Data Integrity Security Project Rapport
by Oliver Joisten

Background and Explanation of the Problem

Protecting our privacy in today's digital world is more important than ever. With cyber threats becoming increasingly common, securing our data during transmission is essential. This project focused on building a secure communication system between a user and a server, using serial communication.

I used Python for the program logic and Tkinter to create a user-friendly graphical interface (GUI) on the client-side. This interface interacts with an ESP32 microcontroller. The system allows users to control functions on the ESP32, like turning on an LED or checking its internal temperature.

To ensure the highest level of data security, we implemented robust cryptographic techniques like HMAC-SHA256, AES-256, and RSA-2048. Additionally, i leveraged a GitHub repository and a Kanban board to keep the project organized throughout the development process.

Approach to Problem

To safeguard data during transmission, we incorporated three key security measures:

- **HMAC-SHA256 (Hash-based Message Authentication Code)**
  This industry-standard technique verifies data integrity, ensuring messages aren't altered in transit. Think of it like a digital fingerprint that detects any tampering. (Wikipedia, org (2012))
- **AES-256 (Rivest–Shamir–Adleman)**
  This powerful encryption method scrambles the data, making it unreadable to anyone without the decryption key. It's like locking your message away in a secure vault. (Wikipedia, org (2024))

- **RSA-2048 (Advanced Encryption Standard)**
  This robust system enables secure key exchange, allowing the client and server to share encryption keys safely. Imagine it as a secure handshake to establish a private communication channel. (Wikipedia, org (1977))

These robust and widely recognized cybersecurity techniques were implemented using These well-established cybersecurity techniques were implemented using the mbedtls library, (Rodgman, D. (2011)) a trusted toolkit for building secure applications. The

client application itself was developed in Python, while the server program runs on an ESP32 microcontroller. Serial communication, a reliable method for transferring data over a physical connection, facilitates communication between the two.

Execution

Software Design

To streamline development, the software was designed with two distinct parts: a client application and a server program. This separation allowed me to work on, test, and improve each component independently. Think of it like building two sides of a bridge at the same time - it gets the whole project finished much faster.

Software Components Overview

GitHub Repository and Project Management

To keep everything organized, i used a GitHub repository for version control. This lets me track changes to the code and easily revert if needed, like having a rewind button for the project. Additionally, i utilized a Kanban board to manage tasks. Think of it as a visual to-do list that keeps everyone on the same page and helps me move tasks smoothly through different stages of completion.

Client Application

The client application, built with Python, offered a user-friendly interface thanks to Tkinter. It packed several handy features:

