MIECT: Security 2017-18

Secure Messaging Repository System

Milestone: Nov 9 and 10 (security overview)
Presentation of final project: 1st week of Jan, 2018
Deadlines: Nov 6 (security overview), Dec 31 (final project)

André Zúquete, João Paulo Barraca

Changelog

- v1.0 - Initial Version.

1 Project Description

The objective of this project is to develop a system enabling users to exchange messages asynchronously. Messages are sent and received through a non-trustworthy central repository, which keeps messages for users until they fetch them. The resulting system is composed by a Rendezvous Point, or Server, and several clients exchanging messages. The system should be designed to support the following security features:

- **Message confidentiality, integrity and authentication**: Messages cannot be eavesdropped, modified or injected by a third party or the server;
- **Message delivery confirmation**: message readers should be able to prove that they have had access to their contents (i.e. have read them);
- **Identity preservation**: There is a direct association between one user and one Portuguese Citizen, and vice-versa, using the Portuguese Citizen Card.

2 Project Description

2.1 System Components

As far as messaging systems go, we can consider the existence of two main components: (i) multiple clients, through which users interact, and (ii) a server, which serves as rendezvous point for all clients to connect.

2.1.1 Repository server

The server will expose a stream (TCP/IP) socket connection through which clients can exchange `JSON` messages with it. Clients never interact directly, they only interact with the server.

This component will store a list of message boxes, and the respective cryptographic material to interact with their owners. Internally, the server will keep a list with at least the following information of each user (not exactly in this format):

```
{
    "id": <identifier>,
    "uuid": <user universal identifier>,
    "mbox": <message box path>,
    "rbox": <sent box path>,
    "secdata": <security-related-data>
}
```

The **id** field represents an internal identifier, defined by the server, which may be used as a short reference to refer to a user. This is an integer value.

The **uuid** field represents a long-term, unique user identifier, defined by the user, which is used as the primary value to distinguish users. Users can freely define this value; it is suggested to use a digest of a public key (or certificate) to create a **uuid**.

The **mbox** field represents the path to a user's message box. A message box is a directory (one per user) with a different file per message. These files are never deleted.

The **rbox** field represents the path to a user's receipt box. A receipt box is a directory (one per user) with a different file per sent message and 0 or more read receipts. A receipt is a confirmation that a message was received and properly decoded by a receiver. Messages in the receipt box are encrypted in a way that only the sender could see it, thus they are not exact copies of the messages delivered to users. This allows a message sender to (i) get

access to the messages it has send and (ii) to check which messages were received and acknowledged by receivers. These files are never deleted.

The **secdata** field contains security-related data relevant for the exchange of secure messages between users through their messages boxes (messages and receipts). For example, it may contain a public key value, or a public Diffie-Hellman value, which can be used to securely deliver a message to his/her owner. It may also contain private users' secrets, such as an asymmetric private key, or a private Diffie-Hellman value, encrypted with a password-derived key. For both cases, all its data should be authenticated (i.e., signed) with the user's (authentication) private key of his/her Citizen Card.

For simplicity, both **mbox** and **rbox** are derived from the user **id** and located below directories `mboxes` and `receipts`.

Messages inside message or receipt boxes are identified by a name formed by a sequence of numbers `U_S`, where:

- In a message box, `U` is the identifier of the sender and `S` is a sequence number for that sender, starting at 1;
- In a receipt box, `U` is the identifier of the receiver and `S` is the sequence number above referred.

For instance, is user 7 sends a first message to user 5, it will appear in the message box of user 5 as `7_1` and in the receipt box of user 7 as `5_1`. Messages in messages boxes already red have a name prefixed with `_`; the same happens with receipts in receipt boxes, but these have an extra extension with the date when the receipt was received by the server.

2.2 Processes

There are several critical processes that must be supported. Students are free to add other processes as deemed required.

- **Create a user message box**: A client issues a `CREATE` message to the server. The server replies with the internal identifier given to the user.
- **List users' messages boxes**: A client issues a `LIST` message to the server. The server replies with a message of same type containing a list of users with a message box;
- **List new messages received by a user**: A client issues an `NEW` message to the server. The server replies with a list of messages not yet red by the user;

- **List all messages received by a user**: A client issues an `ALL` message to the server. The server replies with a list of all messages received by the user.
- **Send message to a user**: A client issues a `SEND` message when it wants to send a message to a user.
- **Receive a message from a user message box**: A client issues a `RECV` message when it wants to receive a message from a user message box.
- **Send receipt for a message**: A client issues a `RECEIPT` message when it wants to send a receipt to a user, in order to acknowledge a received message.
- **List messages sent and their receipts**: A client issues a `STATUS` message when it wants to check the status of a sent message (i.e., if it has a receipt or not).

