Methodology for Setting Up End-to-end Encryption for Cross-platform Applications with Java, Javascript, and Flutter

A Study and Application

Xuan Truong Tran
Technical Pedagogical Institute
Hanoi University of Science and Technology
Hanoi, VietNam
tran@travis.guru

ABSTRACT

In this paper, we present a method for setting up end-to-end encryption (E2EE) for cross-platform applications using the programming languages Java, JavaScript, and Flutter. The research details how these languages are applied in the construction and testing of the E2EE system, as well as evaluating the system's performance and security. Through this research, we expand understanding about the application of E2EE in cross-platform applications, contributing to enhancing the security capabilities for modern information systems.

CCS CONCEPTS

• Security and privacy → Cryptography → Cryptographic primitives • Security and privacy → Cryptography → Key management • Applied computing → Computers in other domains → Personal computers and PC applications → Mobile devices • Software and its engineering → Software organization and properties → Software system structures → Software architectures → Cross-platform development

KEYWORDS

software engineering, security, end-to-end encryption, cross-platform

ACM Reference format:

Xuan Truong Tran. 2023. Methodology for Setting Up End-to-end Encryption for Cross-platform Applications with Java, Javascript, and Flutter: A Study and Application. In *The 12th International Symposium on Information and Communication Technology (SoICT 2023), December 7–8, 2023, Ho Chi Minh City, Vietnam. ACM, New York, NY, USA, 5 pages.* https://doi.org/10.1145/1234567890

SoICT 2023, December 7–8, 2023, Ho Chi Minh City, Vietnam

978-1-4503-0000-0/18/06...$15.00
https://doi.org/10.1145/1234567890

1 Introduction

In the current digital age, the need for information security is increasingly prioritized. End-to-End Encryption (E2EE) plays a crucial role as a solution to protect data from threats and undesired intrusions. This paper focuses on analyzing and presenting the method of setting up E2EE using three popular and powerful programming languages: Java, Flutter, and JavaScript.

Java, a powerful programming language, has been widely recognized and used in numerous enterprise applications and large systems. Flutter, a framework from Google, allows the development of mobile applications for both iOS and Android from a single source code. JavaScript, one of the most popular programming languages in the world, plays a pivotal role in developing web applications.

Using these three languages as a basis, we have succeeded in building an E2EE system for cross-platform applications, allowing data encryption at the sender's end and decryption at the designated receiver's end across various types of devices. This method not only enhances data security but also ensures confidentiality and privacy for users.

This paper will detail the process of setting up the E2EE system using Java, Flutter, and JavaScript, from system design, selection, and use of encryption algorithms to system testing and validation. We will also discuss the challenges encountered during the implementation and provide solutions to address them.

2 Theoretical Background

2.1 End-to-End Encryption (E2EE)

End-to-End Encryption (E2EE) is a security method in which only the communicating parties can access and understand the information. The data is encrypted at the sender's end and can only be decrypted by the designated receiver. This process ensures that no one else can read or alter the information, including Internet service providers, communication service providers, or any attackers who might be monitoring the network.

2.2 AES Block Encryption

AES (Advanced Encryption Standard) is a widely recognized data encryption standard. It is a block encryption algorithm, meaning it encrypts data in fixed blocks (128 bits in the case of AES). AES is chosen for actual data encryption in E2EE because it provides a good balance between security and performance: it is very hard to break but still very fast to encrypt and decrypt.

2.3 RSA Symmetric Encryption

RSA (Rivest–Shamir–Adleman) is one of the most popular public key encryption algorithms. It is widely used in creating digital signatures and in encrypting data. In E2EE, RSA is typically used to encrypt the random AES key (used to encrypt the actual data), as RSA allows encryption with a public key and decryption with a private key.

By combining RSA and AES, we can create a robust E2EE system: RSA allows us to securely transfer the AES key from the sender to the receiver, while AES enables us to efficiently encrypt and decrypt the data.