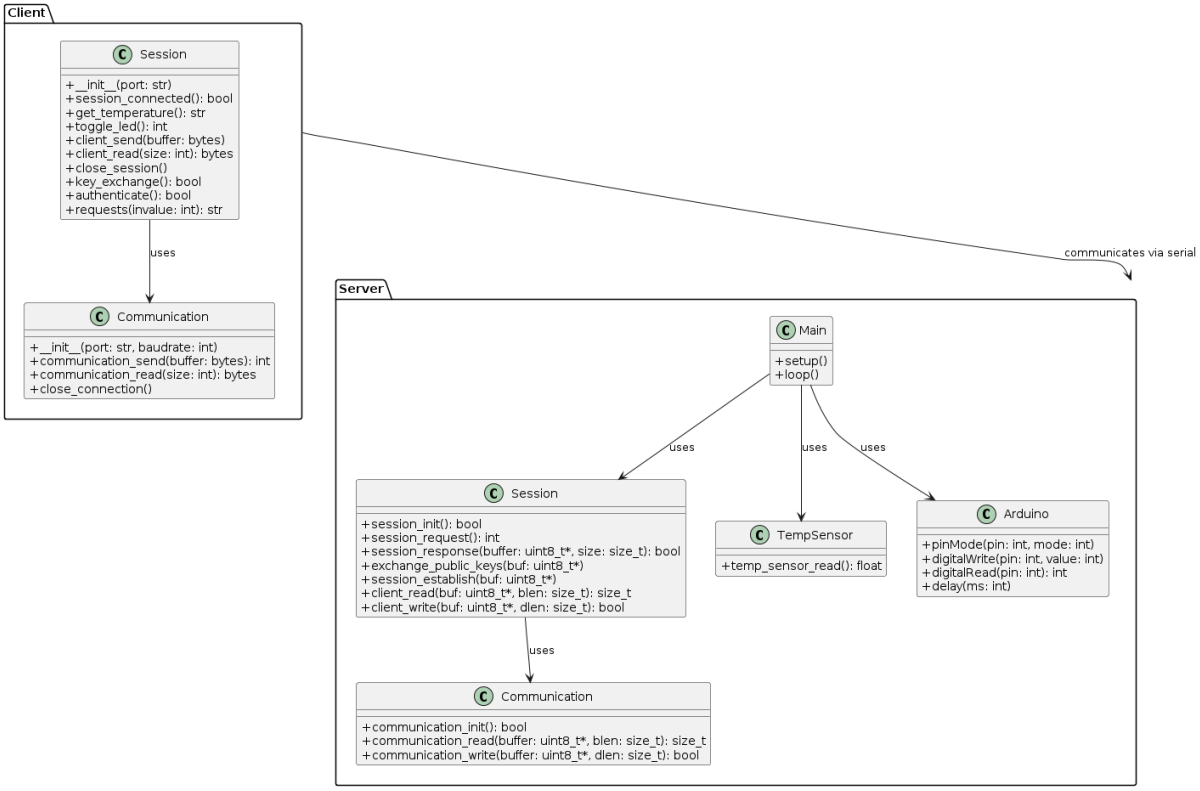
- **Finding available connections:** No need to fumble around - the user interface helps you find the right port to connect to your ESP32.
- **Starting and stopping connections:** Easily connect and disconnect with the click of a button.
- **Checking the ESP32's temperature:** Keep an eye on your device's health with a quick temperature check.
- **Controlling the LED:** Like a virtual light switch, you can turn the LED on and off from the user interface.
- **Keeping track of actions:** The app logs your actions, so you can see what's been happening. Need a clean slate? You can clear the logs too.
- **Seamless updates:** The interface automatically reflects the connection status, keeping you informed.