3 Messages

All messages should be sent in the JSON format (i.e. as JSON objects), and must obey the specified encapsulation format. If the value of a field is not clearly specified, students can enhance and or define its content. Any binary content must be converted to a textual format, such as Base-64. **Students can add more message types or add fields to the messages specified**.

3.1 Server response messages

The server will respond with

```
{
    <command-specific attributes>
}
```

upon a successful. Otherwise, it will respond with

```
{
    "error":"error message"
}
```

3.2 CREATE

Message type used to create a message box for the user in the server:

```
{
    "type": "create",
    "uuid": <user uuid>,
    <other attributes>
}
```

The `type` field is always `create`.

The `uuid` field should contain a unique (and not yet known to the server) user unique identifier. We suggest using the digest of a user's public key certificate, extracted from his/her Citizen Card.

The remaining message fields should contain the security-related information required to encrypt/decrypt messages, and it should be signed with the Citizen Card's credentials (authentication private key). The signature is mandatory and should be checked with the public key certificate belonging to the same message.

The server will respond with the id (an integer) given to the user:

```
{
    "result": <user id>
}
```

3.2.1 LIST

Sent by a client in order to list users with a message box in the server:

```
{
    "type": "list",
    "id": <optional user id>
}
```

The `type` field is a constant with value `list`.

The `id` field is an optional field with an integer identifying a user to be listed.

The server reply is a message containing in a `result` field all information regarding a single user or all users. This information corresponds to the creation message, excluding the `type` field:

```
{
    "result": [{user-data}, ...]
}
```

3.2.2 NEW

Sent by a client in order to list all new messages in a user's message box:

```
{
    "type": "new",
    "id": <user id>
}
```

The `type` field is a constant with value `new`.

The `id` field contains an integer identifying the user owning the target message box.

The server reply is a message containing in a `result` field an array of message identifiers (strings). These should be used as given to have access to messages' contents.

```
{
    "result": [<message identifiers>]
}
```

Note: it is possible to apply this command to all users' message boxes and even without having a message box created for the requesting user!

3.2.3 ALL

Sent by a client in order to list all messages in a user's message box:

```
{
    "type": "all",
    "id": <user id>
}
```

The `type` field is a constant with value `all`.

The remaining fields of the request are similar to the previous one (NEW).

The server reply is similar to the previous one (NEW), but the result an array containing two arrays: one with the identifiers of the received messages, and another with the identifiers of the sent messages.

```
{
    "result": [[<received messages' identifiers>][sent messages' identifiers]]
}
```

Note: in the array of received messages' identifiers, the ones already red can be distinguished from the others by their name (prefixed with _).

3.2.4 SEND

Sent by a client in order to send a message to a user's message box:

```
{
    "type": "send",
    "src": <source id>,
    "dst": <destination id>,
    "msg": <JSON or base64 encoded>,
    "copy": <JSON or base64 encoded>
}
```

The `type` field is a constant with value `send`.

The `src` and `dst` field contain the identifiers of the sender and receiver identifiers, respectively.

The `msg` field contains the encrypted and signed message to be delivered to the target message box; the server will not validate the message.

The `copy` field contains a replica of the message to be stored in the receipt box of the sender. It should be encrypted in a way that only the sender can access its contents, which will be crucial to validate receipts.

The server reply is a message containing the identifiers of both the message sent and the receipt, stored in a vector of strings:

```
{
    "result": [<message identifier>,<receipt identifier>]
}
```

3.2.5 RECV

Sent by a client in order to receive a message from a user's message box.

```
{
    "type": "recv",
    "id": <user id>,
    "msg": <message id>
}
```

The `type` field is a constant with value `recv`.

The `id` contains the identifier of the message box.

The `msg` contains the identifier of the message to fetch.

The server will reply with the identifier of the message sender and the message contents:

```
{
    "result": [<source id,<base64 encoded message>]
}
```

Note: Being able to read a message from a mailbox doesn't mean that it is possible to "understand" the message, as it may be encrypted for someone other than the requester.

3.2.6 RECEIPT

Receipt message sent by a client after receiving and validating a message from a message box:

```
{
    "type": "receipt",
    "id": <user id of the message box>,
    "msg": <message id>,
    "receipt": <signature over cleartext message>
}
```

The `type` field is a constant with value `receipt`.

The `id` contains the identifier of the message box of the receipt sender.

The `msg` contains the identifier of message for which a receipt is being sent.

The `receipt` field contains a signature over the plaintext message received, calculated with the same credentials that the user uses to authenticate messages to other users. Its contents will be stored next to the copy of the messages sent by a user, with a extension indicating the receipt reception date.

The server will not reply to this message, nor will it validate its correctness.

3.2.7 STATUS

Sent by a client for checking the reception status of a sent message (i.e. if it has or not a receipt and if it is valid):

```
{
    "type": "status",
    "id": <user id of the receipt box>
    "msg": <sent message id>
}
```

The `type` field is a constant with value `status`.

The `id` contains the identifier of the receipt box.

