js WebSocket proxy
```

3. Configure both browser clients to use `ws://localhost:8081` as their signaling URL.
4. Open the two browser tabs (for `client-a` and `client-b`), register them, and initiate a call between them.

After these steps, the file `proxy.log` contains the complete signaling trace for the session.

Evidence of Correct Operation

The generated log file `proxy.log` shows that:

- The proxy starts and announces the endpoints:

```
[*] WebSocket proxy listening on ws://localhost:8081
[*] Forwarding to ws://localhost:8080
```

- Two distinct WebSocket client connections are accepted and each is paired with a server connection:

```
[+] New client connected from ::1
[*] Connected to real signaling server
```

- Both clients successfully register at the signaling server via the proxy:

```
[C → S] Raw: {"type":"register","clientId":"client-b", ...}
[S → C] Raw: {"type":"registered","clientId":"client-b"}

[C → S] Raw: {"type":"register","clientId":"client-a", ...}
[S → C] Raw: {"type":"registered","clientId":"client-a"}
```

- An SDP `offer` from `client-a` to `client-b` is logged in full, first as raw JSON and then as pretty-printed JSON. The proxy also logs the copy forwarded from server to the other client:

```
[C → S] Raw: {"to":"client-b","type":"offer","sdp":"v=0\r\no=- 5957032854 ..."}
[S → C] Raw: {"to":"client-b","type":"offer","sdp":"v=0\r\no=- 5957032854 ...",
"from":"client-a"}
```

- Multiple ICE candidates from `client-a` are sent to the server and forwarded to `client-b`:

```
[C → S] Raw: {"to":"client-b","type":"ice","candidate":{
  "candidate":"candidate:2888037346 1 udp 2122260223 172.20.80.1 58796 typ host ...",
  "sdpMid":"0","sdpMLineIndex":0,"usernameFragment":"AdUF"}}

[S → C] Raw: {"to":"client-b","type":"ice","candidate":{
  "candidate":"candidate:2888037346 1 udp 2122260223 172.20.80.1 58796 typ host ..."},
  "from":"client-a"}
```

- Symmetrically, `client-b` sends its own `answer` and ICE candidates to `client-a`, again all visible in the log:

```
[C → S] Raw: {"to":"client-a","type":"answer","sdp":"v=0\r\no=- 4192634683 ..."}
[S → C] Raw: {"to":"client-a","type":"answer","sdp":"v=0\r\no=- 4192634683 ...",
"from":"client-b"}
```

- At the end of the session, the proxy logs the clean teardown of both client and server connections:

```
[*] Client connection closed
[*] Server connection closed
[*] Client connection closed
[*] Server connection closed
```

These entries constitute the deliverables for Task 1.1: a working *ws-proxy* implementation in Node.js and a complete signaling trace in `proxy.log` that shows the registration, offer/answer exchange, ICE candidate exchange, and connection teardown.

1.2 Task 1.2: Reverse Engineer the Protocol

Task Description and Questions

Task 1.2: Reverse Engineer the Protocol (15 points)

Analyze the captured traffic and document the registration protocol.

Questions to answer:

1. What is the WebSocket message format (JSON, binary, etc.)?
2. What fields are in a registration message?

3. What value identifies a client uniquely?
4. How does the server respond to registration?
5. Is there any authentication mechanism (tokens, passwords)?
6. What prevents an attacker from registering with another client's ID?

Deliverables:

1. Complete protocol documentation
2. Example registration messages (request and response)
3. Sequence diagram of registration flow
4. Security analysis

Answers to Questions 1–6

1. WebSocket message format. From the `proxy.log` file we see that all application-level messages are logged as textual JSON. For example:

```
[C → S] Raw: {"type":"register","clientId":"client-b",
              "meta":{"displayName":"client-b"}}
[C → S] JSON: { "type": "register", "clientId": "client-b", ... }
```

There is no evidence of binary frames in the captured trace. *Answer:* The signaling protocol uses WebSocket *text* frames containing JSON objects.

2. Fields in a registration message. A registration request from the client to the server has the form:

```
{
  "type": "register",
  "clientId": "<client identifier>",
  "meta": {
    "displayName": "<human-readable name>"
  }
}
```

Based on the trace:

- `"type"`: string, fixed value `"register"` for registration.
- `"clientId"`: string identifying the client (e.g. `"client-a"`, `"client-b"`).
- `"meta"`: object containing additional metadata.
- `"meta.displayName"`: human-readable label for the client, here equal to the clientId.

Answer: The registration message contains at least the fields `type`, `clientId`, and `meta.displayName`.

3. Unique client identifier. Later messages use a `"to"` field to address a specific peer and a `"from"` field set by the server:

```
{"to":"client-b","type":"offer", ...}
{"to":"client-b","type":"offer", ...,"from":"client-a"}
```

This shows that:

- The application uses `clientId` values (e.g. `"client-a"`, `"client-b"`) to route messages.
- These IDs remain stable throughout the session.

Answer: The value that uniquely identifies a client in the signaling protocol is `clientId`.

4. Server response to registration. Immediately after each registration request, the server replies:

```
[C → S] Raw: {"type":"register","clientId":"client-b", ...}
[S → C] Raw: {"type":"registered","clientId":"client-b"}

[C → S] Raw: {"type":"register","clientId":"client-a", ...}
[S → C] Raw: {"type":"registered","clientId":"client-a"}
```

Answer: The server responds with a JSON message of the form:

```
{
  "type": "registered",
  "clientId": "<same clientId as in the request>"
}
```

This acts as an acknowledgment that the client has been successfully registered.

5. Authentication mechanism. In the captured registration messages there are:

- No password fields.
- No tokens, signatures, or nonces.
- No indication of an external identity provider.

All visible registration messages consist only of `type`, `clientId`, and `meta.displayName`.
Answer: In the observed traffic there is *no explicit authentication mechanism.*
Any client can send a `"register"` message with an arbitrary `clientId`.

6. Protection against ID spoofing. From the perspective of the on-the-wire protocol:

- The client simply chooses its `clientId`.
- The server echoes back the same `clientId` without any cryptographic binding.
- There is no proof of possession (e.g. password, key, token) associated with a given `clientId`.

An attacker who can open a WebSocket connection to the signaling server could register:

