UNIVERSITEIT GENT

FACULTEIT INGENIEURSWETENSCHAPPEN

Information Security Assignment
Security for a (Car) Key App

Report

Academic year 2015-2016

Group 1

Eveline Hoogstoel
Wouter Pinnoo
Stefaan Vermassen
Titouan Vervack

Contents

1 Introduction

1.1 Problem statement

Companies with business in car rental services, or companies looking for a digitalisation of the keys or their company cars, can opt to use a system where drivers can book, (un)lock and start a pre-booked car using a smartphone application. In this assignment, the security aspect of this use case is designed and a prototype is presented that illustrates a working implementation of the designed security mechanisms. Sequence diagrams and an UML diagram of the prototype can be found in Appendix A.

1.2 Requirements

The design of our system will be made with a number of requirements in mind. First of all, simplicity is important to minimise unneeded delays and thus not harming the availability requirement of our system. Although distributed attacks such as Distributed Denial of Service (DDoS) attacks are very hard to avoid in its entirety, some measures will be taken in the proposed system to ensure a reasonable availability. Availability is very important for the users of the system since large delays when, for example, opening a car will be experienced as main weaknesses of the system. Also, the system must be able to operate when the network coverage can occasionally not be assured. However, a few assumptions about network coverage will be made in Section 2.

A second requirement is authentication. The system should protect thieves from impermissible usage of company cars, and will thus have to make sure only authorised users have access to cars they have booked.

Next, all communication that is vulnerable to eaves-dropping (e.g. communication via a public insecure communication network) or any other kind of misuse, will have to be protected to ensure confidentiality. It must be impossible for an intruder to capture classified information (such as personal information).

Finally, data-integrity and non-repudiation are important requirements. The former states that any modifications made on message sent over the (public) communication network will cause the other party to detect that the message is altered. The latter ensures that neither sides of the communication network can question the data-origin validity of a message. One of the reasons that this is an important requirement is that users will actually be billed for the usage of the company cars — after a billing statement is sent, one must not be able to reasonably argue the contents of the bill.

2 System description

First, a short overview of the general architecture of the system will be given. Next, the functionalities and general security mechanisms of each module will be explained.

2.1 Architecture overview

The system we propose consists of three parts: the server, the user and the car. The server will run different modules, i.e., the *booking* module, the *billing* module and the *car communication* module. Each module and its corresponding functionality will be explained in Section 2.2.

An overview of the whole system is depicted in Figure 1. The car will need to send/receive messages to/from the server. For these communications we prefer to use 4G technology, because we have no guarantee that the car has Wi-Fi available and the coverage of 4G is bigger. The user will need to communicate, by using our smartphone application, with the car to identify himself. This communication will happen via NFC. We assume that modern smartphones have an NFC-chip.

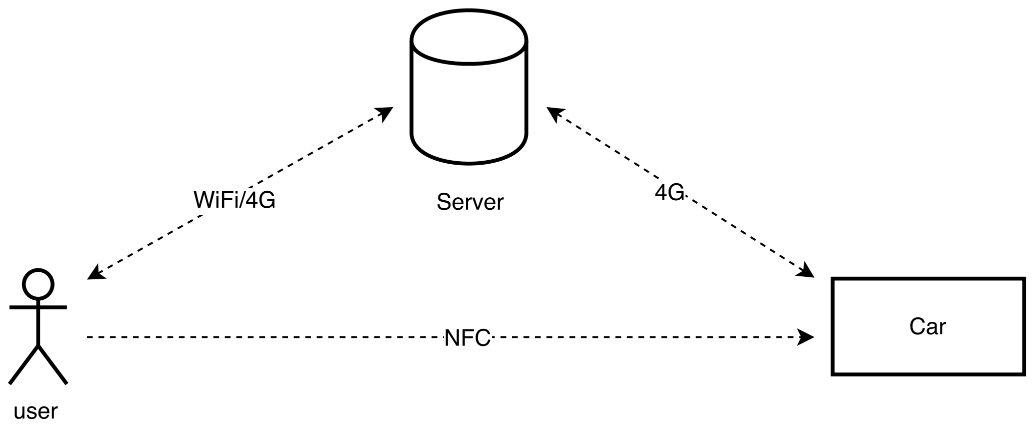

Figure 1: General overview of the architecture.

