Integrated MSc Course on Informatics Engineering, DI/FCT/UNL
Computer Networks and Systems Security / Semester 1, 2019-2020

Practical Evaluation: Work Assignment #2

A Secure REST-Based Messaging Repository System with Mutual TLS Client/Server Authentication and Access Control
(version 1.1)

Submission period: From 9 to 15 December/2019
Deadline for submission: 15/December /2019, 23h59

Summary

Fort this work-assignment students will develop a Secure Messaging Repository Service, providing mutual TLS authentication and access-control for authorized clients performing asynchronous messaging. Among other possibilities for the defined architecture, the system can be addressed as a TLS-Rest enabled service support users to exchange messages, asynchronously, intermediated by the provided service. Messages are sent and received through a non-trustworthy central repository (where Honest-But-Curious) System administrators act, and where attacks can be conducted to beak the confidentiality, integrity and authenticity assumptions on the stored/fetched messages. Considering the possible attacks, the system will keep messages for users until they fetch them with the necessary security guarantees. In the system model there are the following components: a Rendezvous Point or Server, and several clients exchanging messages. The system should be designed to support the following base security features:

- *Message confidentiality, integrity and authentication: Messages cannot be eavesdropped, modified or injected by a third party or the server, resisting to all the attacks in the X.800 threats typology.*
- *Message delivery confirmation: message readers should be able to prove that they have had access to their contents (i.e. have read them), based on authorization and peer-authentication proofs.*

Additionally, depending on extended challenges and evaluation scoring, other security features can be considered by students, to design and develop their solutions.

1. System model and components

As far as messaging systems go, we can consider the existence of two required main components: (i) clients (or multiple clients), through which users interact, and (ii) a server (used as a rendezvous endpoint) for all clients to connect, providing the service with the support of a back-end messaging

repository. The rendezvous will be preferentially materialized as a REST-based endpoint (but other possible solutions are open to students), allowing for the interactions between clients and servers supported by TLS, with mutual-authentication handshake guarantees and appropriate configurable TLS endpoints, via configuration files (in the clients and in the server side). In the following sections, each component in the required architecture is initially specified. Students are free to improve the specification beyond the mandatory base requirements that must be always supported.

1.1 Repository Server

The server will expose a REST service endpoint, implementing TLS with configuration options:

- Enabled Ciphersuites for TLS
- Enabled TLS protocol versions
- Possibility to support SERVER-ONLY unilateral TLS Authentication or CLIENT-SERVER mutual TLS Authentication

The repository server will store a list of message boxes, and the respective cryptographic material and constructions 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>
}
```

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

uuid: this field represents a long-term, unique user identifier (or an user-identifier together with a list of attributes) that can be defined by the user. In this case, the unique identifier can be managed as a hash-value of the attributes that can include owner public keys or public-key certificates and/or other fingerprints. However students are free to manage other hash forms for uuid, using for example dual hash-constructions and/or MACs, to minimize the probability of collisions. The uuid it is used as the primary value to distinguish users. Users can freely define this value under their own generation process; it is suggested to use a digest of a public key (or certificate) to create a uuid.

mbox: represents the path to a user's message box. A message box is a directory (one different directory per user) with a different file per message. These files are never deleted in the server side.

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, and can be used as a primary proof of non-repudiation of potential receivers, with the guarantee that if the receiver accessed messages to read, a non-repudiation proof will be issued automatically in the server side. 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 different 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 the receivers. These files are never deleted.

The *secdata* field must contain security-related data relevant for the exchange of secure messages between users through their messages boxes (messages and receipts). As a reference it may contain a public key value, or a public Diffie- Hellman value (or possibly both), 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, properly encrypted with a password (or password-phrase) derived key. For both cases, all its data should be authenticated (i.e., signed) with the user's (authentication) private key in a user certificate.

For initial simplicity, both mbox and rbox are derived from the user id and located below directories mboxes and receipts, to be managed in a more easy way.

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.

Students must follow the above functionality for the service as the base reference, but can extend the requirements with additional valuable requirments considered as interesting for the system design and related implementation.

1.2 Provided functions and processes

There are several critical processes that must be supported (as operations in the server-side API), however, students are free to add other functions and processes as deemed decided and required. The mandatory requirement is that messages (as data units) must be supported with end-to-end security guarantees for confidentiality, integrity and authentication (including peer-authentication arguments).

- 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).

The security properties must be supported in such a way that cannot be compromised by the adversaries in the server side, including those that can use instrumentation and system management functions and tools to access memory of execution processes or storage.

1.3 Messages and Structure

As initial reference all messages should be supported in a JSON format (i.e. as JSON objects), and must follow a specified (well-defined) encapsulation format. If the value of a field is not clearly specified, students can enhance and or define its content for the better control of the used messages' format. Any binary content must always be converted to a textual format, such as Base-64. Students can add more message types or attributes or add fields to the messages, as complementarily specified. For the minimal scope of the TP2 work, messages in text-base support are addressed. However, it would be interesting and valuable (even that not mandatory) to support enhanced message formats and attributes, namely as pictures, movies, etc. Students are invited to explore S/MIME data representation formats if motivated to enrich the solution.

1.3.1 Server Response Messages

The server will respond with

