Assignment 3: Integrity

Revised: April 3, 2025

1. Introduction

In Homework 2, we engineered a file transfer system that combined RSA's asymmetric encryption for secure key exchange with AES's efficient encryption of file contents. The result? A functional tunnel that shielded data from prying eyes—*confidentiality achieved*. But as many of you astutely noted in your reports, confidentiality alone isn't enough.

The Gaps in Our Armor

Let's dissect the vulnerabilities lurking in our initial design:

1. **Plaintext Metadata Leaks**: File names like *"Q4_Financials.pdf"* or *"Passwords_Backup.txt"* — traveled unencrypted, offering attackers a treasure map of sensitive data. Metadata matters; leaving it exposed is akin to locking a vault but leaving the combo written on the door.
2. **Not Tamper Resistance**: AES ensured encrypted data stayed private but couldn't guarantee *integrity*. Imagine a sealed envelope being opened, altered, and resealed mid-transit. Without a way to detect tampering, the server blindly trusted whatever arrived.
3. **No Sender Authentication**: How could the server confirm the file came from *you*? Without proof of origin, a malicious actor could impersonate the client—classic man-in-the-middle territory.

Enter Digital Signatures: The Trust Layer

This time, we're evolving our system to address these flaws by integrating **digital signatures**—a cryptographic Swiss Army knife for authenticity and integrity. Here's the game plan:

- **Signing Files**: The client generates a signature using their private key before sending a file. Think of this as a virtual fingerprint mathematically tied to the file's contents and the sender's identity.
- **Verification on Arrival**: The server will validate this signature using the client's public key. If the signature checks out, two things are confirmed:
    - **Authenticity**: The file came from the claimed sender.
    - **Integrity**: The file wasn't altered *a single bit* during transit.

Why This Matters

By layering digital signatures atop our existing encryption, we're transforming our system from "secure-ish" to "trustworthy." Filenames will now be encrypted, signatures will deter tampering, and authentication will shut out impersonators. It's the difference between locking your front door and adding a security camera—both are good, but together, they're *bulletproof*.

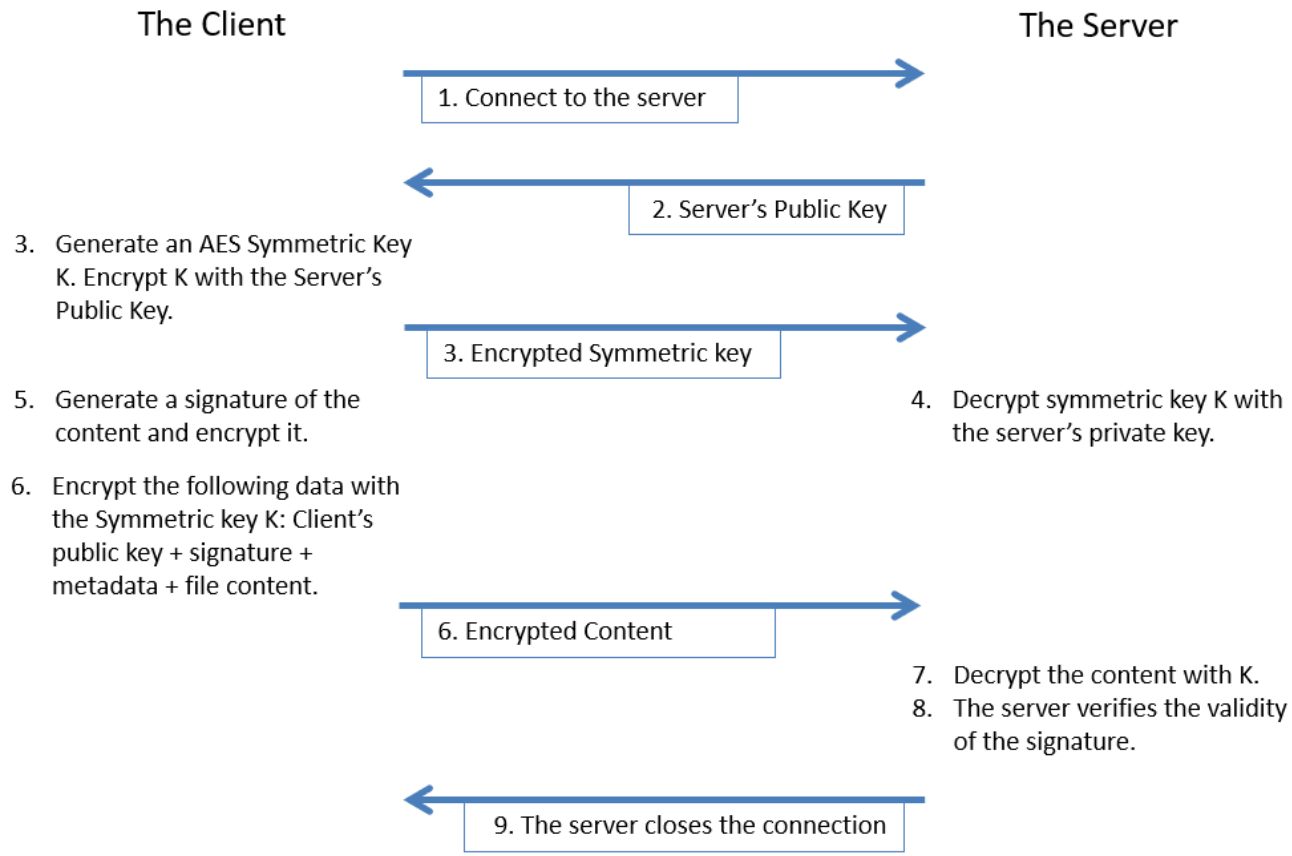
Ready to code this up? Let's dive into the architecture and turn these concepts into Python-powered reality.

