JMessage Specification (r2)

Matthew Green

October 7, 2016

1 Introduction

JMessage is a simple end-to-end encrypted messaging application. A JMessage implementation consists of a server and one or more clients that interoperate to send encrypted messages. JMessage was designed for teaching purposes, and hence it was deliberately designed to include *potentially vulnerable, obsolete cryptography.* While you are free to expand on the current design and improve it, if you use the current version for anything that matters I will laugh at you.

2 Overview

A JMessage deployment consists of two components: (1) a JMessage server, and (2) one or more JMessage clients, which interoperate with the server to exchange messages. Each JMessage client generates its own cryptographic keypairs internally; the secret keys never leave the client. All messages are encrypted end-to-end to the receiving clients. This design ensures that even a curious server operator will not not see the content of the messages.

The JMessage Client. Each JMessage client interacts with the server using a RESTful API. It is responsible for interacting with the user, generating cryptographic keys, encrypting messages sent to other users, and decrypting messages received from other users. The client may also display information such as the cryptographic key fingerprint of another user.

The JMessage Server. The JMessage server does not implement any cryptographic functions. It interacts with the clients via HTTP to provide a simple API for the following functions:

1. Registration of public keys
2. Lookup of public keys
3. Message delivery
4. Retrieval of any undelivered messages

Security note: The JMessage server (as currently implemented) uses unencrypted HTTP to implement this API. In a secure deployment, this would have to be replaced with HTTPS. As described in this specification, the server also does not implement password authentication, which means that any user can impersonate any other user. Obviously this is not good.

2.1 Overview of JMessage encryption

JMessage uses three cryptographic primitives: RSA encryption (with PKCS #1v1.5 padding), DSA signing, and AES encryption.

Each client is responsible for generating and maintaining two long-term keypairs: an RSA encryption keypair $(pk_{\mathsf{RSA}}, sk_{\mathsf{RSA}})$, and a DSA signing keypair $(pk_{\mathsf{DSA}}, sk_{\mathsf{DSA}})$. These keys may be generated each time the client starts up, or they may be generated once and stored persistently on disk. The secret keys are never sent to the server.

A typical JMessage interaction proceeds as follows:

1. The client transmits its public keys pk_{RSA}, pk_{DSA} to the server, along with its chosen username sender_userid. The server stores these keys in a database.
2. When a client with username sender_userid wishes to transmit a message to another user receipient_userid, it first calls the server to look up the recipient's public keys from the database.
3. The sending client then:
   (a) Prepends sender_userid to the message and appends a CRC32 of the resulting string.
   (b) Generates a random AES key K, and encrypts the message (output of step 3a) under this key to obtain C_2.
   (c) Encrypts K under the recipient's RSA key pk_{RSA} to obtain C_1.
   (d) Concatenates C_1 and C_2 and signs the result with its DSA secret key to obtain a signature σ.
   (e) Sends $(\mathsf{sender\_userid}, \mathsf{receipient\_userid}, C_1, C_2, \sigma)$ to the server, which stores the message in the database.
4. The recipient calls the server to obtain any undelivered messages for receipient_userid.
5. For each message, the recipient calls the server to obtain the sender's public keys, then verifies the signature using pk_{DSA}.
6. The recipient decrypts the RSA ciphertext to obtain the AES key, then decrypts the AES ciphertext to obtain the message.
7. If the message validates correctly (*i.e.,* it contains the correct sender user ID and formatting checks), the recipient displays the message, and repeats the process above to send an encrypted "read receipt" for the message.

3 JMessage Encryption

This section provides details on the encryption algorithms and protocol used by JMessage.

3.1 Notation

Let || denote concatenation. We will denote the ASCII space character as `ASCII(0x20)`, the ASCII colon (:) character as `ASCII(0x3A)`, and the ASCII percentage character % as `ASCII(0x25)`.

