**Analysis of ESP32 Firmware and Python Server Code**

**Overview**

1. **ESP32 Firmware (C++)**: A firmware for an ESP32 microcontroller interfacing with a SIM7600 modem for MQTT-based communication, secure provisioning, and Over-The-Air (OTA) firmware updates.
2. **Python Server**: A server-side application using the Paho MQTT library to manage device provisioning and OTA updates, interacting with the ESP32 over an MQTT broker.

Both components are designed to work together in an IoT ecosystem, leveraging MQTT for communication, cryptographic techniques for security, and a state machine for robust operation on the ESP32.

**Structure and Functionality**

**ESP32 Firmware (C++)**

* **Purpose**: Manages connectivity, provisioning, MQTT communication, and OTA updates on an ESP32 with a SIM7600 modem.
* **Key Features**:
  + **State Machine**: Implements a finite state machine (SetupState) to handle modem initialization, network connection, GPRS, certificate upload, SSL setup, MQTT setup/connection/subscription, provisioning, and OTA updates.
  + **MQTT Communication**: Uses the SIM7600’s AT commands for MQTT over SSL/TLS, with topics for status (esp32\_status), commands (server\_cmd), OTA (OTA\_Update), and provisioning (dev\_pass\_req, dev\_pass\_res).
  + **Provisioning**: Requests credentials from the server using ECDH key exchange and stores them encrypted in NVS (Non-Volatile Storage).
  + **OTA Updates**: Supports chunked firmware updates with SHA-256 hash verification and ECDSA signature validation.
  + **Hardware Integration**: Controls RGB LEDs, an optional LCD, and a factory reset pin.
* **Libraries Used**:
  + TinyGsmClient for modem communication.
  + mbedtls for AES encryption, SHA-256 hashing, ECDH, and ECDSA.
  + Adafruit\_NeoPixel for RGB LED control.
  + Preferences for NVS storage.

**Python Server**

* **Purpose**: Acts as the MQTT server to provision devices and deliver OTA updates.
* **Key Features**:
  + **MQTT Client**: Connects to an MQTT broker, subscribes to provisioning and status topics, and publishes responses/OTA data.
  + **Provisioning**: Generates credentials (device ID, username, password) and encrypts them using a shared secret derived via ECDH.
  + **OTA Updates**: Sends firmware in chunks, computes SHA-256 hashes, and signs with ECDSA.
  + **Interactive CLI**: Accepts commands (OTA, reverse old firmware) via a non-blocking input handler.
* **Libraries Used**:
  + paho.mqtt.client for MQTT communication.
  + cryptography for AES, ECDH, ECDSA, and hashing.
  + Threading and queue for concurrent MQTT and input handling.

**Enhanced Security Implementations**

**ESP32 Firmware**

1. **Encryption**:
   * **AES-256-CBC**: Used for encrypting messages and NVS data with dynamically generated keys (aes\_key, device\_key).
   * **PKCS7 Padding**: Ensures proper block alignment for AES encryption.
2. **Key Exchange**:
   * **ECDH (SECP256R1)**: Establishes a shared secret with the server during provisioning, using a client-generated public/private key pair.
3. **Authentication**:
   * **ECDSA (SECP256R1)**: Verifies OTA firmware signatures using the server’s public key.
4. **Secure Storage**:
   * **NVS Encryption**: Credentials (device ID, username, password) are encrypted before storage in NVS using device\_key.
5. **SSL/TLS**:
   * Configures the SIM7600 modem to use SSL/TLS (version 4) with a CA certificate (iot\_inverter2.pem) for MQTT communication.
6. **Randomization**:
   * Uses esp\_random() and esp\_fill\_random() for generating keys, IVs, and UUIDs, enhancing unpredictability.

**Python Server**

1. **Encryption**:
   * **AES-CBC**: Encrypts provisioning credentials using a shared secret derived from ECDH.
   * **Padding**: Implements PKCS7-like padding for AES compatibility.
2. **Key Exchange**:
   * **ECDH (SECP256R1)**: Computes a shared secret with the ESP32’s public key for secure credential delivery.
3. **Authentication**:
   * **ECDSA (SECP256R1)**: Signs firmware with a private key, allowing the ESP32 to verify authenticity.
4. **Secure Key Management**:
   * Stores a persistent private key (server\_private\_key.pem) and generates it only if absent.
5. **TLS**:
   * Configures the MQTT client with a CA certificate (commented out but intended), ensuring secure broker communication.

**Advantages of Current Structure**

**Provisioning**

* **ESP32**:
  + **Dynamic Credentials**: Requests a unique device ID, username, and password from the server, avoiding hardcoded credentials.
  + **ECDH Security**: Ensures credentials are encrypted end-to-end, preventing interception over MQTT.
  + **State Machine**: Robustly handles provisioning timeouts and retries, ensuring reliability even with network issues.
* **Python Server**:
  + **Custom Device IDs**: Generates unique IDs (e.g., ESP32\_abcd1234), improving device management.
  + **Interactive Password**: Allows manual password input, enhancing flexibility and security (if strong passwords are chosen).
  + **Device Tracking**: Logs provisioned devices in devices.json for future reference.

