HSEC Security Systems Protocol

System Overview

OVERVIEW

The system is made of 2 servers, any amount of camera clients and any amount of normal clients. One server is named ***Camera Server***, the other is called ***Client Handler Server.*** The camera server has 2 sockets - One to discover cameras and the other to communicate with connected cameras. The client handler server works with websockets since the normal clients use the website which must use websockets.

We have 2 different databases. One will store camera data and the other, user data. The camera database has: the Mac Address, the last frame, the camera name, and the AES key. Its used to reconnect cameras and to avoid exchanging keys every-time (The mac address is used as the primary key). The user database has: the user email, their password hash, their current session token, session expiry, linked cameras and reset code data.

PROTOCOL

If a TCP message is sent then the message will be prefixed by 4 total bytes (big endian encoding) that describe the length of the message that comes after them. Each message sent in TCP is encrypted with CBC AES.
It would look like this: *00 00 00 12* **0x0** *{ACTION}* **0x0** *{...FIELDS}*
The protocol is byte orientated and is completely event based and asynchronous.

CODE

The code of the Camera Server can be found here
The code of the Client Handler Server can be found here
The code of the Camera Client can be found here
The code of the Client can be found here
Full project code can be found here

The Camera

The camera is a modification of the ESP32-CAM (Can be found here).

OVERVIEW

The camera first sends heartbeats to the broadcast address in hopes the camera server will respond and let the camera know where it can connect. Before the camera will connect it will have to verify if the server that wants to connect is authorized by checking a hardcoded 4 digit code. If its verified it will connect to the server and both will start a 32 bit AES key exchange (using ECDH) - The current AES algorithm is CBC. If the exchange is successful the camera will immediately start transmitting frames from the camera encrypted with the same AES key. If the camera loses power - it will check its internal storage for a saved key and server, if found it will attempt to connect infinitely, it will do the same if it loses connection from the server. If the camera needs to be paired to a new server, a 3 second press on the “BOOT” button located on the ESP board will reset the server data and put the camera back in discovery mode.

PROTOCOL

The camera has 3 states - DISCOVERING, LINKED, REPAIR.
When the state is *DISCOVERING:*
Camera To Broadcast *(heartbeat)*: CAMPAIR-HSEC **0x0** *{CAMERA_MAC}*
Camera Server To Camera *(pair)*: CAMACK-HSEC **0x0** *{SERVER_PORT}* **0x0** *{CAMERA_CODE}*
Camera Server To Camera *(server hello):* exch **0x0** *ecdh* **0x0** *aes* **0x0** *{SERVER_PUB_KEY}*
Camera To Camera Server *(camera hello):* exch **0x0** *ecdh* **0x0** *aes* **0x0** *{CAMERA_PUB_KEY}*
Camera To Camera Server *(camera confirm):* {ENCRYPTED_CONFIRMATION}
Camera Server To Camera *(server confirm):* {ENCRYPTED_CONFIRMATION}
STATE = LINKED

When the state is *LINKED:*
Camera To Camera Server *(camera frame):* CAMFRAME-HSEC **0x0** {ENCRYPTED_FRAME}

When the state is *REPAIR:*
Camera To Camera Server *(relink)*: CAMRELINK-HSEC **0x0** *{CAMERA_MAC}*
Camera Server To Camera *(server confirm)*: CAMREPAIR-HSEC **0x0** *{ENCRYPTED_CONFIRM}*
Camera To Camera Server *(client confirm)*: CAMREPAIRACK-HSEC **0x0** *{ENCRYPTED_CONFIRM}*
STATE = LINKED

When the code is incorrect*:*
Camera To Camera Server *(relink)*: BADCODE-HSEC **0x0** *{CAMERA_MAC}*

The Camera Server

OVERVIEW

The purpose of the camera server is to provide outside code with the tools to interact with the cameras - it mostly doesn't act on its own. The server will listen on a UDP socket for camera heartbeats and will connect to them using a pin code if the user decides they want to. The server can also verify a camera by checking its mac prefix (custom one for HSEC). The exchanges on the UDP socket include only heartbeats and pairing requests. The camera server has the DB of cameras and that will allow outside code to use the following functionalities:

- Discover Cameras: This will start the heartbeat listening (CAMPAIR-HSEC) process and will call a callback when a camera is found.

- Pair camera: This will send the pairing request to the camera with a code provided by the user (CAMACK-HSEC) - This will either be successful (camera will connect to the server) or fail (BADCODE-HSEC).

- Start camera stream: Usually the callbacks for frames are off and the camera server will process those frames alone - when this is called with a mac address of a camera the camera server will call a specified callback with the frame data.

- Stop camera stream: This will disable the callback mentioned above.

- Other use cases using the database, including: Renaming, Camera data, and more.