3 Methods

3.1 Sequence Diagram

The following sequence diagram details the communication process between the client and server when utilizing E2EE (full size: https://github.com/travistran1989/methodology-for-setting-up-e2ee-for-cross-platform-applications-with-java-javascript-and-flutter/blob/454f70d5b882401d790ef7e21f93a383103f9580/E2EE_sequence_diagram_en.png?raw=true):

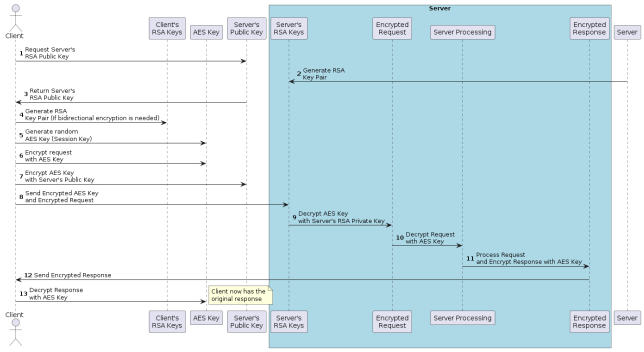

Figure 1: Sequence Diagram

The diagram is divided into six main steps:

1. **RSA Key Pair Generation:** The client requests the server to initiate secure transactions. The server generates an RSA key pair (public and private keys) and returns the public key to the client. The client can also generate its own RSA key pair if two-way encryption is needed. The RSA private key generated by the server is encrypted with the client's JWT Token (representing the client's current session) before being stored in the server's database.
2. **Encryption and Request Sending:** The client generates a random AES key for this session or message, encrypts the data with this AES key, then encrypts the AES key with the server's public key, and sends both to the server.
3. **Server Decryption of the Request:** The server uses its own RSA private key to decrypt the AES key and then uses this AES key to decrypt the original data.
4. **Server Processes the Request and Encrypts the Response:** The server processes the request and generates a response that is encrypted with the same AES key received from the client.
5. **Server Sends Encrypted Response:** The server sends the encrypted response to the client.
6. **Client Decrypts the Response:** The client uses the AES key to decrypt the response.

3.2 Server Deployment using Java

In this section, we will discuss implementing AES and RSA encryption on the Server using Java. The source code of the files includes: **TravisAES.java**, **TravisRSA.java**, and Test.java. The detailed source code of the above files has been posted on GitHub at https://github.com/travistran1989/methodology-for-setting-up-e2ee-for-cross-platform-applications-with-java-javascript-and-flutter.

*3.2.1. **TravisAES** Class*

This is the main class implementing the encryption and decryption using the AES algorithm. This class contains many important methods and variables:

1. **CIPHER_ALGORITHM**: This variable specifies the encryption algorithm and padding method used. In this case, we use AES with PKCS5 padding method.
2. **IV_SIZE** and **IV_LENGTH**: These are the size and length of the Initialization Vector (IV). IV is a random byte string used in the encryption process.
3. **keySize** and **iterationCount**: These are the key size and number of iterations in the key generation process.
4. **encrypt()**: This method performs the data encryption. It generates a secret key from a password and a salt, then uses this key to encrypt the data using the AES algorithm.
5. **decrypt()**: This method performs the data decryption. It uses the same secret key that was generated during the encryption to decrypt the data.

*3.2.2. **TravisRSA** Class*

This is the class that implements encryption and decryption using the RSA algorithm. This class also contains many important methods and variables:

1. **RSA**: Defines the algorithm used, in this case, "RSA".
2. **RSA_ECB_PKCS1PADDING** and **RSA_ECB_OAEPWITHSHA1ANDMGF1PADDING**: Defines the algorithm and operation mode of RSA.
3. **getPublicKey()** and **getPrivateKey()**: These methods are used to generate public and private keys from a base64 string.
4. **encrypt()**: This method performs data encryption using the public key.

5. **decrypt()**: This method performs data decryption using the private key.

