Secure Communication Protocol Design Report

Poornima PUNDIR, Mosè ACARO

1 Introduction

The goal of this protocol is to allow two communicating parties—Alice and Bob—to establish a mutually authenticated shared secret using asymmetric cryptography and then use this secret to operate a secure, encrypted, integrity-protected communication channel.
This report specifies:

1. An Authenticated Key Exchange (AKE) protocol using manually implemented RSA signatures and Diffie–Hellman (DH) key exchange.
2. A symmetric encrypted channel using AES-256-GCM, including integrity, authenticity, and replay protection.

Security is prioritized over availability: any verification failure results in immediate termination.

2 Authenticated Key Exchange Protocol

2.1 Chosen Cryptographic Primitives

- **RSA Signatures:** Used to authenticate Diffie–Hellman values and prove identity. This prevents MITM attacks by binding DH values to identities. RSA is required for the assignment and mirrors authentication patterns in TLS, SSH, and Noise-based protocols.
- **SHA-256:** Hashes handshake components prior to signing. Ensures fixed-size inputs for RSA, provides collision resistance, and prevents algebraic manipulation.
- **Diffie–Hellman Key Exchange:** Used to compute the shared secret $K = g^{ab} \bmod p$. Provides forward secrecy when ephemeral exponents are used and is robust against passive eavesdropping.
- **CA-Signed Certificates (Local CA Trust Anchor):** Each party generates a long-term RSA keypair and obtains a certificate signed by a local CA. The CA certificate is distributed out-of-band and used as the trust anchor during the handshake.
- **Random Nonces ($randA$, $randB$):** Provide freshness and prevent replay of handshake messages.

2.2 Cryptographic Setup

Each party generates:

- An RSA keypair and a CA-signed certificate.
- A DH private exponent (a or b) and corresponding public value ($A = g^a$, $B = g^b$).
- A fresh random nonce ($randA$, $randB$).

Certificates received during the handshake must validate against the trusted CA certificate.

2.3 Protocol Message Sequence

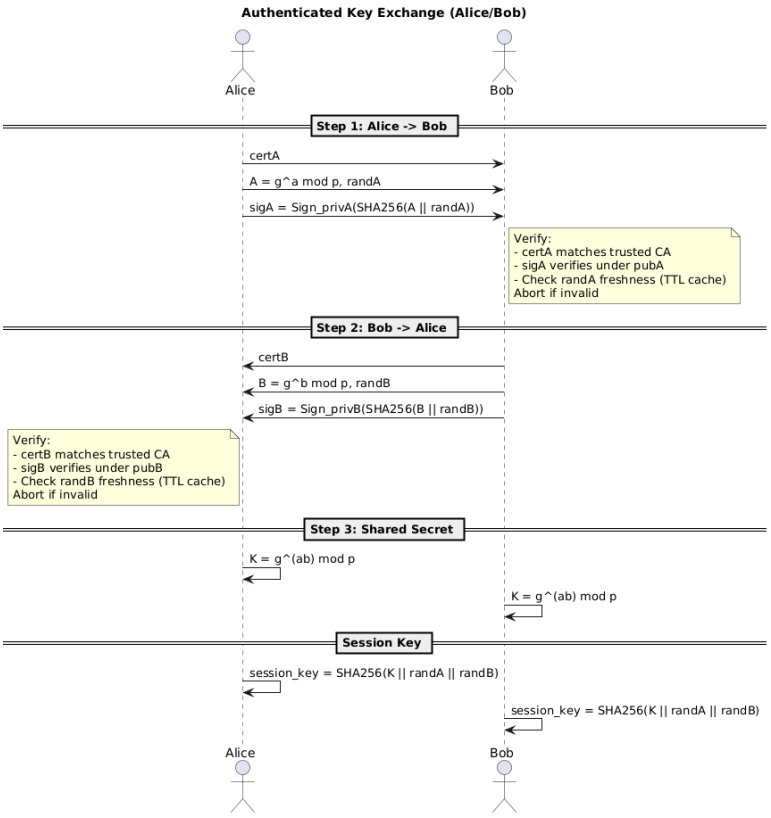

Figure 1: Authenticated Key Exchange Protocol Sequence

Step 1 — Alice → Bob

```
certA
A = g^a mod p
randA
sigA = Sign_privA( SHA256(A || randA) )
```

Bob verifies certA and checks sigA using Alice's public key. If verification fails, the handshake is aborted.

Step 2 — Bob → Alice

```
certB
B = g^b mod p
randB
sigB = Sign_privB( SHA256(B || randB) )
```

Alice verifies certB and checks sigB under Bob's public key. If verification fails, the handshake is aborted.

Step 3 — Diffie–Hellman Shared Secret

Both compute:

$$K = g^{ab} \bmod p$$

Both parties then locally derive the session key from K, $randA$, and $randB$.