```
{"type":"register","clientId":"client-a", ...}
```

and potentially impersonate the legitimate client, unless the server enforces additional checks that are not visible in the trace (for example, rejecting duplicate `clientId`s). *Answer:* Based on the captured traffic, there is nothing in the protocol that prevents an attacker from registering with another client's ID.

Protocol Documentation (Registration Phase)

This subsection summarizes the observed registration protocol independently of the specific trace.

Transport.

- Transport: WebSocket over TCP, here on `ws://localhost:8080` (with the proxy on `ws://localhost:8081`).
- Frame payload: UTF-8 JSON objects (no binary frames observed).

Message types. In the registration phase we observe two message types:

- **Client → Server:** `"type": "register"`
- **Server → Client:** `"type": "registered"`

Registration request (client → server).

`type` String, MUST be `"register"`.

`clientId` String, chosen by the client; intended as unique identifier for this logical endpoint.

`meta` Object containing additional metadata.

`meta.displayName` String, human-readable display name for UI purposes.

Registration response (server → client).

`type` String, MUST be `"registered"`.

`clientId` String, the same `clientId` accepted by the server for this connection.

State machine (informal). For each WebSocket connection:

1. **Initial state:** WebSocket connection established, client is *unregistered*.
2. **Register:** Client sends a `"register"` message with chosen `clientId`.
3. **Ack:** Server responds with `"registered"` for that `clientId`.
4. **Registered state:** Connection is now associated with that `clientId`. Subsequent signaling messages (offers, answers, ICE) may be routed using `"to"` and `"from"` fields containing this ID.

Example Registration Messages

Registration request (from client-b).

```
{
  "type": "register",
  "clientId": "client-b",
  "meta": {
    "displayName": "client-b"
  }
}
```

Registration response (to client-b).

```
{
  "type": "registered",
  "clientId": "client-b"
}
```

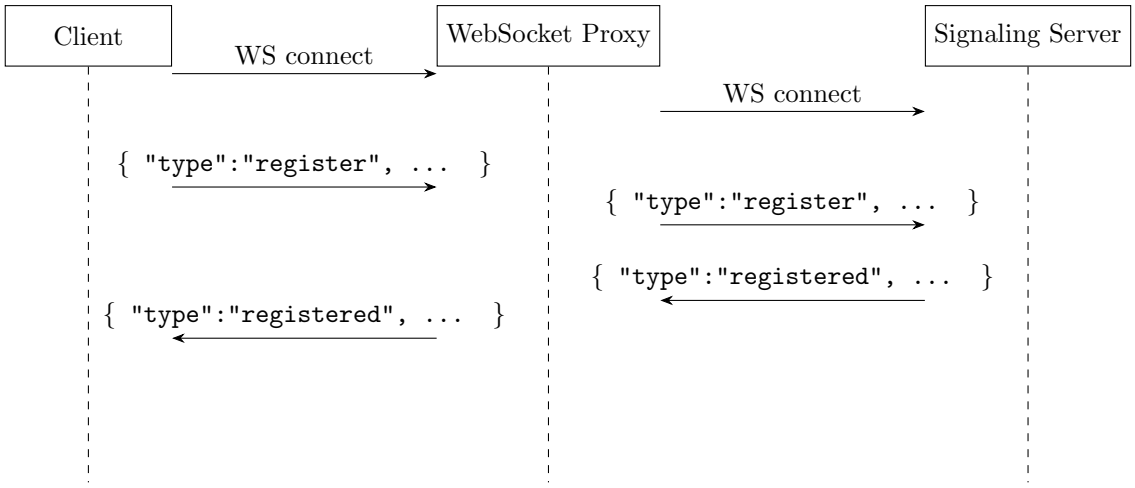
Registration request (from client-a).

```
{
  "type": "register",
  "clientId": "client-a",
  "meta": {
    "displayName": "client-a"
  }
}
```

Registration response (to client-a).