2.2 Server modules

The server consists of following modules:

- *Booking module*: responsible for booking a car, based on communication between the smartphone application and the server.
- *Car communication module*: a part of the server that communicates with the cars.
- *Billing module*: the part of the server that calculates billing details and sends the bills to the users.

These modules were developed separated to increase modularity, so that eventually a module can be moved to other servers. Also, the server could be configured to assign computation power to the module that needs it the most (and for example idling the billing module since its activities don't have to be performed in real-time). In our prototype, all modules will run on the same simulated server, so this server has one private-public key pair. We assume that the public key of the server is available to the cars and the smartphone applications. This can be done by simply hard-coding it in the software. Should the public key be compromised, no security risks will come up because no data can be decrypted without the private key. Also, there's no loss of data-origin guarantee because every message that will be sent to the server will be signed with either the private key of the user or the private key of the car.

2.2.1 Booking module

The booking module will be used to receive booking requests from the smartphone application. It will notify the car communication module when a new booking is requested and approved. We assume that these different modules are located on the same server, so that no network hacking can harm the connection between the modules.

In this module, we focus on authentication and confidentiality. We have to be sure that nobody can issue a booking request for another user. The request should also be received untampered at the server and no man in the middle should be able to read the actual contents of the booking request.

2.2.2 Car communication module

The car communication module will be used to send booking information to the car and to receive billing information from the car. This module will ensure that the communication between the car and the server is secure, since this communication will happen via an untrusted and open network. This module will receive information from the booking module and send the billing information to the billing module. We assume that these different modules are located on the same server, so that no network hacking can harm the connection between the modules.

The first security service the car communication module has to implement is confidentiality. If the messages are not encrypted an attacker could eavesdrop the network and read the messages. To encrypt our messages we will make use of AES encryption. The car or the car communication module will generate an AES key before starting the communication. This key will be exchanged before the communication starts. The second security service we have inspected is authentication, since the information send between car and *car communication module* is sensitive and we want to prevent a replay attack. Each message will be signed with the private key of the sender.

2.2.3 Billing module

The billing module is a part of the server that is located on the same server as the *Car communication module*.

Since the communication between the car and the *car communication module* will be secured, and this module is located on the same server as the *billing module*, the billing does not have to take additional security precautions. The billing module is only responsible for calculating the amount to bill to the user.

2.3 Car

The module in the car will have the following tasks:

- Decide if the car can be opened.
- Decide if the car may be started.
- Store billing information in a buffer.
- Store booking information temporarily.

– Communicate with the car communication module.

All cars that are involved in the system will need to be in possession of a smart card. This smart card will be used to store some information, to communicate with the server and to send signals to the car. We decided to use a smart card in the car for the following reasons [**?**]:

1. Authentication: The smart card is embedded by a unique private key which is generated by the manufacturer of the smart card.
2. Strong device security: The smart card is extremely difficult to duplicate. Smart cards are designed in a way that they detect and react to tampering attempts.

So, with the use of the smart card a malicious user is not able to retrieve the private key. Therefore, we can state that the private key of the car is a unique and secure identifier for the car, so the car can use this for authentication purposes.

Since a car needs to be able to communicate with the server, the smart card will be integrated with a SIM-card. This allows the communication to occur via a mobile network. A cellular network has large coverage but in some places there is no network available, for instance in a parking lot. Therefore the car needs to have some memory available, where he can store information temporarily. For instance, the booking information will be send to the car a few days before the booking and when the trip is finished the car will calculate the billing information and store it in memory until he is able to send it to the server. To store some information the smart card will need 28 bytes per booking entry and 32 bytes per billing entry. If we assume that the server is sending the booking information only 5 days upfront and a car is booked for 5 days every hour, then this information is 3360 bytes. The billing information will be calculated after the trip and will be send to the server when a network is available. We assume that during the next trip, the car will be able to send the information to server, so at most one billing entry will be stored on the smart card. We can conclude that 4 kB of memory is sufficient to store the booking and billing information.