3.2 Key Generation

Each JMessage client generates a 1024-bit RSA keypair $(pk_{\mathsf{RSA}}, sk_{\mathsf{RSA}})$ and a 1024-bit DSA keypair $(pk_{\mathsf{DSA}}, sk_{\mathsf{DSA}})$. Secret keys are never transmitted outside of the client; hence, they may be stored in

any format you like (or held in RAM only). To encode the public keys into a format for transmission to the server, the following procedure is used:

1. Each of $pk_{\mathsf{RSA}}, pk_{\mathsf{DSA}}$ is encoded using the default ASN1/DER encoding used by Java.[1]
2. The resulting DER encoded keys are further encoded using Base64.
3. The Base64 encoded keys are concatenated as:[2]

$$\mathsf{Base64\_RSA\_PubKey}||\mathtt{ASCII(0x25)}||\mathsf{Base64\_DSA\_PubKey}$$

The resulting string is transmitted to the server for key registration. Decoding proceeds by first parsing the resulting key string, then reversing the steps above.

3.3 Encryption

The encryption procedure assumes that the sender has already registered its own public keys $(pk^S_{\mathsf{RSA}}, pk^S_{\mathsf{DSA}})$, and has retrieved the recipient's keypair $(pk^R_{\mathsf{RSA}}, pk^R_{\mathsf{DSA}})$ from the server. Let M be a UTF8 encoded text message.[3] A complete description of the encryption procedure now follows:

1. Generate a random 128-bit AES key K using a secure random number generator.
2. Encrypt K using the RSA encryption with PKCS#1v1.5 padding under the recipient's key pk^R_{RSA} to obtain a ciphertext C_1.
3. Prepend $\mathsf{sender\_userid}||\mathtt{ASCII(0x3A)}$ to the message M to obtain $M_{\mathsf{formatted}}$.
4. Compute a CRC32 on the message $M_{\mathsf{formatted}}$, and append the 4-byte CRC value (in network byte order) to the end of $M_{\mathsf{formatted}}$ to create M_{CRC}.
5. Pad the length of the message M_{CRC} to a multiple of 16 bytes using PKCS5 padding to create M_{padded}.
6. Generate a random 16-byte initialization vector IV using a secure random number generator.
7. Encrypt M_{padded} using AES in CTR mode under K and IV. Prepend IV to the resulting ciphertext to obtain C_2.
8. Separately Base64 encode each of C_1 and C_2 to obtain C_1^{Base64} and C_2^{Base64} (respectively) in UTF8 format.
9. Compute a DSA signature σ on the UTF8 encoded string $C_1^{\mathsf{Base64}}||\mathtt{ASCII(0x20)}||C_2^{\mathsf{Base64}}$.
10. Set σ^{Base64} to be the Base64 encoding of σ (in UTF8 encoding).
11. Output the string $\mathbf{C} = C_1^{\mathsf{Base64}}||\mathtt{ASCII(0x20)}||C_2^{\mathsf{Base64}}||\mathtt{ASCII(0x20)}||\sigma^{\mathsf{Base64}}$.

The resulting string $\mathbf{C}$ is sent to the server for delivery to the recipient.

3.4 Decryption

To decrypt a received message $\mathbf{C}$ from sender S, the recipient follows the procedure below. If any step fails, the decryptor shall abort processing and drop the message silently, *i.e.,* without notifying the user.

[1] In Java this is the default encoding produced by the `getEncoded()` method of `java.security.PublicKey`. Other languages and libraries can also decode this format; you are free to use Google to find example code.

[2] The ASCII character $0x20$ is the space character.

[3] In Java this message can be obtained by calling `String.getBytes("UTF-8")`. Other languages have similar capability.