```
{
  "type": "registered",
  "clientId": "client-a"
}
```

Sequence Diagram of Registration Flow



This sequence diagram shows:

1. The client opens a WebSocket connection to the proxy, which in turn connects to the real signaling server.
2. The client sends a `"register"` message with its chosen `clientId`; the proxy forwards it unchanged to the server.
3. The server responds with `"registered"`; the proxy forwards this response back to the client.

Security Analysis

We now analyze the registration protocol from a security perspective, based solely on the observed traffic.

Lack of authentication. The registration request contains only:

- `type = "register"`
- `clientId = "<arbitrary string>"`
- `meta.displayName = "<arbitrary string>"`

There is no password or token that proves the client is authorized to use a particular `clientId`. *Implication:* Any endpoint that can reach the signaling server can attempt to register as any ID.

Client ID spoofing. Because the server simply echoes back the provided `clientId` in a `"registered"` response, an attacker could:

1. Connect to the signaling server.
2. Send `"register"` with `clientId = "client-a"`.
3. Receive `"registered"` and start receiving any signaling messages addressed to `"client-a"`.

Unless the server performs extra checks (e.g. disallowing duplicates, IP-based restrictions, or out-of-band authentication), impersonation is straightforward.

Transport security. In the lab environment, the signaling uses `ws://localhost`, i.e. plaintext WebSocket over TCP:

- On a local machine this is acceptable for experimentation.
- In a real deployment, a downgrade from `wss://` (WebSocket over TLS) would allow network attackers to:
  - Eavesdrop on registration messages and learn `clientId`s.
  - Inject or modify `register` and `registered` messages.

Summary of security properties.

- The protocol is *simple and functional* for controlled environments (e.g. local lab exercise).
- It provides *no strong identity guarantees*: any client can claim any `clientId`.
- There is no cryptographic protection at the signaling layer; security relies entirely on the transport (and here even that is non-encrypted).
- For a production system, we would recommend:
  - Using `wss://` with TLS.
  - Binding `clientId`s to authenticated user accounts or long-term keys.
  - Having the server enforce uniqueness and authorization for each `clientId`.

This completes the deliverables for Task 1.2: the questions are explicitly answered, the registration protocol is documented, concrete message examples are provided, a sequence diagram is included, and the main security issues are discussed.

2 Task 2 – Attacks on Signaling and Media

2.1 Task 2.3: WebRTC Media Interception Attacker

Goal

The goal of Task 2.3 was to extend the signaling–layer attacks from Tasks 2.1 and 2.2 with a full *media interception* attack against the WebRTC demo application. Instead of only hijacking the registration or observing signaling messages, we implemented an active attacker that:

- Connects directly to the WebSocket signaling server.
- Registers using the *victim*'s `clientId` (e.g. `client-a` or `client-b`).
- Intercepts the WebRTC `offer` that the other party sends to the victim.
- Creates its own `RTCPeerConnection` using the `aiortc` library.
- Generates and sends a valid WebRTC `answer` back to the caller via the signaling server.
- Exchanges ICE candidates and establishes a direct media connection between the attacker and the caller.
- Records the intercepted audio/video stream into a local media file for later inspection.

This demonstrates that weak signaling security can lead not only to impersonation (Tasks 2.1–2.2), but also to a complete compromise of the media channel.

High-Level Architecture

The architecture for Task 2.3 consists of the following components:

- **Signaling server**: the original WebRTC signaling server provided by the assignment, listening on `ws://localhost:8080`.
- **Browser clients**: the two demo pages (Client A and Client B) running in the browser, each configured to use the signaling server at `ws://localhost:8080`.
- **Media interception attacker** (`interceptor_webrtc.py`):
  - A Python script that connects to the same signaling server via WebSocket.
  - Registers as the victim (`client-a` or `client-b`).
  - Uses `aiortc` to implement the WebRTC peer connection on the attacker side.

– Uses `aiortc.contrib.media.MediaRecorder` to save received media into a file (e.g. `recordings/intercepted_media.webm`).

Conceptually, the attacker pretends to be the victim at the signaling layer. As a result, all WebRTC signaling related to that `clientId` (offer, answer, ICE candidates) is delivered to the attacker instead of the real browser tab. The attacker then completes the WebRTC handshake directly with the caller and receives the media stream.

Implementation Overview

The implementation of the attacker is contained in the Python script `interceptor_webrtc.py`. The main steps are:

1. **Configuration and CLI**.
   At startup, the script parses command-line arguments into an `AttackConfig` object with:
   - `server_url`: WebSocket URL of the signaling server (default `ws://localhost:8080`).
   - `victim_id`: `clientId` to impersonate (default `client-a`).
   - `display_name`: optional human-readable name for the registration metadata.
   - `output_file`: path for the recorded media file (default `recordings/intercepted_media.webm`).
2. **Registration hijacking**.
   The coroutine `run_attack` opens a WebSocket connection to the signaling server and sends:

   ```
   {
     "type": "register",
     "clientId": "<victim-id>",
     "meta": { "displayName": "<display-name>" }
   }
   ```

   It then waits for:

   ```
   {
     "type": "registered",
     "clientId": "<victim-id>"
   }
   ```

   confirming that the server now associates this WebSocket connection with the victim's ID.
3. **Intercepting the offer**.
   After registration, the script listens for signaling messages until it receives the first `"offer"` addressed to the victim:

```
{
  "to": "<victim-id>",
  "from": "<caller-id>",
  "type": "offer",
  "sdp": "v=0..."
}
```

This message would normally be delivered to the real victim, but is now intercepted by the attacker.

4. **Creating the RTCPeerConnection and answer**.
   The function `handle_offer_and_media` handles the intercepted offer:

   - Creates an `RTCPeerConnection` with a simple configuration (including a public STUN server).
   - Prepares a `MediaRecorder` targeting the chosen output file. For `.webm` files, the script explicitly passes `format="webm"` to improve compatibility.
   - Installs event handlers for:
     - `"track"`: triggered when audio or video tracks arrive.
     - `"icecandidate"`: triggered when the local ICE agent discovers new candidates.
     - `"iceconnectionstatechange"`: for logging the ICE state.
   - Applies the intercepted SDP as the **remote description**:

     ```
     pc.setRemoteDescription(RTCSessionDescription(sdp, "offer"))
     ```

   - Generates an SDP **answer**, sets it as the local description, and sends it back to the caller via the signaling server:

     ```
     {
       "to": "<caller-id>",
       "type": "answer",
       "sdp": "<attacker's SDP answer>"
     }
     ```

5. **ICE candidate handling**.
   When messages of type `"ice"` are received from the signaling server, the attacker extracts the embedded candidate and calls:

   ```
   await pc.addIceCandidate(RTCIceCandidate(...))
   ```

   Conversely, when the attacker's `RTCPeerConnection` discovers local ICE candidates, the script serializes them and sends them back to the caller using the same JSON structure:

```
{
  "to": "<caller-id>",
  "type": "ice",
  "candidate": {
    "candidate": "candidate:...",
    "sdpMid": "...",
    "sdpMLineIndex": 0
  }
}
```

6. **Media recording**.
   Each time an incoming media track is received in the `"track"` handler:

   - The track (audio or video) is attached to the `MediaRecorder`.
   - The recorder is started once, with a small delay (e.g. 0.5 s) to give both audio and video tracks time to arrive before finalizing the output format.

   When the media tracks end or the connection closes, the recorder is stopped and the output file is finalized on disk.

Execution Scenario and Evidence

A typical experiment for Task 2.3 was performed as follows:

1. Start the signaling server on `ws://localhost:8080`.
2. Open the two WebRTC demo clients in the browser (Client A and Client B) and confirm that a normal call works.
3. Stop any existing call so that a new one can be used for interception.
4. In a terminal, launch the media interception attacker, impersonating `client-b` and writing to a WebM file:

   ```
   python interceptor_webrtc.py \
       --server-url ws://localhost:8080 \
       --victim-id client-b \
       --output recordings/client_a_intercept.webm
   ```

5. In the Client A tab, initiate a call towards `client-b`.
6. Observe in the attacker's terminal:

   - Successful registration as `client-b`.
   - Interception of the `"offer"` from `client-a`.
   - Creation of the `RTCPeerConnection` and sending of the `"answer"`.
   - ICE candidates being exchanged.

- Arrival of audio and video tracks and the start of the `MediaRecorder`.

7. After some seconds, stop the script with `Ctrl+C` to close the connection cleanly.
8. Open the resulting file (e.g. `recordings/client_a_intercept.webm`) with a media player and confirm that both audio and video from the call have been captured.

This experiment provides concrete evidence that the attacker is able to hijack signaling for `client-b`, complete the WebRTC handshake, and record the media stream without any involvement from the legitimate browser tab.

Security Discussion

From a security point of view, Task 2.3 illustrates the impact of combining:

- **Unauthenticated signaling** (as identified in Task 1.2), where any endpoint can register arbitrary `clientId`s.
- **Standard WebRTC APIs** (here implemented with `aiortc`) that faithfully follow the offer/answer and ICE procedures.

Because there is no authentication or authorization on the `register` messages, an attacker can:

1. Register as a legitimate client, e.g. `client-b`.
2. Receive offers that honest peers send to that identity.
3. Complete the WebRTC negotiation and establish a media channel directly with the honest peer.
4. Capture the resulting audio/video without the knowledge of the real `client-b`.

In a real-world scenario this would constitute a severe breach of confidentiality: all communication that the caller believes is going to the intended recipient is actually delivered to the attacker. The exercise therefore highlights the necessity of:

- Strong authentication of signaling endpoints (binding `clientId`s to user accounts, credentials, or long-term keys).
- Use of `wss://` (WebSocket over TLS) and robust access control on the signaling server.
- Careful validation of who is allowed to receive offers and answers for each logical user identity.

Overall, Task 2.3 demonstrates that weaknesses at the signaling layer directly translate into full media interception capabilities.

3 Task 3 – Security Analysis and Mitigations

3.1 Task 3.1: Vulnerability Analysis

Overview

In Tasks 1 and 2 we implemented and exploited a weakness in the WebRTC signaling server: any WebSocket client can register an arbitrary `clientId` without authentication, and the server then routes all subsequent offers, answers and ICE candidates based solely on this string identifier. This design flaw enables an attacker to impersonate another client and, as demonstrated in Task 2.3, to establish a full WebRTC media session with an honest peer and record the audio/video stream.

This subsection analyzes *why* the attack is possible, what information the attacker needs, what the real-world impact would be, and how such attacks could be detected in practice.

Answers to Guiding Questions

1. What security mechanisms are missing from the signaling server? From the captured traffic and our experiments we observe that the signaling server provides only a minimal registration mechanism:

- Clients send a JSON message of the form:

  ```
  { "type": "register", "clientId": "<string>", "meta": { "displayName": "..." } }
  ```

- The server replies with:

  ```
  { "type": "registered", "clientId": "<same string>" }
  ```

  There are several missing security mechanisms:

- **No authentication:** There is no password, token, certificate or any other proof that the client is *entitled* to use a given `clientId`. Identity is self-declared.

- **No authorization or access control:** The server never checks whether the connecting peer is allowed to act as `"client-a"` or `"client-b"`. Any connecting endpoint is treated as trusted once registered.

- **No integrity protection at the signaling layer:** Messages are plain JSON over `ws://`. In a non-local deployment, this would allow man-in-the-middle modification of offers, answers and ICE candidates.

- **No registration policy:** There is no rule such as "only one active registration per `clientId`", no session expiry, and no binding of `clientId` to a user account or device.

In summary, the signaling server treats the `clientId` as *trusted input*, even though it is completely under client control.

2. Why does the server accept duplicate registration? Empirically, when our attacker connects and sends:

```
{ "type": "register", "clientId": "client-b", ... }
```

the server responds with:

