Implementation of a Simple RSA-Based Digital Signature Scheme

Burak EVREN
Department of Computer Engineering
burakevren.26@gmail.com

***Abstract*—This report details the design and implementation of a secure messaging application that utilizes the RSA algorithm for end-to-end encryption and digital signatures. The core objective is to ensure message confidentiality, integrity, authenticity, and non-repudiation. The application demonstrates RSA key generation, hashing using SHA-256, message signing with PKCS#1 v1.5, message encryption with OAEP(Optimal Asymmetric Encryption Padding), and a basic certificate generation and verification mechanism. A man-in-the-middle (MitM) attacker simulation is also implemented to practically demonstrate the security features and the importance of cryptographic protections in a chat environment. The system is built using Python, Flask, SocketIO, and the PyCryptodome library for cryptographic operations.**

***Index Terms*—RSA, Digital Signature, Encryption, PKCS#1, OAEP, SHA-256, Secure Messaging, Man-in-the-Middle Attack.**

I. Introduction

In an increasingly interconnected digital world, the security of communications is highly important. Ensuring that messages are confidential, that their integrity is maintained, and that the identity of the sender can be authenticated are fundamental requirements for trustworthy digital interactions.[1] This project addresses these challenges by implementing a secure messaging application that leverages the RSA algorithm [2] for robust cryptographic protections.

The primary goal of this project is to practically demonstrate the application of RSA-based digital signatures and encryption in a real-time chat system. This involves generating RSA key pairs, signing messages to ensure authenticity and integrity, encrypting messages for confidentiality, and implementing a basic certificate mechanism to associate public keys with user identities. The scope of this study includes the development of the chat application itself, the integration of cryptographic functionalities using the PyCryptodome library [3], and the creation of a man-in-the-middle (MitM) attacker script to simulate threats and showcase the effectiveness of the implemented security measures. This hands-on approach aims to provide a clear understanding of how theoretical cryptographic concepts translate into practical security solutions.

II. Background and Related Work

The security of the developed messaging application is built upon established cryptographic principles and standards.

A. Asymmetric Cryptography and RSA

Asymmetric cryptography, also known as public-key cryptography, uses a pair of mathematically related keys: a public key, which can be shared openly, and a private key, which must be kept secret by the owner.[4] The RSA algorithm, developed by Rivest, Shamir, and Adleman, is a widely adopted asymmetric algorithm whose security relies on the computational difficulty of factoring large prime numbers.[2] In this project, RSA is used for both encrypting messages and generating digital signatures.

B. Digital Signatures

A digital signature provides assurance of message authenticity (the message was sent by the claimed sender), integrity (the message was not altered in transit), and non-repudiation (the sender cannot deny sending the message) [5] [6] This is typically achieved by hashing the message and then encrypting the hash with the sender's private key. The application uses the PKCS#1 v1.5 signature scheme [7] for this purpose.

C. Message Encryption

To ensure confidentiality, messages can be encrypted so that only the intended recipient (who possesses the corresponding private key) can decrypt and read them. "Textbook RSA" encryption is vulnerable to certain attacks, so padding schemes are essential.[4] This project employs OAEP [7], which adds randomness and structure to the message before RSA encryption, enhancing security.

D. Hash Functions (SHA-256)

Cryptographic hash functions generate a fixed-size digest from an input message. Secure hash functions like SHA-256 are designed to be one-way and collision-resistant.[8] In this application, SHA-256 is used to create message digests before signing, as part of the OAEP padding process, and for general integrity checks.

E. Certificates and Trust

To link a public key to a specific user identity, digital certificates are used. While this project implements a simplified certificate generation and verification mechanism managed by the application itself (acting as a local CA), in real-world scenarios, trusted third-party Certificate Authorities (CAs) issue and manage X.509 certificates [9] [10]

F. Threat Model: Man-in-the-Middle (MitM) Attack

The application's security is tested against a simulated Man-in-the-Middle (MitM) attack. A MitM attacker attempts to intercept and potentially alter communication between two parties without their knowledge.[11] The 'attacker.py' script in this project demonstrates how an attacker might try to eavesdrop or tamper with messages, and how the implemented cryptographic measures (signatures and encryption) can thwart or detect such attempts.

III. Proposed Methodology

The methodology focuses on integrating RSA-based cryptographic operations into a web-based chat application.

A. Design Goals

The primary design goals for the application were:

- **Confidentiality**: Ensure messages, when encrypted, are only readable by the intended recipient.
- **Authenticity**: Allow users to verify the origin of messages.
- **Integrity**: Ensure messages cannot be altered in transit without detection.
- **User-Friendliness**: Provide a simple interface for users to chat and observe security features.
- **Demonstration**: Clearly illustrate the impact of encryption and digital signatures, especially in the context of a MitM attack.