2.4 Smartphone Application

The smartphone application will be responsible for the following tasks:

– Register a user

– Book a car

– Issue a command to the car (start, stop, (un)lock car)

This requires that the smartphone application has an internet connection when it is registering and when it is booking a car. To be able to communicate with the Car it will need to have an NFC connection.

To be able to communicate with the server (for registration and booking), the public key of the server is hard-coded into the application. When we need to commence communication, the application will generate an AES key and encrypt it with the server's public key and send this to the server. The server and the application then both know the AES key and can securely communicate with one another.

To be able to communicate with the car, we do the same thing. During booking, we received the public key of the car. Using this key we do another AES key exchange with the car, as done before with the server. Now that the car and the application both know the same AES key they can securely communicate with one another.

We assume that the smartphone application runs inside of a sandbox and is not able to be hacked from the phone itself (e.g. a virus reads the applications' memory, retrieving an AES key, which can then be send to the creator of the virus which might then be able to open the car at a later date as it has the required key for communication). Further more we assume that the private key and UID of the application remain secret as compromising these would allow a malicious user to set up a secure connection. Several Android implementations exist that can secure internal storage [? ?].

2.5 Use cases

2.5.1 Smartphone-server communication

For the communication between the app and server, we will use a mobile network technology, such as 4G or WiFi. Customers can use this application wherever they want. We make the assumption that packets always reach the server, although it is possible that packets are sniffed and replayed.

When we focus on confidentiality, we should consider man-in-the-middle attacks. An example of this sort of attacks is eavesdropping where packets are intercepted by the attacker. To prevent this attack, all packets need to be encrypted. If the message can be intercepted by the attacker, he can not retrieve the actual contents. Booking data will be encrypted using AES, which will use a key that is uniquely generated by the application for each transmission. As the application will generate the AES key, a key exchange process is compulsory. For this purpose, RSA will be used.

2.5.2 Registration

When a customer downloads the application, a registration is necessary. During the registration, the customer will be assigned a userID and his/her driver's license will be validated. The server's public key will be hardcoded in the smartphone application. This imposes no security risk: only the server knows his private key and everyone may send encrypted data to the server. If the server's private key would become compromised, it is sufficient to roll-out a mandatory application update with the new public key built-in.

To start the registration, an AES key will be generated by the application. The AES key will then be encrypted using the server's public key. This message will be send to the server. For this purpose, no measures against replay attacks are necessary. Everyone may issue a registration request to the server and no user-specific data is included in this message. If the server receives this message, it will decrypt it using its own private key and generate a userID. This userID will be signed using the server's private RSA key. PKCS1-1.5 is used in this encryption to enforce structure. This way, the application can verify the identity of the server by decrypting the message using the AES and checking the PKCS structure.

To be able to perform a secure AES key exchange for an existing user, the server needs to know the application's public key to verify the identity in the key exchange process. For this purpose, the application will generate a new private/public key pair one time during the registration. The public key will be sent using the AES key, generated in the first step of this process. Also the userID will be added to the message, as the server has to look up the corresponding AES key to decrypt the message containing the application's public key.

From now on, we can issue an AES key exchange for each transmission, to ensure freshness of the AES key. This process is discussed in Section 2.5.3. After the key is negotiated, the registration info, including the driver license, will be send to the server using AES encryption. Here again PKCS1-1.5 will be used. However, the server needs to determine which AES key to use, as it has one for every customer. For this purpose, the userID will also be send. This will be done using RSA encryption, to prevent disclosure or adjustments by an attacker, where he would try to assign the registration info to another user. The attempt can for instance be to link a valid driver's license to a user that doesn't have one. When the server receives the registration info, the existing AES key will be erased from the temporary dictionary.