```
{
    <command-specific attributes>
}
upon a successful. Otherwise, it will respond with
{
    "error":"error message"
}
```

1.3.2 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>
}
```

1.3.3 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}, ...]
}
```

1.3.4 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 mes- sage 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!

1.3.5 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]]
}
```

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

1.3.6 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>]
}
```

1.3.7 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 mes- sage 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.

1.3.8 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 mes- sages 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.

1.3.9 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>
        },
        ...]
  }
 }
```

2. Users

Users will be represented as the principals that will use the system. Users are described by user-certificates that must exist in a certification chain of a "pseudo-CA". So users' certificates must be issued by this CA. Please refer to te LAB materials to setup and to design your solution, This pseudo-CA must be used as the root-chain of all certificates in the system and the CA root certificate must be the unique certificate initially trusted by clients and by the server.

3. RESOURCES

Initial source code of a draft server (not as the final REST based service) will be provided (available in Java or Python). This draft server implements the described features, without any security mechanisms. Therefore, its code needs of course to be enriched to incorporate all the security-related features, as well as the final code to be deployed as a REST-based TLS enabled service. Students are free to implement their own server from scratch, if this is considered a better option.

4. Functionality and evaluation criteria

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

- (1 point) Dynamic setup of session keys and security association parameters between clients and the server prior to exchange any command/response. This must consider the minimal setup requirements in both sides.

- (1 point) Message authentication (with the established session key and security association parameters) 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 re- quest, and not for any other request);

The previous criteria must be covered by the use of TLS enabled channels between clients and servers. TLS endpoints must be configurable for the required TLS ciphersuites, authentication modes (including mutual TLS authentication) and required TLS protocol versions, subjacent to the TLS handshake between clients and the server.

- (1 point) Register relevant security-related data in a 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.

The previous criteria must be established in order to be used as countermeasures considering the adversary model definition and provided as stated above under "end-to-end" security arguments involving the users (clients).

- (3 points): used as extra points for highlights in each implemented solution

  Highlights and extra-evaluation (among others considered by the students):

  - Flexibility provided by TLS Parameterization of the Service
  - Flexibility in providing different cryptographic algorithms and constructions
  - Use of Diffie-Hellman Authenticated Key-Exchanges
  - Solution provided as a Cloud-as-a-Service solution (that could be ready for deployment in a Cloud-Computing+Storage Provider) or in Cloud-Provisioned Virtual Machines.
  - With the provided protection, messages and the system must be entirely protected from the "honest-but-curious" adversaries and system administrators managing the system in the server side. The more protection you provide for this case, the better will be the evaluation
  - Replication and consistency for high-availability and support against DoS or message-deleting form server instances could be a very interesting feature. Possibly this will be more difficult to conjugate with the required solution in useful time. However during the implementation you can think on how to provide those extended features and then you can discuss this in your final report (to be considered as an extra-evaluation criteria of your work).

Up to 3 (two) bonus points will be awarded if the solution correctly implements other interesting security features not necessarily 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 the report must clearly describe the solution proposed for the provided functionality, addressing the designed and implemented functional requirements, highlighting the considered features of the designed and implemented solution and clearly presenting the system model, software architecture, adversary model definition, implementation and validation details. Using materials, code snippets, or any other content from external sources without a clear understanding and proper reference and discussion in the report (e.g. Wikipedia, colleagues, StackOverflow), will imply that the entire project will be considered as poor-grading or can 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, with proper references in the report.

A template with a reference organization of the report (as a technical paper) is available, as the inspiration to address it as a technical paper with convenient soundness

5. Dates and submission process

The delivery of the WA2 work-assignment wil be conducted in the same way as the WA1. Groups (or individual students) must have a GitHub repository, shared with the professor. Groups and students must be ready for a demo and discussion in a “on-demand required basis”. The submission of WA2 will use a Google form that will be open between 9 and 15 December (with the deadline on 15/Dec/2019, 23h59m, Portugal TIME). To submit the form, students must include the REPORT (using an initial provided template), the GitHub URL of the project development and must answer to the mandatory FOM questions.