*3.2.3. **Test** Class*

This is the class that performs encryption and decryption testing. It uses both **TravisAES** and **TravisRSA** classes to perform encryption and decryption. This test encryption and decryption process includes the following steps:

1. Generate an RSA key pair and an AES secret key.
2. Encrypt the message using the AES secret key.
3. Encrypt the AES secret key using the RSA public key.
4. Send both the encrypted message and the encrypted AES key to the server.
5. The server uses its RSA private key to decrypt the AES key.
6. The server uses the decrypted AES key to decrypt the message.
7. The server processes the message and creates a response message.
8. The server encrypts the response message using the same AES key and sends it back to the client.
9. The client uses its AES key to decrypt the response message to get the original message.

*3.2.4. **dataType** Variable*

The **dataType** variable in both **TravisAES** and **TravisRSA** classes is used to determine how the encrypted data is represented when returned from the encryption methods. Specifically, **dataType** is an **enum** type variable with two values: **HEX** and **BASE64**. These are the two most common ways to represent binary data in text form:

1. **HEX** (Hexadecimal): Binary data will be represented as a string of hexadecimal numbers. Each binary byte will be represented by two hexadecimal characters, from 0 to F. For example: the binary byte **10101010** will be represented as **AA** in hexadecimal.
2. **BASE64**: Binary data will be represented as a string of base64 characters. Every 3 binary bytes will be represented by 4 base64 characters, from A to Z, a to z, 0 to 9, and two special characters + and /. For example: 3 binary bytes **10101010 11001100 11110000** will be represented as **qMy8** in base64.

Depending on the value of the **dataType** variable, the **encrypt()** and **decrypt()** methods will use the **toHex()** or **toBase64()** function to transform the encrypted binary data into a corresponding hexadecimal or base64 string. Similarly, they will use the **fromHex()** or **fromBase64()** function to convert the received hexadecimal or base64 string back into binary data for decryption.

3.3 Mobile Client Deployment using Flutter

The mobile application using Flutter also has a similar structure to the Java application with three main classes: **TravisAES**, **TravisRSA**, and **test_dart_encryption_example**.

*3.3.1. **TravisAES** Class*

The **TravisAES** class performs the encryption and decryption operations using the AES algorithm. Some of the main variables and methods of this class include:

1. **_keySize, _saltLength, _ivSize, _ivLength, _iterationCount**: Parameters for the key generation process from the password and for the IV (Initialization Vector).
2. **_dataType**: An enum type variable, determines the representation of the encrypted string when returned from the encryption method, in **hex** (hexadecimal) or **base64** form.
3. **encryptWithSalt(), encrypt()**: These methods perform the encryption of a clear text string with a password and return the encrypted string.
4. **decryptWithSalt(), decrypt()**: These methods perform the decryption of an encrypted string with a password and return the clear text string.
5. _generateRandom(), _generateCipher(), _generateKeyDerivator(), _generateKey(): Support methods in the key generation and encryption process.

*3.3.2. **TravisRSA** Class*

The **TravisRSA** class performs the encryption and decryption operations using the RSA algorithm. Some of the main variables and methods of this class include:

1. **keySize**: The size of the RSA key.
2. **_dataType**: Similar to in the **TravisAES** class, this enum type variable determines the representation of the encrypted string when returned from the encryption method, in **hex** (hexadecimal) or **base64** form.
3. **mode**: The encryption mode of RSA, can be PKCS1 or OAEP.
4. **publicKey, _privateKey**: The public and private keys used in the encryption and decryption process.
5. **encryptByPublicKey(), encryptByPublicKeyGetByDataType(), encryptByBase64PublicKey(), encrypt()**: These methods perform the encryption of a clear text string with a public key and return the encrypted string.
6. **decryptByPrivateKey(), decryptByPrivateKeyAndDataType(), decryptByBase64PrivateKeyAndDataType(), decrypt()**: These methods perform the decryption of an encrypted string with a private key and return the clear text string.
7. generateKeyPair(), secureRandom(), getBase64PublicKey(), getBase64PrivateKey(), publicKeyFromBase64(), privateKeyFromBase64(), getEncryptCipher(), getDecryptCipher(): Support methods in the key generation and encryption process.