2.4 Session Key Derivation

The symmetric session key is derived as:

$$session\_key = \text{SHA256}(K \,||\, randA \,||\, randB)$$

2.5 Known Attacks Prevented by the Asymmetric Protocol

- **MITM attacks:** An attacker cannot modify or replace the Diffie–Hellman values because each DH contribution is individually authenticated with an RSA signature. Alice signs $(A \,\|\, randA)$, and Bob signs $(B \,\|\, randB)$, so any attempt to alter these values causes signature verification to fail. As a result, a man–in–the–middle attacker cannot intercept or modify the key exchange without being detected.
- **Impersonation attacks:** Only the legitimate party possesses the corresponding RSA private key needed to generate valid signatures. Since the CA certificate is distributed out-of-band, attackers cannot substitute their own public keys.
- **Replay attacks:** Fresh random nonces (`randA`, `randB`) are included in all signatures. Replays are mitigated by tracking recently seen nonces within a short time window; replays outside this window may still succeed.
- **Passive eavesdropping:** Diffie–Hellman provides secrecy even if the attacker records every message. Without knowing either party's DH private exponent, the shared secret $K = g^{ab} \bmod p$ cannot be computed.

3 Symmetric Secure Channel Design

3.1 Encryption Scheme

AES-256-GCM provides confidentiality, integrity, authenticity, and replay protection.

3.2 Message Format

```
SeqNum || IV || Ciphertext || AuthTag
```

Sequence number is included as AES-GCM AAD.

3.3 Known Attacks Prevented by This Channel Design

- **Replay Attacks.** Each encrypted message includes a monotonically increasing sequence number transmitted as authenticated associated data (AAD) under AES-GCM. Because the sequence number is integrity-protected and must be strictly increasing, the receiver rejects any repeated messages or attempts to reorder previously valid ciphertexts.
- **Message Modification and Forgery.** AES-GCM provides an integrated authentication tag that covers both the ciphertext and the sequence number. Any bit-level modification to the ciphertext, IV, or AAD results in an authentication failure during GCM tag verification. An adversary lacking the session key cannot generate a valid authentication tag, rendering message forgery computationally infeasible.
- **Reordering Attacks.** Since the receiver enforces strict monotonicity of sequence numbers, ciphertexts delivered out of order are automatically rejected.
- **Forced Connection Termination.** Because session termination requires an authenticated `CLOSE_NOTIFY` and a corresponding authenticated `CLOSE_ACK`, an attacker cannot simulate a clean shutdown. Any attempted interruption appears only as a network failure rather than a legitimate protocol-driven close.
- **Message Injection.** Without knowledge of the session key, an attacker cannot construct a valid ciphertext-tag pair. Since tags are cryptographically inseparable from the encrypted payload, no adversary can inject new messages into the channel that would successfully authenticate at the receiving endpoint.

3.4 Secure Session Termination

Control messages exchanged:

- `CLOSE_NOTIFY`
- `CLOSE_ACK`

Both parties delete all session keys and state.

4 Limitations

- **No DoS protection.** The protocol cannot mitigate flooding or repeated connection attempts.
- **CA trust is out-of-band.** If the CA certificate distribution or private key is compromised, all authentication fails.

- **Endpoint compromise.** If an endpoint is compromised, the attacker gains full control and can impersonate indefinitely.
- **No identity hiding.** Certificates are transmitted in plaintext.
- **Forward secrecy depends on fresh DH.** Security requires fresh DH exponents for every session. Reusing DH keys breaks FS and allows past sessions to be decrypted after RSA key compromise.
- **No traffic analysis protection.** Encrypted messages still leak size and timing, allowing an attacker to infer communication patterns.
- **Raw RSA signatures without padding.** The implementation signs the SHA-256 hash directly with RSA (textbook RSA), which is weaker than standardized padding schemes such as RSASSA-PSS.
- **No specific identity binding.** Only the role is checked; sessions can be associated with the wrong party in multi-party settings.

5 Working Application Example

The protocol implementation was verified using the Python `unittest` framework. A total of 20 automated tests were executed to validate each individual modules and the integrated application logic.

Server Output

```
[INFO] Using cert certs/server.crt...
[INFO] Server listening on 127.0.0.1:65432
[INFO] Connection established...
[INFO] Peer certificate verified...
[INFO] Derived session key using DH.
[DEBUG] Received (encrypted): b'\x00\x00...'
Received: ciao
[DEBUG] Sent: b'\x00\x00...'
[DEBUG] Received (encrypted): b'\x00\x00...'
Received: hello world!
[DEBUG] Sent: b'\x00\x00...'
[DEBUG] Received (encrypted): b'\x00\x00...'
[INFO] Client requested secure close.
```

Client Output

```
[INFO] Using cert certs/client.crt...
[INFO] Connected to server at 127.0.0.1...
[INFO] Peer certificate verified...
[INFO] Derived session key using DH.
Enter message: ciao
[DEBUG] Sent: b'\x00\x00...'
[DEBUG] Server response: b'\x00\x00...'
Server response: ACK
Enter message: hello world!
[DEBUG] Sent: b'\x00\x00...'
[DEBUG] Server response: b'\x00\x00...'
Server response: ACK
Enter message: quit
[INFO] Sending close notification...
[DEBUG] Waiting for Server ACK...
[INFO] Received CLOSE_ACK.
```

6 Conclusion

This protocol implements a secure authenticated key exchange using RSA and Diffie–Hellman. AES-256-GCM provides a confidential and integrity-protected communication channel.