Server Communication

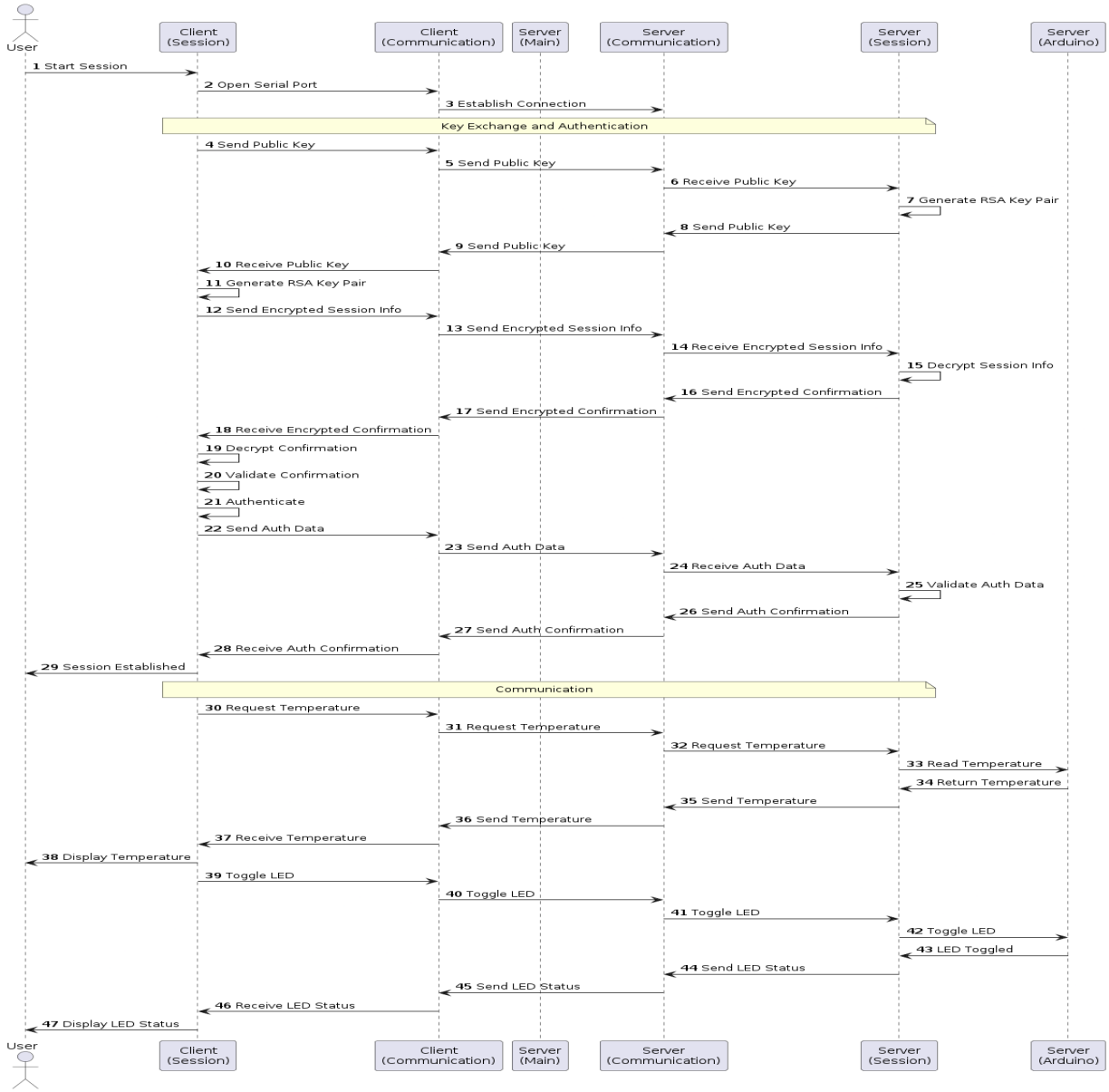
The ESP32 microcontroller acts as the brain behind the server side, using the familiar Arduino library to talk back and forth. To keep your data safe during its journey, i employed powerful security measures built into the mbedtls library. Think of it like a high-tech vault that protects your data with three layers of security:

- **Data Check: HMAC-SHA256** makes sure nothing gets tampered with on the way, like a digital fingerprint that verifies everything arrives intact.
- **Data Scramble: AES-256** scrambles the data, making it unreadable to anyone who shouldn't see it. Imagine it like locking your messages in a secure code.
- **Secure Key Exchange: RSA-2048** ensures only authorized devices can access the data, like a secret handshake that keeps everything confidential.