2.5.3 Key Exchange

The key exchange process is shown in Figure 3. To start the process, an AES key is generated by the smartphone application. The AES key is then encrypted using the app's private RSA key. PKCS1-1.5 is used in this encryption to enforce structure. This way the server can verify the identity of the application by decrypting the the AES key and checking the PKCS structure. For this purpose, the server also needs the userID, which will also be added to the message. Next to the userID and the AES key, also a timestamp will be included in the message. The reason for this, is to prevent replay attacks, where the attacker re-sends old valid messages to the server. Using timestamps, one can validate the freshness of the message by checking the timestamp against its own clock. When the timestamp, adjusted with some possible skew, differs to much from the server time, the attack is detected. The whole message is encrypted using the server's public key, so the server is the only entity that can read the actual content of the message by decrypting it with its own private key.

When the server receives the message, it will decrypt it using its own private key and check the timestamp. The server will then look up the app's public key, that it has received during the registration process. For this purpose, the userID is used, as the public keys are stored in a dictionary where userIDs are mapped on public keys. It will then decrypt the AES key and verify the PKCS structure to verify the app's identity. If this is successful, the AES key will be linked to the corresponding user ID in a temporary dictionary.

The AES key will be used by the application to send data to the server. When the server receives this data, the AES key will be erased from the dictionary and a new key exchange is necessary. The AES key is valid for only one data transfer. The advantage is that if the AES key would become compromised, this forms no security risk. The disadvantage is that a key exchange is necessary for each data transfer. As we assume that bookings are done at most a few times a day, this overhead is negligible. However, this AES key could even be reused for several days without imposing a serious security risk.

We have taken appropriate measures against replay attacks, using timestamps, and eavesdropping, as the message is encrypted using the server's public key. Hackers may try to assign fake AES keys to victims by sending fake request. However, since we have signed the AES key using the app's private key, these attacks will be detected. Attackers don't possess the private key of the application and the message is structured using the PKCS scheme.

If a request fails, the server will send no error messages back to the transmitter to prevent information leakage.

2.5.4 Booking

A booking request will always start with an AES key exchange, discussed in Section 2.5.3. After the key is negotiated, the booking info will be send to the server using AES encryption. Here again PKCS1-1.5 will be used. However, the server needs to determine which AES key to use, as it has one for every customer. For this purpose, also the userID will be sent. This will be done using RSA encryption, to prevent disclosure or adjustments by an attacker, where he would try to assign the booking request to another user.

By using AES encryption, with an AES key that is only known by the application and the server, we can ensure non-repudiation. Only the client can send data using the correct userID and AES key combination. If the attacker sends malicious requests to the server, the AES encrypted data can not be decrypted as it doesn't posses the right AES key. We have also taken measures against replay attacks by adding timestamps to the messages. When the timestamp, adjusted with some possible skew, differs too much from the server time, the attack is detected.

After the booking module confirmed the booking it will sends the booking info back to the smartphone application, for viewing purposes, and because the smartphone needs the booking id and public key of the car to be able to successfully communicate with the car as is explained in Section 2.5.5. This information is once again exchanged by encryption with an AES key that has been obtained from a new key exchange between the server and the smartphone application.

The carcommunication module gets a notification of a new booking from the booking module and will buffer this information. When a network is available to the car, the carcommunication module sends an update of the new bookings. First a key exchange needs to be performed, to negotiate an AES key. This process is analogue to the process shown in Figure 3, but acts between the car and the car communication module. After the key exchange the car communication module sends the booking info encrypted with the AES key to the car. The car communication module will also add a timestamp in the message to prevent a replay attack.

