Report of the implementation of the RSA-PSS Signature Algorithm

08.03.2024

1 Briefing of the Background and Problem-Statement

1.1 About RSA-PSS Algorithm:

RSA-PSS (Probabilistic Signature Scheme) is a padding scheme for RSA signatures standardized in RFC 8017. It is capable of providing enhanced security and resistance against various cryptographic attacks.

1.2 Problem-Statement:

The task is to implement the RSA-PSS signature algorithm according to RFC 8017 specifications. The implementation should target a security level of 128 bits with an RSA modulus size of 3072 bits. Additionally, SHA-256 should be used as the hash function, and a salt length of 32 bytes is specified.

2 Components and Theory

2.1 RSA Algorithm:

The RSA (Rivest-Shamir-Adleman) algorithm, an asymmetric cryptographic scheme, is widely used for secure data transmission and digital signatures. It is structured upon the mathematical properties of prime numbers and modular arithmetic for its security. Generation of key involves selecting large prime numbers and computing public and private key pairs. Encryption and decryption methods use modular exponentiation. Signing and verification uses private and public keys to create and validate digital signatures. RSA's security has a computational difficulty of factoring large composite numbers, and it is widely employed in secure communication protocols such as SSL/TLS and SSH.

2.2 RSA-PSS Signature Scheme:

The RSA-PSS (Probabilistic Signature Scheme) is a digital signature scheme that incorporates probabilistic padding enhancing the security of RSA signatures. It includes the following components:

1. **Probabilistic Padding:** RSA-PSS prevent chosen plaintext attacks through randomized padding and It's capable of ensuring unpredictability in the resulting signatures.
2. **Salt Generation:** In order to enhance security and prevent production of identical signatures from identical messages, a random salt is generated and included in the padding scheme.
3. **Hash Function (e.g., SHA-256):** RSA-PSS employs a cryptographic hash function, such as SHA-256, to compute a fixed-length hash value of the message. This hash value is then used in the padding scheme.
4. **Mask Generation Function (MGF1):** MGF1 generates a mask from the hash value, which is then applied to the padded message for randomness enhancing the security.

2.3 Security Considerations:

Choice of appropriate parameters is important to achieve the desired security level in cryptographic systems. This includes considerations as follows:

1. **Key Size:** The size of the RSA modulus directly impacts the security of the system. A larger key size, such as 3072 bits, provides stronger security against brute-force attacks and advances in computational power.

2. **Hash Functions:**It is essential to select a secure hash function, such as SHA-256, to generate cryptographic hashes with strong collision resistance. This ensures the integrity of the message and prevents attackers from forging signatures.

3. **Salt Length:** A longer salt length, such as 32 bytes, increases the randomness and uniqueness, enhancing the security of the RSA-PSS scheme. It eliminates the risk of collision attacks and ensures the production of distinct signatures through identical messages.

3 Implementation Strategy:

3.1 Key Generation:

1. **Utilize Secure Methods:** Ensure that the key generation process uses cryptographically secure random number generators to generate prime numbers for the RSA modulus (N) and the public exponent (e). This prevents adversaries from predicting or guessing the generated keys.

2. **Randomness:** Guarantee randomness in the selection of prime numbers to avoid biases or patterns that could compromise the security of the RSA key pair. Randomness ensures that each key pair is unique and unpredictable, enhancing the overall security of the cryptosystem.

3.2 Signing:

1. **Message Encoding:** Encode the message to be signed using a secure encoding scheme to prevent data corruption or manipulation during transmission. This encoding ensures that the message maintains its integrity throughout the signing process.

2. **Salt Generation:** Generate a random salt value for each signature to introduce randomness and uniqueness into the signature generation process. The salt helps prevent attacks such as hash collisions and enhances the security of the signature.

3. **Padding:** Apply the RSA-PSS padding scheme to the message before signing to ensure that the message length is compatible with the RSA modular exponentiation operation. Padding adds additional bits to the message, enhancing its security against cryptographic attacks.

4. **Modular Exponentiation:** Compute the RSA signature using modular exponentiation with the private key. This operation involves raising the padded message to the private exponent modulo the RSA modulus, resulting in the generation of the signature.

3.3 Verification:

1. **Message Decoding:** Decode the signed message and extract the original message content to verify its integrity and authenticity. Proper decoding ensures that the message remains unchanged since it was signed.

2. **Salt Extraction:**Extract the salt value from the signature to facilitate the verification process. The salt is crucial for recalculating the hash of the message during verification and ensures that the signature remains valid.

3. **Padding Verification:** Verify the padding of the signed message to ensure that it maintains the RSA-PSS padding scheme. Proper padding verification guarantees that the message was signed using the correct scheme preventing padding-related attacks.

4. **Modular Exponentiation:** Modular exponentiation is performed with the public key for verification of the signature. This operation involves raising the signature to the public exponent modulo the RSA modulus, resulting in the original message hash. If the computed hash matches the expected value, the signature is deemed valid.

4 Code Development

4.1 Algorithm Implementation:

4.1.1 Key Generation:

- Input: Desired key size in bits (default: 3072)
- Output: RSA key pair (public key, private key)
- Steps:
  1. Generate two large prime numbers, p and q, each of size bits/2 bits.
  2. Compute the modulus n = p * q.
  3. Compute Euler's totient function, phi(n) = (p-1) * (q-1).
  4. Choose a public exponent e (typically 65537).
  5. Compute the private exponent d such that e * d 1 (mod phi(n)).
  6. Construct the RSA public key (n, e) and the RSA private key (n, d).