B. Threat Model

The application considers an active MitM attacker, as implemented in 'attacker.py'. This attacker can:

- Intercept all messages passing through the server.
- Read unencrypted messages.
- Modify unencrypted messages before forwarding them.
- Attempt to modify encrypted or signed messages (which should be detected).

The attacker cannot, however, derive private keys or break the underlying RSA cryptography with the given key sizes (2048).

C. Construction and Attack Strategy

The application's core cryptographic functionalities are implemented in 'rsa_utils.py' using the PyCryptodome library.[3]

1. Key Generation:
Upon user registration in 'app.py', a 2048-bit RSA key pair (public and private) is generated for each user using 'rsa_utils.generate_key_pair()'. This function internally calls 'RSA.generate(2048)' from PyCryptodome. The 2048-bit key size is chosen in line with NIST recommendations for security strength of 112 bits.[12]

2. Message Signing (PKCS#1 v1.5):
When a user sends a message with the "End-to-End Encryption" option enabled, the message content is first hashed using SHA-256. The 'rsa_utils.sign_message(message, private_key)' function then uses the sender's private key to sign this hash. This is implemented using 'pkcs1_15.new(private_key).sign(h)' from PyCryptodome [7] [13] PKCS#1 v1.5 defines a standard way to format the hash before applying the RSA private-key operation.

3. Signature Verification:
The recipient verifies the signature using 'rsa_utils.verify_signature(message, signature, public_key)'. This function re-hashes the received message and uses the sender's public key (obtained from their certificate) with 'pkcs1_15.new(public_key).verify(h, signature)' [7] [13] A successful verification confirms authenticity and integrity.

4. Message Encryption (OAEP):
If encryption is enabled, the plaintext message is encrypted using the recipient's public key via 'rsa_utils.encrypt_message(message, public_key)'. This function employs 'PKCS1_OAEP.new(public_key).encrypt(message)' from PyCryptodome.[7] OAEP is a padding scheme that enhances RSA's security by adding randomness and structure, making it resilient against certain attacks that affect "textbook" RSA.[14] It uses a hash function (SHA-256 in this application) and a Mask Generation Function (MGF1, also based on SHA-256) internally to prepare the message for RSA encryption [7] [15] This makes the encryption probabilistic.

5. Message Decryption:
The recipient decrypts the message using their private key with 'rsa_utils.decrypt_message(encrypted_message, private_key)', which calls 'PKCS1_OAEP.new(private_key).decrypt(encrypted_message)'. The OAEP unpadding process also inherently checks the integrity of the padding.

6. Certificate Management:
The application implements a basic certificate authority. When a user registers, 'app.py' calls 'rsa_utils.generate_certificate(user_id, public_key, ca_private_key)'. This creates a simple certificate structure containing the user's ID and public key, signed by a local "CA" private key held by the listening server. These certificates are sent to clients and used to retrieve other users' public keys. Verification is done via 'rsa_utils.verify_certificate(certificate, ca_public_key)'. This simulates a PKI environment for trust.

7. Integration in Chat Application (‘app.py‘, ‘app.js‘):
The Flask server (‘app.py‘) handles user connections, message relay, and cryptographic operations via SocketIO. The JavaScript frontend (‘app.js‘) allows users to log in, select recipients, type messages, toggle encryption/signing, and displays received messages along with their security status (e.g., ”verified,” ”tampered”).

8. Attacker Simulation (‘attacker.py‘):
The ‘attacker.py‘ script connects to the server as a separate client. When its ”default” is active, it intercepts messages logged by ‘message_handler.py‘. It can display unencrypted messages and if ”tampering” mode is on it offers the operator an option to modify them before they are forwarded (via the server) to the intended recipient. This demonstrates how unsigned/unencrypted messages are vulnerable and how signed/encrypted messages resist or reveal tampering.