The `msg` contains the identifier of sent message for which receipts are going to be checked.

The server will reply with an object containing the sent message and a vector of receipt objects, each containing the receipt data (when it was received by the server), the identification of the receipt sender and the receipt itself:

```
{
    "result":{
        "msg": <base64 encoded sent message>,
        "receipts": [
            {
                "data":<date>,
                "id":<id of the receipt sender>,
                "receipt":<base64 encoded receipt>
            },
            ...]
        }
}
```

4 Resources

The source code of a draft server will be provided (both in Java and Python). The server implements all the described features except security mechanisms. Therefore, its code needs to be enriched to incorporate security-related features. Students are free to implement their own server from scratch.

5 Functionalities to implement

The following functionalities, and their grading, are to be implemented:

- (1 point) Setup of a session key between a client and the server prior to exchange any command/response;
- (1 point) Authentication (with the session key) and integrity control of all messages exchanged between client and server;
- (1 point) Add to each server reply a genuineness warrant (i.e., something proving that the reply is the correct one for the client's request, and not for any other request);

- (1 point) Register relevant security-related data in a user creation process;
- (2 points) Involve the Citizen Card in the user creation process;
- (2 points) Encrypt messages delivered to other users;
- (1 point) Signature of the messages delivered to other users (with the Citizen Card or another private key) and validation of those signatures;
- (1 point) Encrypt messages saved in the receipt box;
- (2 points) Send a secure receipt after reading a message;
- (1 point) Check receipts of sent messages;
- (1 point) Proper checking of public key certificates from Citizen Cards;
- (1 point) Prevent a user from reading messages from other than their own message box;
- (1 point) Prevent a user from sending a receipt for a message that they had not red.

To simplify the implementation, you may:

- Assume the use of well-established, fixed cryptographic algorithms. In other words, for each cryptographic transformation you do not need to describe it (i.e., what you have used) in the data exchanged (encrypted messages, receipts, etc.). **A bonus of 2 points** may be given if the complete system is able to use alternative algorithms.
- It can be assumed that the server has a non-certified, asymmetric key pair (Diffie-Hellman, RSA, etc) with a well-know public component.

Grading will also take into consideration the elegance of both the design and actual implementation.

Up to 2 (two) bonus points will be awarded if the solution correctly implements interesting security features not referred above.

A report should be produced addressing:

- the studies performed, the alternatives considered and the decisions taken;
- the functionalities implemented; and
- all known problems and deficiencies.

Grading will be focused in the actual solutions, with a strong focus in the text presented in the report (4 points), and not only on the code produced!

It is strongly recommended that this report clearly describes the solution proposed for each functionality.

Using materials, code snippets, or any other content from external sources without proper reference (e.g. Wikipedia, colleagues, StackOverflow), will imply that the entire project will not be considered for grading or will be strongly penalized. External components or text where there is a proper reference will not be considered for grading, but will still allow the remaining project to be graded.

The detection of a fraud in the development of a project (e.g. steal code from a colleague, get help from an external person to write the code, or any other action taken for having the project developed by other than the responsible students) will lead to a grade of 0 (zero) and a communication of the event to the University academic services.

6 Project phases

The security features should be fully specified prior to start its implementation. The client application must be written from scratch, while the server **may** use the code provided.

We recommend the following steps for a successful project development:

- Develop a complete, non-secure client application. This step can start immediately.
- Start the report with a draft of the security specification.
- Start the coding of security features by adding secure data to each user profile, upon their initial registration.
- Use the secure data of proper selected users to protect messages and receipts.
- Involve the Citizen Card in the whole process and use its certificates to extract user identity information.
- Protect the communication between the client and server.

There will be a **project milestone in November, 9 and 10**. In this milestone students should provide a **written overview** (through their CodeUA project, see Section 7) of the security mechanisms they have designed (not implemented!) for their project. This milestone **does not involve any grading**, and will serve to assess the progress of students and to give them some feedback. Since the feedback will be given based on the written

contents, they must be stored in CodeUA a couple of days before the milestone date. Please send an **email to the course teachers** upon the deliver of the overview.

7 Delivery Instructions

You should deliver all code produced and a report before the deadline. That is, 23.59 of the delivery date. The delivery date will be a day in the first week of January, 2018.

In order to deliver the project you should create a project in the CodeUA[1] platform. The project should be named after the course name (`security2017`), the practical class name (e.g. `p2`) and the group number in the class (e.g. `g5`), with yields the following complete format for the examples given before: `security2017-p2g5`). **Please commit to this format to simplify the evaluation of the projects! Outliers will be penalized!** Each project should be given access to all the course professors (André Zúquete and João Paulo Barraca).

Each CodeUA project should have a `git` or `svn` repository. The repository can be used for members of the same group to synchronize work. After the deadline, and unless otherwise requested by students, the content of the repository will be considered for grading.

[1] `https://code.ua.pt`