2.5.5 (Un)locking and starting the car

This use case illustrates multiple actions as the security measures needed for the actions are the same. When a user wants to (un)lock or start a car, we have to be certain that he is the legitimate user who booked the car, so authentication is needed. Next to authentication, confidentiality is needed to prevent eavesdropping (even though this is not the easiest task in the case of NFC), especially to be able to support technologies other than NFC in the future. Even though eavesdropping has been prevented by confidentiality, we also have to protect against replay attacks by using a timestamp or sequence number. A sequence diagram depicting the interactions in this usecase can be found in Figure 5.

The car has already received the booking information through the *car communication module* (see Section 2.2.2), this information includes the public key of the smartphone application and the time interval during which the user is able to use the car. The smartphone application acquired the public key of the car during the booking (see Section 2.5.4).

When the user issues a start, stop or (un)lock command (in Step 1), an AES key exchange between the smartphone and the car is started over NFC (see Step 2). To be certain that no one other than the legitimate user sets up a secure connection, the exchange contains a timestamp that is encrypted with the smartphone app's private key. This ensures authenticity, confidentiality and protects against replay attacks.

We then create a message that contains the command appended by the booking id (to verify booking information) and a timestamp to prevent replay attacks, this is what you see in Step 3. We do not sign this message because we know you are the legitimate user due to the authentication that happened in the AES key exchange. If you did not adhere to authentication during this key exchange, you could not have set up this secure connection. In Step 4 this message is then encrypted with the AES key to prevent eavesdropping. The message is then send to the car in Step 5.

When the car receives the message it will decrypt the message using the AES key it received in the key exchange with the smartphone application, as shown in Step 6. The car then checks if the timestamp is correct (within 100ms of the current time). Finally, it will check if the booking id corresponds with the currently active booking. In case everything is correct, the command will be executed as depicted in Step 7.

In case the car is locked, the billing will be processed by the *billing module*, as is explained in the next section, Section 2.5.6.

2.5.6 Billing

As discussed in Section 2.5.5, every time the user locks a car, a command is sent to the car. If the car detects that the command was to lock the car, the *billing* use case is initiated, which is briefly discussed in this section and depicted in Figure 6.

Once the car receives the command to lock the car, it stores data about the finished trip (travelled distance, duration, etc.) in step 1. In step 2, a key exchange between the car and the *car communication module* on the server is initiated, so that they both have an AES key that can be used for the encryption of the transfer of the billing data (trip data: travelled distance, duration, etc.). This AES key exchange is similar to that described in Section 2.5.4: the only difference is that the Car UID is used instead of a User UID. Once both parties (the car and the server) have knowledge of the AES key, the trip data, along with a timestamp and the Car UID is transferred in step 5. The server uses its stored AES key to decrypt the message and checks whether the message is still valid with the timestamp. If this can be verified, the billing information is passed to the *billing module*, which processes the bill and sends it to the user via mail.

2.6 Technical decisions

2.6.1 Keylengths

For RSA, a key length of 2048 bits is used, which allows for 112 bits of security strength. A recent study of the US National Institute of Standards and Technology (NIST) [**?**] (January 2016) indicates that this security strength will still be safe through 2030. Encryption of new data from

2031 and beyond however, will be disallowed to use a security strength of 112. Instead, a security strength of 128 bits (RSA 3072) should be used from 2031 and beyond. We conclude that a compatibility until 2030 is sufficiently long, especially for a prototype of a system that is currently not common on the market of car security. Hardware implementations in the car will probably have changed earlier than 2030.

In case the key length should be changed, the proposed proof-of-concept system will not work anymore. Not only the server software and smartphone application will have to be updated, the smartcards inside the cars will have to be replaced. With ease-of-use in mind, the proof-of-concept system can be adapted to detect version changes such that cars with old smartcards will automatically stop working when the key length is updated, or the smartphone application will force the user to update the application for further usage.

The used library for AES key generation (the Java Crypto library) uses a default AES key length of 128 bits. According to [**?**], an AES key length of 128 bits will even be acceptable after 2030.