Improve our current system.

Before diving into coding, let's outline how digital signatures work and identify the **new components** needed for the system.

Process Overview:

1. **Client-Side (Steps 5–6):**
    - Generate a **hash digest** of the file using a cryptographic hash function.
    - Encrypt this hash digest with the **client's private key** to create the digital signature.
    - Send the original file and the encrypted signature through the existing secure tunnel (secured by RSA and AES).
2. **Server-Side (Steps 7–8):**
    - Recompute the hash digest from the received file using the same hash function.
    - Decrypt the client's signature using the **client's public key** to retrieve the original hash digest.
    - Validate the signature by comparing the recomputed hash with the decrypted hash.



The numbers in the figure above have been revised to correspond to the description.

2. New Components

The following are the new components of this assignment.

- **Cryptographic Hash Function:** To generate/verify digests, the client and server must implement the same hash algorithm (e.g., SHA-256).
- **Client-Side RSA Key Pair:**
    - The client must generate its own **private/public key pair** (previously, only the server might have had keys).
    - A **client RSA key generation function** is required (not just server-side).
- **Signature Verification Logic on Server:** This is a mechanism for decrypting the signature with the client's public key and validating it.

Complete the missing sections in the provided template code files following the same approach as in the previous assignment. Note that we've inserted a user input prompt

between steps 7 and 8. This prompt lets you edit the received file stored in the server's Upload folder. Modifying the file's content will fail the subsequent signature verification. If you leave the file unchanged, the verification should succeed as expected.

3. Deliverables

1. **Python Codes**: Submit the completed Python files (client3.py and server3.py). You should adequately document the code and print out messages so the TA can easily see what happens during the execution. The TAs will test your code using their test files. Ensure your code includes additional print statements (after each significant step) to help the TAs easily trace the execution flow.
2. **Report in PDF**: Include the following screenshots as part of your submission:
    - We made many assumptions on both sides of the connection for the client-server program to work as designed. Please summarize all of them. You may have to study the programs to collect all of them.
    - Please list all the authentication and integrity measures we have achieved in the code. Explain how they were accomplished.
    - Attach the test files:
        1. Original text file, along with a successful signature verification.
        2. Modified text file**,** along with a failed signature verification.