3.4 Web Client Deployment using JavaScript

The web application using JavaScript has several main parts in the util.js file, including functions for encrypting and decrypting data.

*3.4.1. **encryptWithPublicKey** Function*

This function uses the JSEncrypt library to encrypt a plaintext with a public key. The public key is provided in base64 format and is converted to PEM format before use.

*3.4.2. **TravisAES** Object*

TravisAES is an object that contains methods related to encrypting and decrypting data using the AES algorithm. This object includes methods such as:

1. **init**: Initializes values for some attributes such as **keySize**, **ivSize**, and **iterationCount**.
2. **generateKey**: Generates a key from a provided password and salt.
3. **encryptWithIvSalt** and **decryptWithIvSalt**: Encrypt and decrypt data with the provided password, salt, and initialization vector (IV).
4. **encrypt** and **decrypt**: Encrypt and decrypt data with the provided password. This function will automatically generate salt and IV.

*3.4.3. **processResponseInterceptor** Test Function*

This test function handles responses from the server. If the response is the result of a login request, it will store the returned token and public key in **localStorage**. It also generates a random password and encrypts this password with the public key before storing it in **localStorage**. If the response is secured (specified by the **secure-api** header), the function will decrypt the data using the stored password.

*3.4.4. **processRequestInterceptor** Test Function*

This test function handles requests sent to the server. If the request needs to be secured (specified by the **Secure-Api** header), it will encrypt the request data with the stored password. The function also adds the stored token to the **Authorization** header of the request.

4 Results

The end-to-end encryption method that we have researched and applied in cross-platform applications with Java, Flutter, and JavaScript has yielded positive results. Specifically, we have achieved:

1. **High security**: Data is encrypted from the client side and only decrypted when it reaches the server. This ensures that user data is maximally protected from mid-transmission attacks.
2. **Cross-platform compatibility**: We have successfully implemented this method on both web applications (using JavaScript), mobile applications (using Flutter), and servers (using Java). This allows the end-to-end encryption method for applications to operate seamlessly and securely across various platforms.
3. **Good performance**: Although encryption and decryption may take some processing time, we have optimized this process to ensure that it does not significantly affect the user experience.
4. **Ease of use**: The **TravisRSA** and **TravisAES** classes are designed to be very user-friendly, with methods that are logically and clearly organized. Encryption and decryption operations are performed through common class methods, simplifying the use of end-to-end encryption (E2EE) in applications.

5 Discussion

This paper provides a solution for a rather practical problem in the field of information technology: how to set parameters for AES and RSA encryption for consistent use (encryption and decryption) across different programming language platforms. This is a challenging issue because each programming language has its own way of handling things and often they are not compatible with each other.

Part of the problem comes from the different implementations of encryption between programming languages. For example, an encryption algorithm may work well on Java but does not function as expected on JavaScript or Flutter. This is also true when choosing encryption parameters. These minor differences can lead to incompatibility and cause errors.

In this paper, we have addressed this issue by selecting and applying a standard for both AES and RSA to ensure compatibility across platforms. Through research and testing, we have chosen a suitable set of parameters and provided a comprehensive solution to this problem.

This paper not only helps to solve the specific problem of end-to-end encryption across multiple platforms but also opens up new research directions in creating cross-platform compatible solutions in the field of information security.

6 Conclusion

In this paper, we have researched and implemented a robust end-to-end encryption method for cross-platform applications using Java, Flutter, and JavaScript. We have demonstrated that using a common standard for AES and RSA across different platforms not only enhances security but also creates a seamless user experience across platforms.

In addition, we have presented a solution to the problem of encryption parameter compatibility between different programming languages, a practical issue that developers often face when building cross-platform applications. By choosing a suitable set of parameters and providing supporting tools, we have helped to solve this problem.

We hope that this research work will provide developers with a useful tool to build secure and efficient applications across multiple platforms.