Of course, when stating that the proposed key length of 2048 bits is sufficiently strong until 2030, we assume no quantum computer algorithms are used. If that should be the case, a length of 2048 is probably not safe until 2030. Especially the RSA keys will be vulnerable and will have to be extended. However, although some algorithms already exist for quantum computing attacks, probably computers capable of running such algorithms on 2048-bit keys will not exist in the upcoming years.

2.6.2 Performance

Performance of the proposed system design and its corresponding proof-of-concept implementation is an important quality measures since it can directly influence the availability and ease-of-use of the system. A user does not want to wait long before opening a car. In this section, a brief discussion is made on the performance of the proposed system.

The use case where a lack of performance would be experienced the most by a user is when opening or closing a car. One intuitively expects the car to immediately respond to a command from the smartphone — just like the car immediately responds to a signal sent from a classical (wireless) car key. A delay is however inevitable according to the requirements of the system: the system must verify that the person trying to interact with the car has the right permissions to do so; and that commands received from that person actually origin from that user. This delay is caused by encryption and decryption of the commands sent from the smartphone application to the car. We minimize this delay in our system design by letting the car known upfront which user the car might expect to interact with the car. In addition, the smartphone application knows upfront to which car he is allowed. Because of this, only an AES key exchange and AES encryption and decryption has to be done when locking or unlocking a car.

The use case of billing has no direct impact on the user experience, but if this part of the system requires a lot of computation time, it might affect the *car communication module* on the server. An improvement on the proposed system can be made that causes bills only to be sent at the end of the day, or at night for example. In this case, when locking a car, the car will not send the details of the trip to the server immediately, but will instead store it temporarily. A disadvantage of this modification would be that the user possibly has to wait for his bill. In case the billing is done via e-mail for example, the user does not directly gets a bill for the trip he has just made, but has to wait until the car decides to send the trip data to the server.

3 Proof-of-concept application

A proof-of-concept implementation was made based on the previously discussed architecture and technical decisions. We chose to use a Java console application, so that we could focus on implementation of the security mechanisms, rather than designing a UI. In addition, the Java language has great libraries for security algorithms that we could use in our prototype. The UML diagram of the prototype is depicted in Figure 7.

After running the console application, one can execute *commands* on it to simulate a use case. Consider for example the following use case:

1. A user registers on the system.
2. The user retrieves the list of cars, chooses one and books the car.
3. The user tries to open another car – access is denied.
4. The user opens and starts the car he has booked.
5. The user stops and locks the car.
6. The user is billed for the time and distance travelled with his car.