See protocol in the **Camera** section.

The Client Handler Server / Main Server

OVERVIEW

The purpose of this server is to interact with connecting clients and use the camera server to interact with cameras. Since the users interact with the server using JS which is known for its asynchronous abilities, I implemented a system using “transaction ids”. The client will send a certain request and will include a transaction ID. The function they called to send that request will store that request in an awaiting_response list while waiting for a response matching the transaction ID. If it arrives the “Promise” will resolve, if it does not within a certain time the function will reject with a Timed Out error. This protocol will be different for streaming requests like discover cameras, and stream camera. In those cases- the function they called will add the transaction id to a streaming list and when the Client Handler Server receives a transaction ID that should be streamed it will keep it and send responses to it, then when the client wants to stop the stream the client will send a message to the server and the server will automatically remove the transaction ID from the streaming function (the client will too).

Note: The protocol this server uses to exchange data with the client uses JSON. In the protocol description below I will remove the transaction_id key for simplicity, will also remove the “success” key from the server response for simplicity (appears after commands like rename, pair, and more).

PROTOCOL

Discovery:

Client To Server *(start discovery):* {type: “discover_cameras”}

Server To Client *(started discovery):* {status: ***status,*** data: “Discovery started”}

Server To client *(camera discovered):* {status:***status,*** data: {mac:***mac***, ip:***ip***, port:***port***, type: “camera_discovered”}}

Client To Server *(stop discovery):* {type: “stop_discovery”}

Server To Client *(stopped discovery):* {status: ***status,*** data: “Discovery stopped”}

Streaming:

Client To Server *(start stream):* {type: “stream_camera”, mac: ***camera_mac***}

Server To Client *(started stream):* {status: ***status,*** data: ***reason***}

Server To Client *(camera frame):* {status: ***status,*** data: {mac: ***camera_mac***, frame: ***frame***, type: “frame”}}

Client To Server *(stop stream):* {type: “stop_stream”, mac: ***camera_mac***}

Server To Client *(stopped stream):* {status: ***status,*** data: ***reason***}

Renaming:

Client To Server *(renaming camera):* {type: “rename_camera”, mac: ***camera_mac***, new_name: ***camera_mac***}

Server To Client *(renamed camera):* {status: ***status,*** data: ***reason***}

Pairing:

Client To Server *(pair camera):* {type: “pair_camera”, mac: ***camera_mac***, ip: ***camera_ip***, port: ***port***, code: ***code***}

Server To Client *(paired camera):* {status: “success”, data: {mac: ***camera_mac***, ip: ***camera_ip***}}

Server To Client *(failed pairing camera):* {status: “error”, data: ***reason***}

Sharing:

Client To Server *(share camera):* {type: “share_camera”, mac: ***camera_mac***, email: ***share_to_email***}

Server To Client *(shared camera):* {status: ***status,*** data: ***reason***}

Account related:

Client To Server *(password login):* {type: “login_pass”, email: ***email***, password: ***password***}

Server To Client *(password logged in):* {status: ***status,*** data: {session_id: ***token***, info: ***info***}}

Client To Server *(session login):* {type: “login_session”, email: ***email***, session_id: ***session_token***}

Server To Client *(session logged in):* {status: ***status,*** data: {session_id: ***token***, info: ***info***}}

Client To Server *(signup):* {type: “signup”, email: ***email***, password: ***password***}

Server To Client *(signed up):* {status: ***status,*** data: {session_id: ***token***, info: ***info***}}

Client To Server *(password reset request):* {type: “request_password_reset”, email: ***email***}

Server To Client *(password reset request sent):* {status: ***status,*** data: {info: ***info***, time_left: ***time_left***}}

Client To Server *(password reset):* {type: “reset_password”, email: ***email***, reset_code: ***code***, new_password: ***pass***}

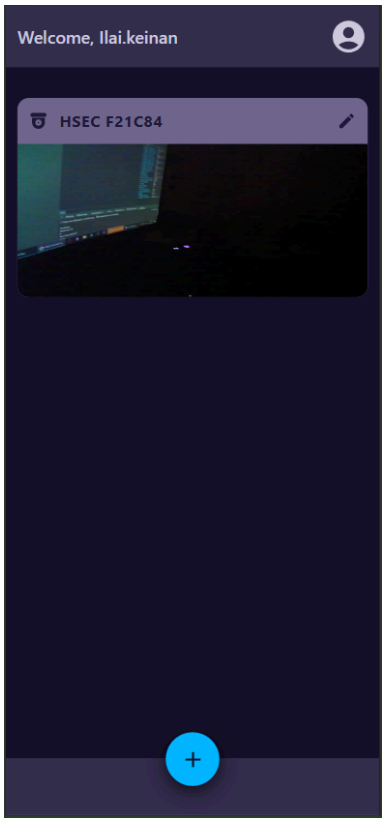
Server To Client *(password reset):* {status: ***status,*** data: {info: ***info***}}