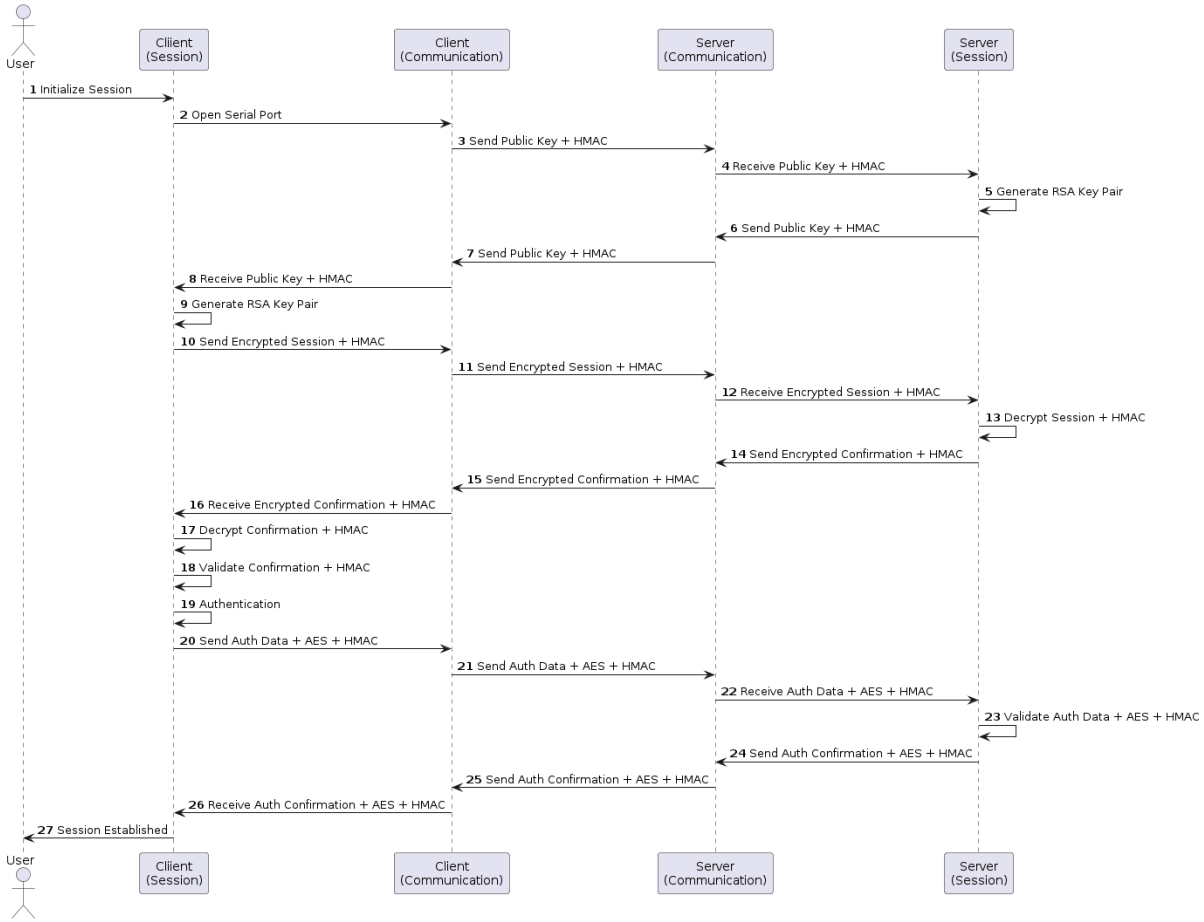
1. UML Diagrams


Client
Session
+__init__(port: str)
+session_connected(): bool
+get_temperature(): str
+toggle_led(): int
+client_send(buffer: bytes)
+client_read(size: int): bytes
+close_session()
+key_exchange(): bool
+authenticate(): bool
+requests(invalue: int): str
uses
Communication
+__init__(port: str, baudrate: int)
+communication_send(buffer: bytes): int
+communication_read(size: int): bytes
+close_connection()
communicates via serial
Server
Main
+setup()
+loop()
uses
uses
uses
Session
+session_init(): bool
+session_request(): int
+session_response(buffer: uint8_t*, size: size_t): bool
+exchange_public_keys(buf: uint8_t*)
+session_establish(buf: uint8_t*)
+client_read(buf: uint8_t*, blen: size_t): size_t
+client_write(buf: uint8_t*, dlen: size_t): bool
TempSensor
+temp_sensor_read(): float
Arduino
+pinMode(pin: int, mode: int)
+digitalWrite(pin: int, value: int)
+digitalRead(pin: int): int
+delay(ms: int)
uses
Communication
+communication_init(): bool
+communication_read(buffer: uint8_t*, blen: size_t): size_t
+communication_write(buffer: uint8_t*, dlen: size_t): bool

System Sequence Diagram



Below is a UML diagram illustrating the secure communication process:



Tools, Methods, and Languages Used

- **Languages**:
  - **Python** for the client-side.
  - **C/C++** for programming the server code for the ESP32.
- **Libraries**:
  - **Tkinter** for the GUI.
  - **Pyserial** for serial communication.
  - **mbedtls** for cryptographic functions.
- **Tools**:
  - **GitHub** for version control.
  - **Kanban Board** for task management.

- **VS Code** for coding.
- **PlatformIO** for deploying and running the server code.