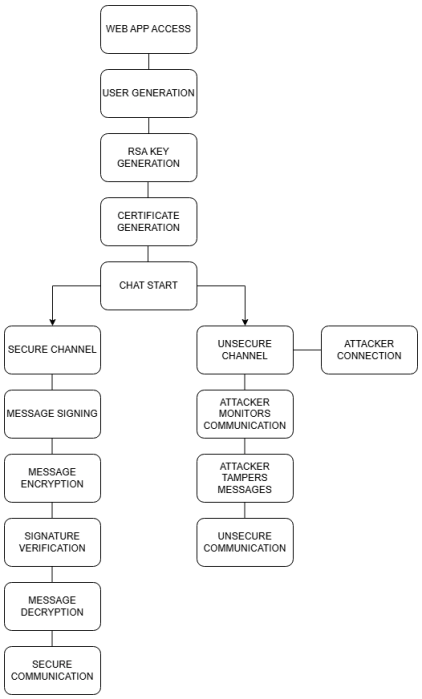

Fig. 1: Flowchart of proposed application

D. Evaluation Metrics

The project’s success is evaluated by:

- Correct generation and use of RSA key pairs.
- Successful signing and verification of messages, demonstrating authenticity and integrity.
- Successful encryption and decryption of messages, demonstrating confidentiality.
- The chat application’s ability to clearly indicate the security status of messages.
- The attacker simulation’s effectiveness in highlighting the vulnerabilities of unsecured communication and the protections offered by the implemented cryptographic measures.

E. Experimental Setup

- **Language**: Python 3.x
- **Libraries**:
  - PyCryptodome [3]: For RSA operations (key generation, PKCS#1 v1.5 signing, OAEP encryption), SHA-256 hashing.
  - Flask: For the web server backend.
  - Flask-SocketIO: For real-time bidirectional communication between clients and server.
- **Frontend**: HTML, CSS, JavaScript.
- **Environment**: Personal computer.

IV. Security Evaluation

The application’s security relies on the established strengths of the chosen cryptographic primitives:

- **RSA (2048-bit keys)**: Provides strong protection against brute-force attacks for factoring the modulus, ensuring the private keys remain secure.[12]
- **SHA-256**: Offers robust collision resistance, making it computationally infeasible to find two different messages that hash to the same value, which is crucial for signature integrity.[8]
- **PKCS#1 v1.5**: A standard signature scheme that, when used with a strong hash function and adequate key length, provides message authenticity and integrity.[7] The attacker simulation demonstrates that if a signed message is tampered with by ‘attacker.py‘, the signature verification on the recipient’s side will fail, alerting the user.
- **OAEP**: Enhances RSA encryption by adding randomness, making it semantically secure and resistant to certain plaintext attacks.[14] It also provides a degree of integrity checking for the encrypted data. If the attacker modifies an OAEP-encrypted ciphertext, decryption will likely fail.
- **Certificates**: The basic certificate mechanism, while not a full PKI, allows users to obtain public keys of other users in a way that is signed by the application’s ”CA.” This helps in authenticating the source of public keys within the application’s context.

The ‘attacker.py‘ script serves as a practical evaluation tool. When tampering is active:

- Unencrypted, unsigned messages can be read and modified by the attacker, and the recipient receives the tampered message unknowingly.
- Signed messages, if tampered, will fail verification, and ‘app.js‘ will display a warning.
- Encrypted messages cannot be read by the attacker. If the attacker attempts to modify the ciphertext of an OAEP-encrypted message, it will likely fail decryption or result in garbled plaintext.

This effectively demonstrates the value of the cryptographic protections implemented.

V. Implementation

The project is structured into a backend server, frontend client interface, and an attacker simulation script. Detailed code explanation is not in scope of this report but needed explanation about used functions are already given in "Proposed Methodology" part.

Directory Layout

```
app/
   assets/
      lock-icon.svg
   backend/
      message_handler.py
   components/
      index.html
   crypto/
      rsa_utils.py
   static/
      css/style.css
      js/app.js
   templates/
      index.html
   utils/
       logger.py
attacker.py
app.py
requirements.txt
```

VI. Personal Contributions

Burak EVREN is solely responsible for all aspects of this project, including the conceptualization, literature review, design, implementation of the chat application and cryptographic functionalities, development of the attacker simulation, testing, and the preparation of this final report.

VII. Discussion and Conclusion

This project successfully demonstrated the implementation of RSA-based digital signatures and encryption within a functional chat application. The core objectives of providing message confidentiality, integrity, and authenticity were achieved using PKCS#1 v1.5 for signatures and OAEP for encryption, with SHA-256 as the underlying hash function. The inclusion of a simplified certificate mechanism helped in managing public keys within the application's context.

The man-in-the-middle attacker simulation proved to be a valuable component, vividly illustrating the vulnerabilities of unencrypted and unsigned communications and, conversely, the robustness provided by the implemented cryptographic measures. Users can observe firsthand how signed messages, if tampered, lead to verification failures, and how encrypted messages remain confidential from the attacker.