```
{ "type": "registered", "clientId": "client-b" }
```

even if a legitimate browser tab has already registered as `"client-b"` earlier. This indicates that:

- The server maintains a simple mapping from a WebSocket connection to a `clientId`, but
- It does *not* globally enforce uniqueness of `clientId` values.

Possible internal behaviors (all of which are insecure in this context) include:

- **Overwrite:** The last registration silently overwrites any previous association for the same `clientId`. New signaling messages for `"client-b"` are now routed to the attacker.
- **Multiple mappings:** The server may allow several connections to share the same `clientId` and simply choose one when routing messages (undefined behavior).

In either case, the lack of a "*reject duplicate registration*" policy is a direct enabler of impersonation: the attacker can take over an existing identifier at any time.

3. What information does an attacker need to perform this attack? The attack is remarkably cheap in terms of required information. The attacker needs:

- **The signaling server endpoint:** e.g. `ws://localhost:8080` or, in a real deployment, `wss://signaling.example.com/ws`.
- **A valid client identifier:** e.g. `"client-a"` or `"client-b"`. In our demo these IDs are obvious from the UI or from static JavaScript. In a real system, user IDs or room IDs can often be guessed, enumerated or leaked from logs, URLs or error messages.
- **Basic knowledge of the JSON message format:** in practice, this can be obtained simply by observing one normal registration with a WebSocket proxy (Task 1.1) or browser dev tools.

No credentials, no prior access to the victim's machine and no special network position are required: any host that can connect to the signaling server and guess a valid `clientId` can attempt the attack.

4. Could this attack be performed remotely? How? In the lab, all components run on `localhost`. However, if the signaling server were exposed on a public or corporate network, the very same attack could be executed remotely:

1. The attacker discovers the signaling URL, for example:
   - By inspecting the JavaScript of the web application.
   - From documentation, reverse engineering, or leaked configuration.
2. The attacker runs a modified version of `interceptor_webrtc.py` pointing to that URL:

   ```
   python interceptor_webrtc.py \
       --server-url wss://signaling.example.com/ws \
       --victim-id alice
   ```

3. When another user (Bob) initiates a call to `"alice"`, the offer is delivered to the attacker's script instead of Alice's device.
4. The attacker completes the WebRTC handshake and either:
   - Establishes a direct media channel with Bob (full interception), or
   - Acts as an in-the-middle relay between Bob and Alice, forwarding offers/answers while recording the media.

The only precondition is that the signaling server be reachable from the attacker's network. There is no requirement for local access to the victim's machine.

5. What is the real-world impact of this vulnerability? In a realistic WebRTC application (video conferencing, telemedicine, customer support, etc.), this class of vulnerability has severe consequences:

- **Confidentiality breach:** Audio and video that users believe are exchanged directly with a trusted party can be recorded by an attacker without their knowledge. This includes sensitive conversations, private images and business-critical meetings.
- **Impersonation and fraud:** The attacker appears as the legitimate identity (`"alice"`, `"support"`, etc.). Other participants may share secrets or authorize actions based on a falsely trusted identity.
- **Selective interception or targeting:** Because registration is per-identifier, an attacker can selectively hijack "high value" identities (e.g. an executive's account) and ignore others, making detection harder.
- **Man-in-the-middle potential:** With additional logic, the attacker can forward signaling and media between two honest peers while eavesdropping or altering the content, enabling phishing, manipulation or injection of forged video/audio.

- **Regulatory and legal exposure:** In sectors such as healthcare or finance, such an interception vulnerability would likely violate confidentiality requirements (e.g. GDPR, HIPAA) and could lead to serious legal and reputational damage.

In summary, the vulnerability transforms the signaling server into a weak single point of failure: if it cannot reliably authenticate who is behind a `clientId`, the security guarantees of end-to-end media encryption at the WebRTC layer are undermined.

Root Cause Explanation

At a deeper level, the root cause of the vulnerability is a **broken trust model** in the signaling design:

- The system treats `clientId` as a trustworthy long-term identity, even though it is:
  - Client-chosen.
  - Not cryptographically bound to any secret or account.
  - Not validated for uniqueness or ownership on the server.
- There is no separation between:
  - *Transport-level identity* (a particular WebSocket connection).
  - *Application-level identity* (the logical user or device the connection claims to represent).
- The protocol and code were designed to satisfy functional requirements (routing offers/answers) but not security requirements (authenticating and authorizing who can register which ID).

This leads to the classic anti-pattern of *"identifier = identity = authorization"*: possession of a string (`"client-b"`) is treated as sufficient proof that the caller *is* client B and may receive all messages intended for them.

Impact Assessment

Based on our experiments in Task 2.3:

- The attacker can reliably:
  - Hijack registrations for existing clients.
  - Intercept offers and ICE candidates intended for those clients.
  - Establish a WebRTC media session with an honest peer.
  - Record audio and video from the call to disk.

- There is no visible warning or error in the browser UI: from the caller's perspective, the call simply connects and appears normal (except that the victim may never see it).
- The vulnerability affects *all* users of the system, because any `clientId` is spoofable.

Therefore, the overall impact can be classified as:

- **High confidentiality impact:** full interception of media streams.
- **High authenticity impact:** arbitrary impersonation of any signaling identity.
- **Potential integrity impact:** with a more advanced attacker that relays or modifies media, users may be misled or manipulated.

Even though our lab setup is local and for educational purposes, a similar flaw in a production signaling server would be considered a critical vulnerability.

Attack Detection Strategies

Although the current lab server does not implement any explicit detection, several practical strategies could be used in a real system to detect or at least *raise suspicion* of such attacks:

- **Logging and alerting on duplicate registrations:**
  - Log every registration with `clientId`, timestamp, source IP and user-agent.
  - Trigger an alert when the same `clientId` is registered from multiple IP addresses or devices within a short time window, especially if an existing session is still active.
- **Concurrency checks:**
  - Disallow or warn about multiple simultaneous connections claiming the same `clientId`.
  - Force explicit user re-authentication when a "takeover" of an identity is requested.
- **Anomaly detection in signaling patterns:**
  - Monitor for unusual sequences such as frequent re-registrations of the same ID, or registrations that occur only briefly before incoming calls.
  - Flag endpoints that never send media but only receive (consistent with a pure recording attacker).

- **Endpoint awareness and user notifications:**
  - Notify users when their account/identity is used from a new device/IP.
  - Provide UI indicators showing active devices or sessions for each account.
- **Correlation with ICE / network-level data:**
  - Compare the IP ranges of ICE candidates and WebSocket connections to expected locations. Sudden changes (e.g. a user normally in Greece now appearing from another country) can be flagged.
- **Security-oriented logging of media sessions:**
  - Record which identities participated in which sessions, from which IPs, and for how long, to support forensic analysis if suspicious activity is reported.

These strategies do not *fix* the root cause (missing authentication and authorization), but they provide a detection and response layer that can help identify ongoing or past exploitation of signaling weaknesses.

In conclusion, Task 3.1 shows that the media interception attack of Task 2.3 is not a narrow implementation bug but the natural consequence of a flawed signaling design. Without strong binding between `clientId` and a verified user identity, the signaling server cannot distinguish honest clients from attackers, and the security of WebRTC end-to-end encryption becomes largely irrelevant.

3.2 Task 3.2: Propose Mitigations

In this task we propose concrete mitigations that would prevent the attacks implemented in Tasks 2.1–2.3. We focus on two main hardening measures at the signaling layer:

1. Authenticated registration using signed tokens (e.g. JWTs).
2. Session management that binds WebSocket connections to authenticated sessions and enforces unique client identities.

For each mitigation we describe how it works, how it prevents the attack, the implementation complexity, and the main limitations or trade-offs.

Mitigation 1: Authenticated Registration with Tokens

How it works. Instead of allowing arbitrary JSON messages of the form:

```
{ "type": "register", "clientId": "client-b", "meta": {...} }
```

the signaling server requires that every registration include a *signed authentication token*, for example a JWT (JSON Web Token):

```
{
  "type": "register",
  "clientId": "client-b",
  "token": "<signed JWT>",
  "meta": { "displayName": "client-b" }
}
```

The token is issued by the application backend over HTTPS when the user logs in. It is cryptographically signed by the server and encodes:

- The authenticated user identity (e.g. `"sub": "user123"`).
- The allowed signaling identifiers (e.g. `"clientIds": ["client-b"]`).
- An expiration time and possibly device or session identifiers.

On receiving a `"register"` message, the signaling server:

1. Verifies the token signature and checks that it is not expired.
2. Checks that the requested `clientId` is contained in the token's allowed set.
3. Binds the WebSocket connection to both the user identity and the `clientId`.

If any check fails, the registration is rejected.

How it prevents the attack. In our attack, the attacker simply connects and sends:

```
{ "type": "register", "clientId": "client-b", ... }
```

with no proof that they are the legitimate owner of `"client-b"`. Under the token-based scheme:

- The attacker would need a *valid* token that authorizes `"client-b"`.
- Tokens are only issued after user login (e.g. password, OAuth, SSO), so random external attackers cannot obtain such a token.
- Even if the attacker guesses the `clientId`, the server will reject the registration because the token and `clientId` do not match.

Thus, knowing or guessing a `clientId` is no longer sufficient to impersonate that identity. The attack in Task 2.3 fails at the very first step (registration).

Implementation complexity.

- **Moderate**: many web backends already implement user authentication and session handling.
- Requires:
  - A token issuing component (login endpoint, JWT library).
  - Secret key management (for signing/verifying tokens).
  - Changes in the signaling server to parse and validate tokens on `"register"`.
  - Small modifications in client-side JavaScript to include the token when registering over WebSocket.
- Tokens integrate well with existing infrastructures (reverse proxies, API gateways), and the logic is relatively straightforward once the auth backend exists.

Limitations and trade-offs.

- **Token theft**: if an attacker compromises the browser or steals the token (e.g. via XSS or local malware), they can still impersonate the user until the token expires.
- **Revocation complexity**: short-lived tokens mitigate theft, but require refresh flows; revoking a specific token before it expires can be non-trivial.
- **No device-level binding**: tokens authenticate the user, not the physical device; a remote attacker with the token is indistinguishable from the real user from the server's perspective.

Despite these limitations, token-based authentication removes the catastrophic weakness where `clientId` alone determines identity.

Mitigation 2: Session Management and Unique Client Identity Binding

How it works. In addition to authentication tokens, the signaling server can implement stricter *session management* for WebSocket connections. The main ideas are:

- **Bind WebSocket connections to authenticated HTTP sessions:**
  - The user first authenticates via HTTPS and receives a secure cookie (e.g. `HttpOnly`, `Secure`, `SameSite`).
  - When opening the WebSocket to the signaling server, the browser includes this cookie, allowing the server to map the connection to the logged-in user.
- **Enforce one active registration per clientId per user:**

  - The server maintains a table mapping each `clientId` to a single active WebSocket connection (and user).
  - If a second connection attempts to register the same `clientId`, the server can:
    * Reject the new registration outright; or
    * Explicitly tear down the old connection and log an event "session takeover"; or
    * Require additional confirmation (e.g. a second factor) before allowing the takeover.

- **Track session metadata:** IP address, user agent, timestamps, and possibly device identifiers, to support anomaly detection.

How it prevents the attack. This mitigation addresses both the *who* and the *how many* aspects of identity:

- The attacker can no longer open an "anonymous" WebSocket and freely choose a `clientId`; the server will associate the connection with a specific authenticated user session.
- If the attacker is not logged in as the legitimate user behind `"client-b"`, the registration is rejected.
- Even if the attacker somehow shares the same user account, the uniqueness rule for `clientId` ensures:
  - Either the original session is preserved and the attacker is refused; or
  - A visible "session takeover" occurs, which can be logged and notified to the user.

Our Task 2.3 attacker relies exactly on the ability to silently *add* another registration for `"client-b"` from a different process and IP. With strict session binding and uniqueness, this behavior becomes either impossible or highly suspicious and detectable.

Implementation complexity.

- **Moderate to high**, depending on the existing architecture:
  - If the application already uses secure cookies and a central session store, binding WebSocket connections to sessions is straightforward.
  - Implementing robust uniqueness checks and "takeover policies" requires careful design to avoid locking out legitimate multi-device use cases (e.g. user logged in from a laptop and a phone).
- Requires persistent state on the signaling server (or a shared store like Redis) to track which `clientId` is active where.

Limitations and trade-offs.

- **Multi-device support**: users may legitimately want the same identity on multiple devices; strictly enforcing "one connection per `clientId`" may be too restrictive and require more sophisticated policies.
- **Stateful scaling**: keeping per-identity session state complicates horizontal scaling of the signaling server; stateless designs are easier to scale.
- **Not sufficient alone without authentication**: uniqueness checks without strong authentication would still allow an attacker to "legitimately" register the ID first.

Overall, session management and unique binding strongly complement token-based authentication, by preventing silent hijacks and making session takeovers observable.

Comparison of Approaches

Security strength.

- **Authentication tokens** directly address the root cause from Task 3.1: they ensure that only authorized users can register a given `clientId`. They provide a strong cryptographic binding between user identity and signaling identity.
- **Session management and uniqueness** focus on preventing and detecting *simultaneous* or *surprise* registrations for the same ID, which stops silent takeovers and aids in detection.

Implementation effort.

- Tokens are relatively easy to add if an auth backend already exists; many frameworks provide JWT support out of the box.
- Session management with uniqueness requires more invasive changes in the signaling server (state tracking, concurrency handling, multi-device policy design).

Attack resilience.

- Token-based schemes are vulnerable if tokens are stolen, but otherwise robust against remote attackers.
- Session management can detect anomalous behavior (e.g. sudden re-registrations from new IPs) and support forensic analysis, even if an attacker temporarily acquires valid credentials.

In practice, the two mitigations are complementary rather than mutually exclusive: tokens provide *who you are*, while session management governs *how and where that identity is used*.

Recommendation

Our recommended design for a real-world WebRTC deployment is:

- **Primary mitigation:** introduce *authenticated registration* using signed tokens (or secure cookies) so that every `clientId` used in signaling is tied to a verified user identity and cannot be freely chosen by arbitrary WebSocket clients.
- **Secondary mitigation:** implement *session management and unique identity binding* on the signaling server to:
  - Prevent silent duplicate registrations for the same `clientId` from different network locations.
  - Log and optionally require confirmation for session takeovers.

Together, these mitigations would block the attacks demonstrated in Tasks 2.1–2.3: an external attacker who only knows the signaling endpoint and a victim's `clientId` would no longer be able to impersonate that identity or intercept its media.

4 Bonus Task – Securing WebSocket Signaling with TLS/SSL

Goal

The goal of the bonus task was to harden the WebRTC signaling channel by migrating from plaintext WebSocket (`ws://`) to encrypted WebSocket over TLS (`wss://`). Concretely, we wanted to:

- Generate a self-signed certificate suitable for local development.
- Run the signaling server over HTTPS/WSS on `https://localhost:8443`.
- Update the WebRTC clients so they connect via `wss://` instead of `ws://`.
- Verify that all WebSocket traffic is encrypted and that our previous registration-hijacking attacker can no longer eavesdrop or modify signaling in a man-in-the-middle position.

Certificate Generation and Server Configuration

For the lab environment we generated a self-signed certificate and key using `openssl`. An example command is:

```
openssl req -x509 -newkey rsa:2048 \
    -keyout key.pem -out cert.pem \
    -days 365 -nodes -subj "/CN=localhost"
```

The files `cert.pem` and `key.pem` are then mounted into the signaling server container under `/certs/`. The modified signaling server (`server.js`) creates an HTTPS server using these files and attaches a WebSocket server to it:

```
const https = require("https");
const fs = require("fs");
const WebSocket = require("ws");

const server = https.createServer({
  key: fs.readFileSync("/certs/key.pem"),
  cert: fs.readFileSync("/certs/cert.pem"),
});

const wss = new WebSocket.Server({ server });

server.listen(8443, () => {
  console.log("Secure signaling server running on https://localhost:8443");
});
```

The application logic for registration and message routing is unchanged: clients still send JSON messages with `type`, `clientId`, and optional `to`/`from` fields; only the transport is now protected by TLS.

Client-Side Changes (WSS Configuration)

The WebRTC demo clients are served by a small Express application (`webrtc-static-server.js`) which also exposes a `/config` endpoint. We updated this endpoint so that clients automatically receive the secure signaling URL:

```
app.get('/config', (req, res) => {
  const wsHost = "wss://localhost:8443";
  res.json({
    SIGNALING_URL: wsHost,
    NAME: process.env.NAME || 'client',
    ROLE: process.env.ROLE || 'caller',
    TOKEN: process.env.REG_TOKEN || null
  });
});
```

On the browser side, the JavaScript client simply reads the `SIGNALING_URL` field from `/config` and constructs its WebSocket as:

```
const ws = new WebSocket(config.SIGNALING_URL); // wss://localhost:8443
```

Because the certificate is self-signed, Firefox initially displays a warning page ("Potential Security Risk Ahead"). For the development environment we explicitly accept this certificate once . After that, developer tools show the

WebSocket connection with a lock icon and the tooltip “The connection used to fetch this resource was secure”.

(You can include the actual screenshots as figures if you like:)

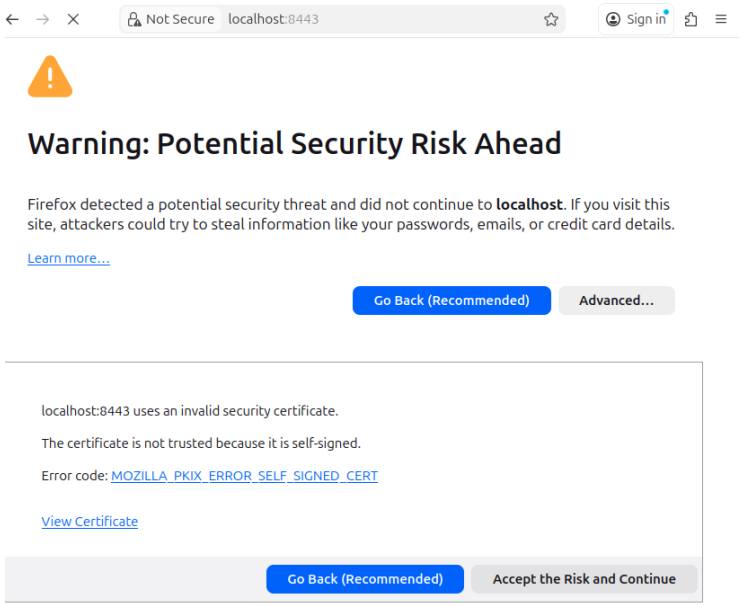


Figure 1: Firefox warning when accessing https://localhost:8443 with a self-signed certificate.

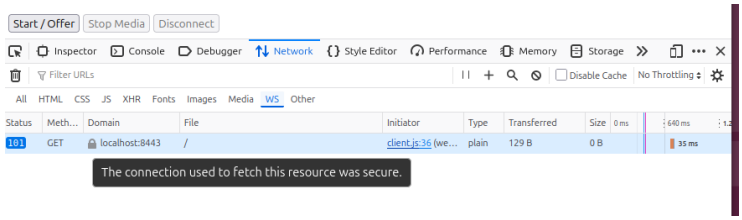


Figure 2: Browser developer tools showing an established secure WebSocket connection (WSS).

Testing and Observations

We performed the following tests:

1. **Normal call over WSS.**
   With the HTTPS/WSS server running on `8443`, we opened the WebRTC

clients in the browser. After accepting the self-signed certificate, the clients successfully registered and completed a WebRTC call using `wss://localhost:8443` as their signaling endpoint.

2. **Encrypted signaling traffic.**
   Inspecting the network traffic (e.g. via browser DevTools or packet capture) shows that the WebSocket handshake and subsequent frames are carried inside TLS records. The signaling JSON messages (registration, offers, answers, ICE candidates) are no longer visible in plaintext.

3. **Effect on the existing attacker.**
   Our previous `ws://`–based attacker cannot transparently intercept or modify signaling without being part of the TLS trust chain. An off-path attacker that does not control the certificate or DNS can no longer mount a trivial man-in-the-middle on the signaling channel.

Security Analysis of TLS/WSS Mitigation

TLS/WSS directly addresses several issues identified in the security analysis:

Confidentiality and integrity. All signaling messages are now protected by TLS, providing:

- **Confidentiality**: third parties on the network cannot read JSON messages such as `"register"`, `"offer"`, or `"ice"`.
- **Integrity**: attackers cannot modify signaling messages in transit without detection.

This prevents classic on-path man-in-the-middle attacks where an attacker simply sits between client and server and alters `clientId` values or SDP content.

Server authentication. Even with a self-signed certificate, the browser now authenticates that it is talking to a specific TLS endpoint (the certificate we accepted earlier). In a production deployment, the certificate would be issued by a trusted CA, preventing an attacker from impersonating the signaling server without compromising the CA trust model.

Limitations. TLS by itself *does not* solve the core logical vulnerability that the signaling server accepts arbitrary `clientId` values. If an attacker can legitimately reach the real signaling server and send WebSocket messages over WSS, they can still attempt to register as `client-a` or `client-b`. To fully prevent registration hijacking, TLS must be combined with application-level mitigations such as authentication tokens or per-user credentials (as discussed in Task 3.2).

Nevertheless, TLS significantly raises the bar:

- Off-path network attackers can no longer passively observe or modify signaling.

- On-path attackers must now break or bypass TLS rather than simply sniffing and injecting unencrypted WebSocket frames.

Summary

The bonus task demonstrates that the WebRTC signaling channel can be secured using standard Web technologies:

- The signaling server was migrated from `ws://` to `wss://` by wrapping it in an HTTPS server using a self-signed certificate.
- The WebRTC clients were updated to use the new secure endpoint and to fetch configuration dynamically from a `/config` REST endpoint.
- Network traces now show encrypted TLS records instead of readable JSON messages, and the previous attacker can no longer transparently intercept or tamper with signaling traffic.

When combined with the logical mitigations proposed in Task 3.2 (such as authentication tokens and stricter session management), this transport upgrade provides a solid foundation for a more secure WebRTC deployment.

5 Conclusion

In this assignment we moved from basic protocol observation to a full end-to-end security evaluation of a WebRTC signaling system. In Task 1 we implemented a WebSocket proxy that transparently relayed traffic between browser clients and the original signaling server, allowing us to capture and document the registration protocol in detail. This reverse-engineering step made explicit how identifiers, message types and routing fields are used in practice, and already revealed that the design relied entirely on self-declared `clientId` values without any authentication.

Building on this understanding, Task 2 showed that this design flaw is not merely theoretical. By implementing an active media interception attacker with `aiortc`, we demonstrated that an adversary who can reach the signaling server and guess a valid `clientId` can register as a victim, receive their offers and ICE candidates, complete the WebRTC handshake and record the resulting audio/video stream. From the honest caller's point of view, the call appears normal, highlighting how signaling weaknesses can silently undermine the confidentiality and authenticity of otherwise encrypted media channels.

In Task 3 we systematized these observations into a vulnerability analysis and mitigation plan. We identified the root cause as a broken trust model where the server treats a bare identifier as identity and authorization. We then proposed concrete countermeasures: authenticated registration based on signed tokens and stricter session management with unique identity binding and better logging. These measures directly address both who is allowed to use a

given `clientId` and how that identity can be used across concurrent sessions, making registration hijacking significantly harder and easier to detect.

Finally, in the bonus task we complemented these logical mitigations with transport-layer protection by migrating from `ws://` to `wss://` using TLS. The resulting setup prevents passive eavesdropping and simple man-in-the-middle attacks on the signaling channel, and our tests confirmed that WebSocket traffic is now encrypted and that the original `ws://`-based attacker can no longer observe the JSON signaling messages. At the same time, our analysis emphasized that TLS alone is not sufficient: without proper authentication and session handling at the application layer, an attacker who legitimately reaches the signaling endpoint can still attempt to impersonate users.

Overall, the project illustrates an important lesson for secure real-time communication systems: end-to-end encryption at the media layer is only as strong as the signaling infrastructure that binds identities to cryptographic endpoints. Secure WebRTC deployments must therefore combine transport security (TLS), robust authentication/authorization and careful session management to protect both signaling and media from powerful adversaries.

6 External Links

- Live Demo of Media Interception
- The GitHub Project Repository