1. Contact the server to obtain the public key pk^S_{DSA} for the sender.
2. Parse the the string $\mathbf{C}$ as $C_1^{\mathsf{Base64}}||\texttt{ASCII(0x20)}||C_2^{\mathsf{Base64}}||\texttt{ASCII(0x20)}||\sigma^{\mathsf{Base64}}$.
3. Base64 decode each of $C_1^{\mathsf{Base64}}, C_2^{\mathsf{Base64}}, \sigma^{\mathsf{Base64}}$ individually to obtain the values C_1, C_2, σ.
4. Verify the DSA signature σ using pk^S_{DSA} on the message $C_1^{\mathsf{Base64}}||\texttt{ASCII(0x20)}||C_2^{\mathsf{Base64}}$. If verification fails, abort.
5. Decrypt the RSA ciphertext C_1 using the recipient's secret key sk^R_{RSA} to obtain K.
6. Parse C_2 to obtain the prepended IV. Decrypt C_2 using AES in CTR mode to obtain M_{padded}.
7. Verify and remove the PKCS5 padding to obtain M_{CRC}. Abort if the padding is incorrectly structured.
8. Parse M_{CRC} as $M_{\mathsf{formatted}}||CRC$, where CRC is 4 bytes long. Compute a CRC32 on the message M, and compare to CRC. Abort if the comparison fails.
9. Parse $M_{\mathsf{formatted}}$ as $\mathsf{sender\_userid}||\texttt{ASCII(0x3A)}||\texttt{M}$. Verify that $\mathsf{sender\_userid}$ is the correct user ID for S. Abort if the username does not match.
10. If the incoming message is not a Read Receipt message, send a Read Receipt to the sender. (See §3.5 below.)
11. Output M.

3.5 Read Receipts

Whenever a client successfully receives and decrypts a message, it automatically transmits a distinguished *read receipt* message to the sender. The read receipt message is a transmitted as a normal encrypted message using the encryption procedure above. Let `<messageID>` be the integer message ID received from the server when the message is delivered (see §4.5). The plaintext of the message has the following structure:

```
>>>READMESSAGE <messageID>
```

Clients are free to filter out and ignore read receipt messages. They may also display information about these receipts to the user, at the developer's preference. It is important that the client should never respond to a receipt with its own receipt, as tragedy will quickly ensue.

3.6 Key Fingerprints

JMessage key fingerprints are computed using the following process:

1. First, obtain the *encoded* public key (this is the result computed at the end of §3.2 above).
2. Encode this result as a UTF-8 string.
3. Hash the resulting string using SHA256.
4. Encode the resulting 32-byte hash as a hexadecimal string.

4 Server Interaction

The JMessage server supports five functions: key registration, key lookup, message delivery, message lookup, and a function to list all registered users. Requests and responses are transmitted using plain HTTP GET or POST requests with JSON encoding used to transmit data structures. No

HTTPS encryption or password authentication is used.[4] The format of these messages is described below.

HTTP communications: By default the reference server operates on port 80. All GET and POST requests should include the following header:

```
Accept: application/json
```

4.1 Registering a key

To register a key consisting of a string `<encoded keydata>` to username `<sender username>`, first construct the following JSON object:

```
{"keyData": "<encoded keydata>"}
```

Now POST the message to the following path:

```
/registerKey/<sender username>
```

4.2 Key lookup

To look up a key for `<username>` on the server, a client issues an HTTP GET to the following path:

```
/lookupKey/<username>
```

On success this returns a JSON object named `keydata` that contains an opaque encoded string.

4.3 Enumerating all registered users

To produce a list of all registered users, a client issues an HTTP GET to the following path:

```
/lookupUsers
```

On success this returns a JSON object containing `numUsers` and an array of usernames `users`.

4.4 Obtaining undelivered messages

To obtain all undelivered messages for user `<username>`, the client issues an HTTP GET to the following path:

```
/getMessages/<username>
```

In response, this produces a JSON integer `numMessages`, and a JSON array `messages`. Each element in the messages array consists of the long integers `sentTime`, `messageID` and the strings `message` and `senderID`.

[4] Yes, we know this is hilariously stupid.

4.5 Sending a message

To transmit a message `<encryptedMessage>` for receiver ID `<recipient username>` with integer message ID `<messageID>` to the server from user `<sender username>`, first construct a JSON object with the following structure:

```
{"recipient":  "<recipient username>", "messageID":  "<messageID>",
"message":  "<encryptedMessage>" }
```

Now POST the message to the following path:

```
/sendMessage/<sender username>
```

Message IDs are integers chosen by the sending client. Clients are free to use any strategy they wish for assigning message IDs to outgoing messages. These message IDs will be used in formulating *read receipts* (see §3.5).

5 Client configuration

Each client should be equipped with the ability to specify an arbitrary server hostname, port and client username. The client does not need to check for messages automatically (*i.e.,* without user interaction), but the JMessage reference client does this using a background thread.