**Advantages**:

* Prevents credential reuse across devices.
* Secures initial setup against eavesdropping via ECDH and AES.
* Allows centralized control and auditing of device credentials.

**OTA Updates**

* **ESP32**:
  + **Chunked Delivery**: Processes firmware in 1024-byte chunks, supporting large updates over constrained networks.
  + **Integrity and Authenticity**: Verifies SHA-256 hash and ECDSA signature, ensuring the firmware is untampered and from a trusted source.
  + **Recovery Mechanisms**: Tracks missing chunks, requests retransmission, and reverts to the previous partition on failure.
* **Python Server**:
  + **Chunked Transmission**: Sends firmware in base64-encoded chunks with sequence numbers, ensuring ordered delivery.
  + **Acknowledgment**: Waits for ESP32 progress updates (OTA:PROGRESS), enabling retransmission of lost chunks.
  + **Signature**: Signs the firmware, providing non-repudiation.

**Advantages**:

* Robust against network interruptions due to chunking and retransmission.
* Ensures only authorized firmware is installed via ECDSA.
* Supports rollback to previous firmware, reducing bricking risk.

**Difficulty of Hacking**

**Potential Attack Vectors**

1. **MQTT Interception**:
   * **Mitigation**: SSL/TLS encrypts MQTT traffic, and provisioning uses ECDH-derived keys. An attacker would need the server’s private key or a man-in-the-middle (MITM) exploit against the CA.
   * **Difficulty**: High (requires CA compromise or private key theft).
2. **Firmware Tampering**:
   * **Mitigation**: ECDSA signature verification ensures only signed firmware is accepted. The attacker needs the server’s private key to forge a signature.
   * **Difficulty**: Very High (private key is not exposed unless server is compromised).
3. **Credential Theft**:
   * **Mitigation**: Credentials are encrypted in NVS and never sent in plaintext. Physical access to the ESP32 and key extraction would be required.
   * **Difficulty**: Moderate to High (requires physical access and advanced hardware skills).
4. **Provisioning Spoofing**:
   * **Mitigation**: ECDH ensures the shared secret is unique per session, and the server’s public key is hardcoded in the ESP32.
   * **Difficulty**: High (requires spoofing the server and breaking ECDH).
5. **Physical Attacks**:
   * **Mitigation**: Factory reset clears credentials, but no tamper detection is implemented.
   * **Difficulty**: Low to Moderate (physical access is straightforward, but exploiting it requires expertise).

**Overall Hacking Difficulty**

* **Rating**: High
* **Reason**: The use of modern cryptographic standards (ECDH, ECDSA, AES-256, TLS) and secure key management makes remote attacks challenging. Physical attacks are the weakest link but still require significant effort to extract keys or bypass verification.

**Draft Status and Testing/Debugging Needs**

**ESP32 Firmware**

* **Draft Indicators**:
  + Hardcoded MQTT credentials (ESP32, 12345) and server details suggest placeholder values.
  + Incomplete error handling (e.g., processURC has duplicate conditions, some functions lack full implementation).
  + OTA rollback (revertToPreviousFirmware) is basic and untested.
* **Testing Needs**:
  + **Modem Commands**: Verify all AT commands (e.g., +CMQTTCONNECT, +CCERTDOWN) work with the SIM7600 under various network conditions.
  + **Provisioning**: Test ECDH key exchange and credential decryption with the server.
  + **OTA**: Simulate network drops, missing chunks, and invalid signatures to ensure robustness.
  + **Debugging**: Add more logging in processURC and state transitions to diagnose failures.

**Python Server**

* **Draft Indicators**:
  + TLS setup is commented out (client.tls\_set(ca\_certs=CA\_CERT)), risking insecure connections.
  + Error handling is minimal (e.g., no retry logic for failed publishes).
  + Hardcoded credentials (ESP32, 12345) and CA path are placeholders.
* **Testing Needs**:
  + **MQTT Connectivity**: Test with the actual broker and TLS enabled.
  + **Provisioning**: Validate ECDH and AES encryption/decryption compatibility with the ESP32.
  + **OTA**: Test with a real firmware file, including edge cases (file corruption, interruptions).
  + **Debugging**: Log full MQTT payloads and add timeouts/retries for chunk acknowledgments.

**Conclusion**

The ESP32 firmware and Python server provide a solid foundation for a secure IoT system with provisioning and OTA capabilities. Enhanced security through ECDH, ECDSA, AES, and TLS makes hacking difficult without significant resources or physical access. The state machine and chunked OTA structure offer reliability and scalability, though the draft nature requires thorough testing and debugging to ensure production readiness. With proper refinement, this system could be highly robust and secure for real-world deployment.