7 Appendix

To help readers better understand the practical implementation of end-to-end encryption, we have included the complete source code of the application in a public repository on GitHub. This source code includes specific details of the files mentioned in the [Methods] section.

Below are the links to the detailed source code:

7.1 Server Deployment using Java

[1] TravisAES.java https://github.com/travistran1989/methodology-for-setting-up-e2ee-for-cross-platform-applications-with-java-javascript-and-flutter/blob/2a21aba44263296e05d8fac527ca57b8d6ce3e6a/src/java/TravisAES.java

[2] TravisRSA.java https://github.com/travistran1989/methodology-for-setting-up-e2ee-for-cross-platform-applications-with-java-javascript-and-flutter/blob/2a21aba44263296e05d8fac527ca57b8d6ce3e6a/src/java/TravisRSA.java

[3] Test.java https://github.com/travistran1989/methodology-for-setting-up-e2ee-for-cross-platform-applications-with-java-javascript-and-flutter/blob/2a21aba44263296e05d8fac527ca57b8d6ce3e6a/src/java/Test.java

7.2 Mobile Client Deployment using Flutter

[4] travis_aes.dart https://github.com/travistran1989/methodology-for-setting-up-e2ee-for-cross-platform-applications-with-java-javascript-and-flutter/blob/0308d19a9a995d6608dbd38020c20223872cb7ed/src/flutter/travis_aes.dart

[5] travis_rsa.dart https://github.com/travistran1989/methodology-for-setting-up-e2ee-for-cross-platform-applications-with-java-javascript-and-flutter/blob/0308d19a9a995d6608dbd38020c20223872cb7ed/src/flutter/travis_rsa.dart

[6] test_dart_encryption_example.dart https://github.com/travistran1989/methodology-for-setting-up-e2ee-for-cross-platform-applications-with-java-javascript-and-flutter/blob/0308d19a9a995d6608dbd38020c20223872cb7ed/src/flutter/test_dart_encryption_example.dart

[7] pubspec.yaml https://github.com/travistran1989/methodology-for-setting-up-e2ee-for-cross-platform-applications-with-java-javascript-and-flutter/blob/2a21aba44263296e05d8fac527ca57b8d6ce3e6a/src/flutter/pubspec.yaml

7.3 Web Client Deployment using JavaScript

[8] util.js https://github.com/travistran1989/methodology-for-setting-up-e2ee-for-cross-platform-applications-with-java-javascript-and-flutter/blob/2a21aba44263296e05d8fac527ca57b8d6ce3e6a/src/javascript/util.js

ACKNOWLEDGMENTS

We would like to express our sincere gratitude to Pitagon., JSC, which provided us with a practical and indispensable environment to conduct and develop this research. The unwavering support and dedication from the team members have enabled us to make significant findings in this study.

REFERENCES

[1] W. Stallings. 2017. *Cryptography and Network Security: Principles and Practice* (7th Edition). Pearson.

[2] W. Diffie, M. Hellman. 1976. *New Directions in Cryptography*, 22(6), pp.644-654. IEEE Transactions on Information Theory.

[3] R. Rivest, A. Shamir, L. Adleman. 1978. *A Method for Obtaining Digital Signatures and Public-Key Cryptosystems*, 21(2), pp.120-126. Communications of the ACM.

[4] J. Daemen, V. Rijmen. 2002. *The Design of Rijndael: AES - The Advanced Encryption Standard*. Springer.

[5] E. Rescorla. 2000. *HTTP Over TLS. RFC 2818*. Internet Engineering Task Force.

[6] A. Singh. 2016. *Flutter: A new tech for building native mobile apps*. Medium.

[7] Oracle. 2014. *Introduction to Java Programming Language*. Oracle.

[8] Mozilla. 2017. *JavaScript Guide*. Mozilla Developer Network.

[9] A. Herzog, N. Shahmehri, C. Duma. 2007. *An Ontology of Information Security*, 1(4), pp.1-23. International Journal of Information Security and Privacy.

[10] Open Web Application Security Project. 2017. *OWASP Top Ten Project*. OWASP Foundation.