Detailed Execution and Problem Solving

- **Keeping Track:** To stay organized and manage tasks efficiently, i set up a GitHub repository and a Kanban board. Think of it as a digital toolbox with a to-do list, keeping everything on track and everyone in the loop.
- **Building the Client:** The client application was built using Python, with Tkinter creating a user-friendly interface. It offered features like finding available connections, starting and stopping communication, checking the ESP32's temperature, controlling the LED, and keeping track of actions with logs that can be cleared. The interface even updates automatically based on the connection status, creating a smooth user experience.
- **The Server and Talking Tech:** The ESP32 microcontroller acts as the server, using the familiar Arduino library to communicate with the client. For top-notch security, i used the mbedtls library to implement a three-layered vault to protect your data:
- **Data Check: HMAC-SHA256** ensures nothing gets tampered with, like a digital fingerprint verifying everything arrives intact.
- **Data Scramble: AES-256** scrambles the data, making it unreadable to anyone who shouldn't see it. Imagine it like locking your messages in a secure code.
- **Secure Key Exchange: RSA-2048** ensures only authorized devices can access the data, like a secret handshake that keeps everything confidential.
- **Security First:** i designed a secure protocol for all communication and implemented session management that automatically expires after **1 minute** of inactivity to prevent unauthorized access.
- **Testing and Challenges:** Thorough testing verified the effectiveness of encryption and data integrity features. I also ensured sessions automatically terminated after a minute of inactivity. One of the biggest hurdles was securing communication. Debugging was tricky because printing information to the terminal interfered with the connection. My solution involved sending error messages from the server to the client's log window, though these messages weren't always detailed enough. Despite these obstacles, i successfully achieved secure communication

Conclusions

- This project really boosted my understanding of how to build secure connections. I learned how to use powerful tools like HMAC-SHA256, AES-256, and RSA-2048 to make sure data stays safe, can't be faked, and arrives exactly as it was sent. The mbedtls library was a huge help in putting these security layers in place.

Areas for Improvement

- **Smarter Error Handling:** Right now, the system could be better at handling unexpected situations. Imagine a more helpful error message that guides the user instead of just leaving them confused.
- **Communication Options:** The system could benefit from additional ways to talk back and forth. Think of it like having more languages available, making it more versatile. For example, adding TCP/IP would open up new possibilities.
- **Enhanced Interface:** The user interface could be even more user-friendly with some extra features and a visual refresh. Imagine a more polished and intuitive design that makes interacting with the system a breeze.
- **Detailed Documentation:** Creating more detailed documentation for each part of the system would make it easier for others to understand and maintain it in the future. Think of it like a comprehensive instruction manual that leaves no room for questions.

References

- Roseman, M. (2007) *Modern TK Best Practices*, *TkDocs Home*. Available at: https://tkdocs.com/ (Accessed: 05 June 2024).
- Foundation, P.S. (2001) *Graphical user interfaces with TK*, *Python documentation*. Available at: https://docs.python.org/3/library/tk.html (Accessed: 05 June 2024).
- Rodgman, D. (2011) *Knowledge base*, *Knowledge Base - Mbed TLS documentation*. Available at: https://mbed-tls.readthedocs.io/en/latest/kb/ (Accessed: 05 June 2024).
- Wikipedia, org (2012) *HMAC*, *Wikipedia*. Available at: https://en.wikipedia.org/wiki/HMAC (Accessed: 07 June 2024).
- Wikipedia, org (2024) *Advanced encryption standard*, *Wikipedia*. Available at: https://sv.wikipedia.org/wiki/Advanced_Encryption_Standard (Accessed: 07 June 2024).
- Wikipedia, org (1977) *RSA (cryptosystem)*, *Wikipedia*. Available at: https://en.wikipedia.org/wiki/RSA_(cryptosystem) (Accessed: 07 June 2024).

- Arduino, F. (2005) *Serial*, *Serial - Arduino Reference*. Available at: https://www.arduino.cc/reference/en/language/functions/communication/serial/ (Accessed: 05 June 2024).
- Sysop, W. (2011) *C and C++ reference*, *cppreference.com*. Available at: https://en.cppreference.com/w/ (Accessed: 05 June 2024).