4.1.2 RSA-PSS Signature Generation:

- Input: Private key, message to be signed
- Output: RSA-PSS signature
- Steps:
  1. Hash the message using SHA-256 to obtain the message digest **'h'**.
  2. Apply the PSS padding scheme to the message digest **'h'**.
  3. Generate a random salt of length 32 bytes.
  4. Apply the MGF1 mask generation function to the salt.
  5. Concatenate the salt with the padded message digest.
  6. Apply modular exponentiation to the concatenated value using the private key to obtain the RSA-PSS signature.

4.1.3 RSA-PSS Signature Verification:

- Input: Public key, message, signature
- Output: Boolean indicating the validity of the signature
- Steps:
  1. Hash the message using SHA-256 to obtain the message digest **'h'**.
  2. Apply the PSS padding scheme to the message digest **'h'**.
  3. Extract the salt from the padded message digest.
  4. Apply the MGF1 mask generation function to the extracted salt.
  5. Verify if the computed maskedDB matches the padded message digest.
  6. Apply modular exponentiation to the signature using the public key to obtain the recovered message digest.
  7. Verify if the recovered message digest matches the original message digest.

4.1.4 JSON to Cookie Conversion:

- Input: JSON data
- Output: Cookie-friendly string
- Steps:
  1. Encode the JSON data into bytes.
  2. Base64-encode the bytes.
  3. Replace characters that are not compatible with cookies (such as '+', '/', and '=') with alternatives.

4.1.5 Main Function:

- Input: Base URL of the server
- Output: Quote received from the server
- Steps:
  1. Generate RSA key pair using **"generate_rsa_key_pair()"** function.
  2. Prepare the message to be signed (e.g., "Your grade is 12").
  3. Sign the message using RSA-PSS algorithm with the private key.
  4. Convert the message and signature to JSON format.
  5. Convert the message and signature to JSON format.
  6. Convert the JSON data to a cookie-friendly string.
  7. Send a request to the server to obtain a quote using the cookie.
  8. Print the received quote..

4.2 Error Handling:

The code deals with robust error handling mechanisms for addressing edge cases, invalid inputs, and potential vulnerabilities. This includes checking for proper input arguments, handling exceptions during key generation, signing, and verification processes, and providing informative error messages to guide users in case of failures.

4.3 Documentation:

Provide clear documentation and comments within the code for a better readability and maintainability purpose.

5 Testing and Validation

5.1 Test Plan:

The test plan aims to thoroughly assess the various features(e.g: functionality, robustness, and security) of the RSA-PSS signature generation and verification system. It covers various aspects such as positive tests to validate correct behavior, negative tests to handle invalid inputs, boundary tests to examine edge cases, interoperability tests to ensure compatibility with other systems, and exception handling to address potential vulnerabilities. Also, Integration testing will verify the interaction between different components, while regression testing ensures stability(no introduction of new issues) through any modifications. Performance testing examines the system's efficiency under various workloads. Overall, the reliability and effectiveness of the RSA-PSS implementation through this comprehensive test plan is ensured.

5.2 Validation:

Validation: Execute the implemented code against the test plan to validate correctness, functionality, and compliance with RFC 8017 standards.

5.3 Performance Evaluation:

Assess the performance of the implementation in terms of execution time, memory usage, and scalability.

6 Conclusion

6.1 Summary:

The implemented RSA-PSS signature algorithm follows the specifications outlined in RFC8017, utilizing SHA-256 as the hash function and mask generation function, with a salt length of 32 bytes. Key generation generates RSA key pairs with a size of 3072 bits ensuring security, employing secure methods to ensure randomness and avoid common pitfalls in random number generation. The signing process implements the RSA-PSS scheme, including message encoding, salt generation, padding, and modular exponentiation, while verification validates signatures using RSA-PSS, including message decoding, salt extraction, padding verification, and modular exponentiation. Error handling mechanisms are implemented to handle edge cases, invalid inputs, and potential vulnerabilities.

The implemented code effectively achieves the objectives of the assignment, providing a secure and efficient RSA-PSS signature algorithm. However, challenges were encountered in ensuring compatibility with the given requirements while avoiding reliance on external libraries. Implementing the algorithm using low-level building blocks required careful consideration of cryptographic principles and algorithmic intricacies. Through this process, valuable lessons were learned about cryptographic protocols, secure coding practices, and the importance of thorough testing.

Overall, the implemented RSA-PSS signature algorithm represents a successful endeavor in cryptographic engineering, demonstrating proficiency in key generation, signing, and verification operations while adhering to security standards and best practices.

6.2 Future Directions

- **Performance Optimization:** Careful observation of techniques to optimize the performance of key generation, signing, and verification operations(such as parallelization, algorithmic optimizations, or hardware acceleration).
- **Enhanced Error Handling:** Strengthen error handling mechanisms to provide more informative error messages. Improved resilience against malicious inputs, and ensure robustness in adverse conditions.
- **Security Hardening:** Conduct a comprehensive security check for identification and addressing potential vulnerabilities, implementing additional security features such as secure key storage, and stay updated with the latest cryptographic recommendations.
- **Usability Enhancements:** Enhanced user experience by developing user-friendly interfaces, providing clear documentation, and integrating the RSA-PSS implementation into existing cryptographic libraries or frameworks.
- **Compatibility and Interoperability:** Ensure compatibility with various platforms, programming languages, and cryptographic standards to facilitate interoperability and integration with other systems seamlessly.
- **Research and Innovation:** Stay updated of advancements in cryptography research, explore new cryptographic primitives or protocols, and contribute to the development of innovative solutions for secure communication and data protection.

References

- StackOverFlow
- medium
- I consulted ChatGPT to clarify theoretical concepts related to RSA-PSS signature scheme implementation.