Execution of this use case in the console application is shown in Listing 1. For the convenience, all lengthy keys are truncated. Before each command, the app prints which commands are available in the app. On the next line, the user entered a command in this example. All content after this command is output of the app, ended by an empty line.

```
> java -jar app.jar
Commands: quit; register; list cars; (book|open|close|start|stop) car <carUid>
register
APP: Generated AES key: E48...
SmartphoneApp -> Server: Send msg pub_server(E48)
Server generated 0
Server -> SmartphoneApp: Send msg AES(priv_server(0))
SmartphoneApp received UID: 0
SmartphoneApp -> Server: Send msg pub_server(0)||AES(308...)
Server received 308...
Server has stored the app's public key for 0, registration is now complete

Commands: quit; register; list cars; (book|open|close|start|stop) car <carUid>
list cars
Available cars: [1, 2, 3]

Commands: quit; register; list cars; (book|open|close|start|stop) car <carUid>
book car 1
APP: Generated AES key: 9F5...
SERVER: Received AES key for user 0: 9F5...
BOOKINGMODULE: Has decrypted the bookinginfo with the temp AES key: user 0 has booked car 1
    from Thu May 12 17:14:48 CEST 2016 to Fri May 13 21:01:28 CEST 2016
CARCOMMUNICATIONMODULE: Generated AES key: 008...
CARCOMMUNICATIONMODULE: Send (encrypted with public key): 2D8...
CAR: Decrypted message with public key: 000...
CAR: Message was: 008...
CARCOMMUNICATIONMODULE: message: [B@74ad1f1f
CARCOMMUNICATIONMODULE: send (encrypted with AES): 581...
CAR: Decrypted message with AES key: 000...
CAR: Message was: 7B2...
--
APP: Car 1 has been booked.

Commands: quit; register; list cars; (book|open|close|start|stop) car <carUid>
open car 2

Commands: quit; register; list cars; (book|open|close|start|stop) car <carUid>
```

```
ERROR -- You're messing with car 2 while the booking information suggests you're only allowed
    to car 1
open car 1
APP: Generated AES key: 141...
APP: Encrypting AES with public key.
CAR: Decrypted message with public key: 000...
CAR: Message was: 141...
APP: Encrypting message with AES key.
APP: Sending message to car.
CAR: Decrypted message with AES key: 000...
CAR: Message was: 756E6C
CAR: User with UID=0 executed command=unl
Car 1 has been unlocked (booking UID 0).

Commands: quit; register; list cars; (book|open|close|start|stop) car <carUid>
start car 1
APP: Generated AES key: A47...
APP: Encrypting AES with public key.
CAR: Decrypted message with public key: 000...
CAR: Message was: A47...
APP: Encrypting message with AES key.
APP: Sending message to car.
CAR: Decrypted message with AES key: 000...
CAR: Message was: 737461
CAR: User with UID=0 executed command=sta
Car 1 has been started (booking UID 0).

Commands: quit; register; list cars; (book|open|close|start|stop) car <carUid>
stop car 1
APP: Generated AES key: C4A...
APP: Encrypting AES with public key.
CAR: Decrypted message with public key: 000...
CAR: Message was: C4A...
APP: Encrypting message with AES key.
APP: Sending message to car.
CAR: Decrypted message with AES key: 000...
CAR: Message was: 73746F
CAR: User with UID=0 executed command=sto
Car 1 has been stopped (booking UID 0).

Commands: quit; register; list cars; (book|open|close|start|stop) car <carUid>
close car 1
APP: Generated AES key: 811...
APP: Encrypting AES with public key.
CAR: Decrypted message with public key: 000...
CAR: Message was: 811...
APP: Encrypting message with AES key.
APP: Sending message to car.
CAR: Decrypted message with AES key: 000...
CAR: Message was: 6C6F63
CAR: User with UID=0 executed command=loc
CAR: Generated AES key: 400...
CAR: Send (encrypted with public key): 747...
CARCOMMUNICATIONMODULE: Decrypted (public key) message: 000...
CARCOMMUNICATIONMODULE: Message was: 400...
CARCOMMUNICATIONMODULE: Decrypted (AES) message: 7B2...
BILLINGMODULE: User with UID 0 billed for the amount of EUR 2.85383700001E12
```

Listing 1: Example execution

4 Conclusion

This assignment focussed on the design of several security mechanisms needed to ensure the predefined (non-)functional requirements. The process of designing the use cases and required security features led to a deeper understanding of which security mechanisms should be used in practice — which mechanisms are most appropriate for the respective use cases. Also, the actual implementation of the prototype gave rise to unexpected technical problems that were not noticed during the design, for example with incompatible key lengths. The predefined requirements were met by the designed prototype that can later optionally be implemented in an actual smartphone application.