The Client

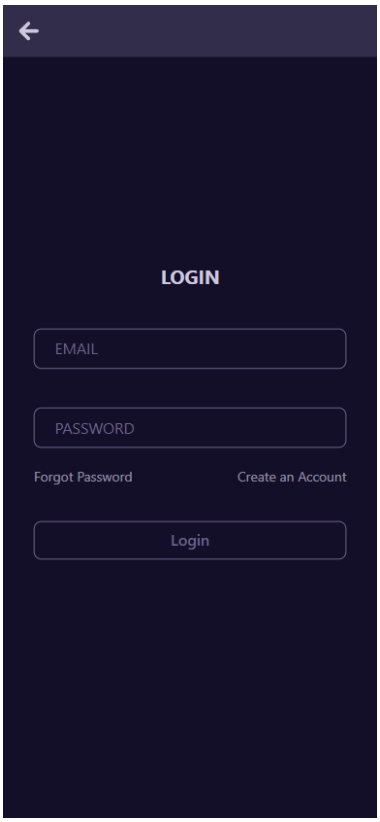
OVERVIEW

The client has several screens and connects with the Client Handler Server.

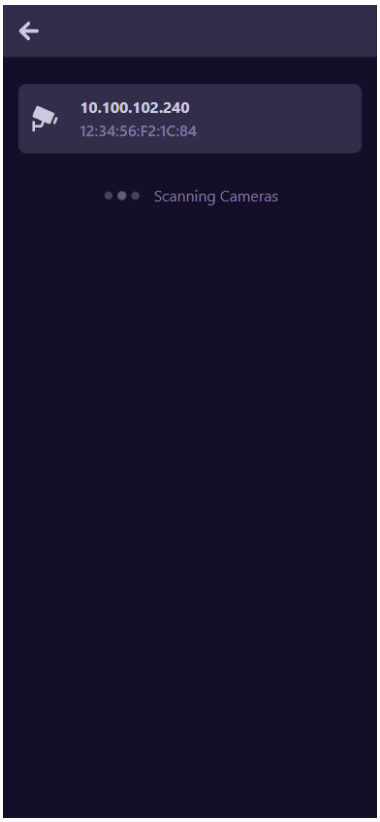
the main screen:



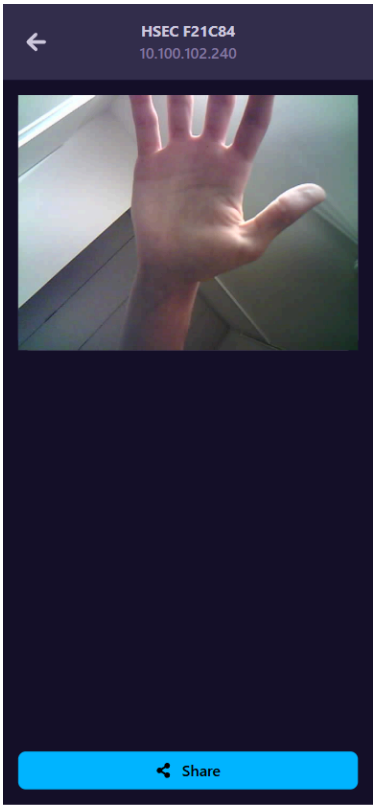
The login screen:



The discovery screen:



The stream screen:



Known Vulnerabilities

- Since we reuse the IV instead of letting it scroll, our system in vulnerable to AES replay attacks. For example, when re-pairing a camera the server will expect the camera to send it a confirm message encrypted with the AES key and the other way around, if a MITM would record that exchange the first time they'd be able to impersonate the camera and/or the server. The fix would be to use AES correctly and use a nonce.

- The problem described above could also let an attacker figure out the AES key if they know the confirm message which are currently hard-coded. Same fix as above.

- We have a possibility of SQL injection. The fix would be to sanitize the user input before querying the SQL Database.

- When the server asks to pair to a camera it will send it its code (without encryption) so the camera could confirm the owner. The fix would be first making a secure connection using ECDH and AES, the problem is the camera has only 2 sockets available so we could use the maximum buffer size for camera frames so we wont be able to do this without making the camera unavailable when a pairing request is sent.

- The client and Client Handler Server talk using Websockets that currently are not using TLS. This problem could be solved easily when the client's IP will be linked to a domain, currently its not.

Known Issues / TODO

- We could optimize streaming by using AES that works for streaming. That way we'd be able to stream using UDP and send smaller chunks faster.

- Complete the playback feature