A. Limitations

The current implementation has certain limitations inherent in a student project scope:

- **Simplified CA**: The application acts as its own CA, which is not secure for a real-world scenario. A proper PKI with trusted third-party CAs would be required.
- **Key Management**: Private keys are generated and managed by the server for simplicity in this demonstration. In a secure system, private keys should be generated and stored securely on the client-side.
- **No Forward Secrecy**: The RSA-based key exchange for encryption does not provide forward secrecy.
- **Basic UI/UX**: The focus was on cryptographic functionality rather than a polished user experience.

B. Future Work

Potential enhancements could include:

- Implementing a more robust key exchange mechanism (e.g., Diffie-Hellman) to establish session keys for symmetric encryption (hybrid encryption), which is more efficient for large messages.
- Exploring client-side private key storage.
- Integrating with a more formal PKI or using self-signed certificates with a clear trust model.
- Enhancing the attacker simulation with more sophisticated attack vectors.

In conclusion, this project provides a practical and educational insight into applying fundamental cryptographic techniques to secure digital communications. It underscores the importance of digital signatures and encryption in protecting against common threats and building trust in online interactions.

VIII. Ethical Statement

I, Burak EVREN, affirm that this project and the accompanying report are my original work. All sources used have been appropriately acknowledged. The cryptographic tools and techniques have been implemented for educational and demonstrative purposes to understand security principles and not for any malicious intent.

IX. AI Tools Usage

Gemini, a large language model, was used to assist in developing project idea and conducting necessary research using "Deep Research" function. It also used to generate README.md file to free me from writing and formatting myself.

References

[1] Forouzan, B. A. (2008). *Introduction to cryptography and network security*. McGraw-Hill.

[2] Rivest, R. L., Shamir, A., & Adleman, L. (1978). A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM, 21*(2), 120–126. https://doi.org/10.1145/359340.359342

[3] Eijs, H. (2025). *PyCryptodome* (Version 3.23.0) [Computer software]. https://www.pycryptodome.org

[4] Christof Paar and Jan Pelzl., *Understanding Cryptography: A Textbook for Students and Practitioners (1st. ed.)*. Springer Publishing Company, Incorporated, 2009. Available: https://gnanavelrec.wordpress.com/wp-content/uploads/2019/06/2.understanding-cryptography-by-christof-paar-.pdf

[5] National Institute of Standards and Technology. (2013). *Digital Signature Standard (DSS)* (FIPS PUB 186-4). U.S. Department of Commerce. https://doi.org/10.6028/NIST.FIPS.186-4

[6] Stallings, W. (2017). *Cryptography and network security: Principles and practice* (7th ed.). Pearson.

[7] Moriarty, K., Kaliski, B., Jonsson, J., & Rusch, A. (2016). *PKCS #1: RSA Cryptography Specifications Version 2.2* (RFC 8017). Internet Engineering Task Force. https://doi.org/10.17487/RFC8017

[8] National Institute of Standards and Technology. (2015). *Secure Hash Standard (SHS)* (FIPS PUB 180-4). U.S. Department of Commerce. https://doi.org/10.6028/NIST.FIPS.180-4

[9] International Telecommunication Union. (2019). *Information technology – Open Systems Interconnection – The Directory: Public-key and attribute certificate frameworks* (ITU-T Recommendation X.509). https://www.itu.int/rec/T-REC-X.509-201910-I/en

[10] Adams, C., & Lloyd, S. (2003). *Understanding Public-Key Infrastructure: Concepts, Standards, and Deployment Considerations* (2nd ed.). Addison-Wesley Professional.

[11] CrowdStrike. (2025, January 17). *What is a man-in-the-middle (MITM) attack?* https://www.crowdstrike.com/en-us/cybersecurity-101/cyberattacks/man-in-the-middle-mitm-attack/

[12] National Institute of Standards and Technology. (2020). *Recommendation for Key Management, Part 1: General* (NIST Special Publication 800-57 Part 1 Revision 5). U.S. Department of Commerce. https://csrc.nist.gov/pubs/sp/800/57/pt1/r5/final

[13] PyCryptodome Developers. (n.d.). *PKCS#1 v1.5 (RSA)*. PyCryptodome Documentation. Retrieved May 18, 2025, from https://pycryptodome.readthedocs.io/en/latest/src/signature/pkcs1_v1_5.html

[14] Bellare, M., & Rogaway, P. (1995). Optimal Asymmetric Encryption – How to encrypt with RSA. In A. De Santis (Ed.), *Advances in Cryptology — EUROCRYPT '94* (Vol. 950, pp. 92-111). Springer. https://doi.org/10.1007/BFb0053428

[15] PyCryptodome Developers. (n.d.). *PKCS#1 OAEP (RSA)*. PyCryptodome Documentation. Retrieved May 18, 2025, from https://pycryptodome.readthedocs.io/en/latest/src/cipher/oaep.html