A Appendix: Figures

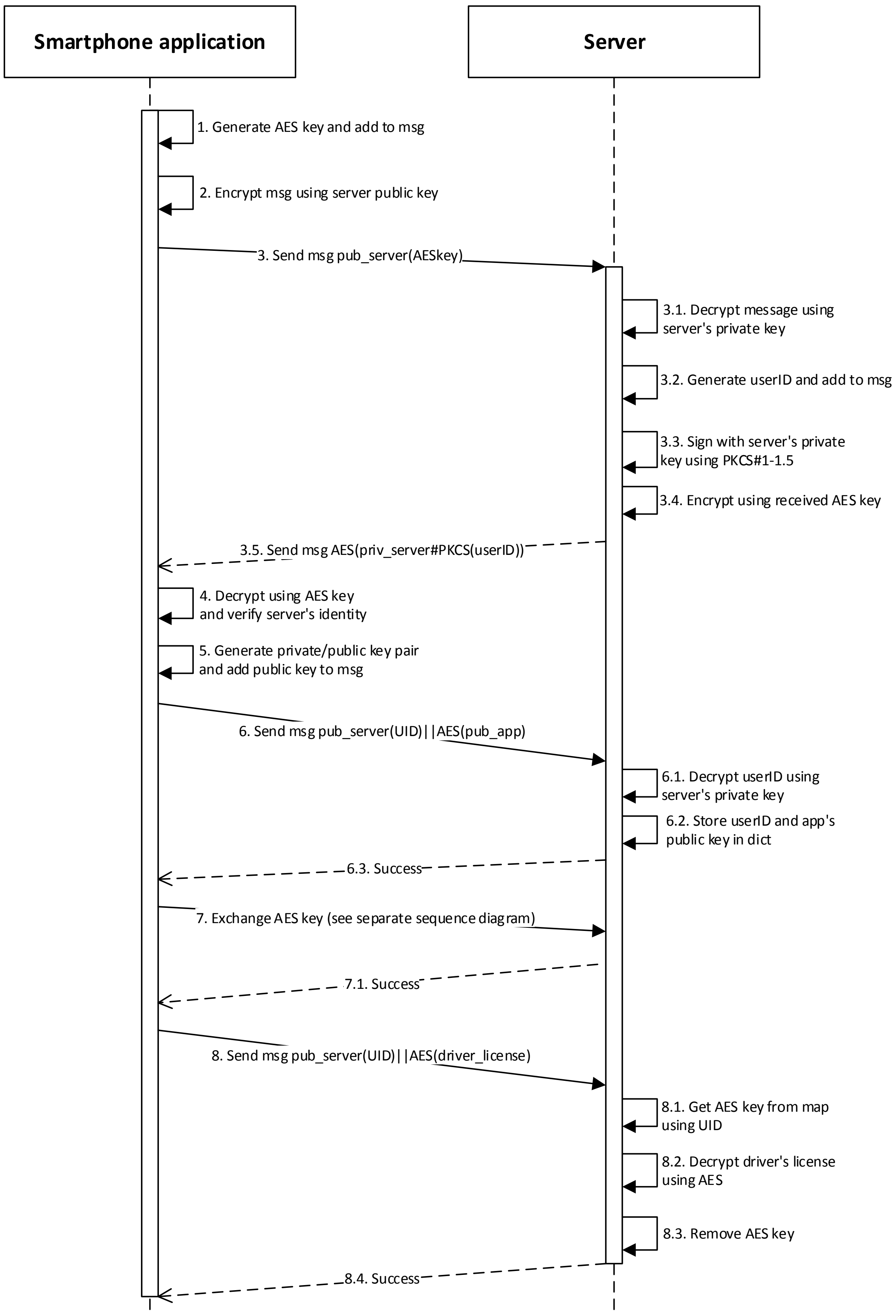
Smartphone application
Server
1. Generate AES key and add to msg
2. Encrypt msg using server public key
3. Send msg pub_server(AESkey)
3.1. Decrypt message using server's private key
3.2. Generate userID and add to msg
3.3. Sign with server's private key using PKCS#1-1.5
3.4. Encrypt using received AES key
3.5. Send msg AES(priv_server#PKCS(userID))
4. Decrypt using AES key and verify server's identity
5. Generate private/public key pair and add public key to msg
6. Send msg pub_server(UID)||AES(pub_app)
6.1. Decrypt userID using server's private key
6.2. Store userID and app's public key in dict
6.3. Success
7. Exchange AES key (see separate sequence diagram)
7.1. Success
8. Send msg pub_server(UID)||AES(driver_license)
8.1. Get AES key from map using UID
8.2. Decrypt driver's license using AES
8.3. Remove AES key
8.4. Success

Figure 2: Flow of the registration of a new user, including driver's license check

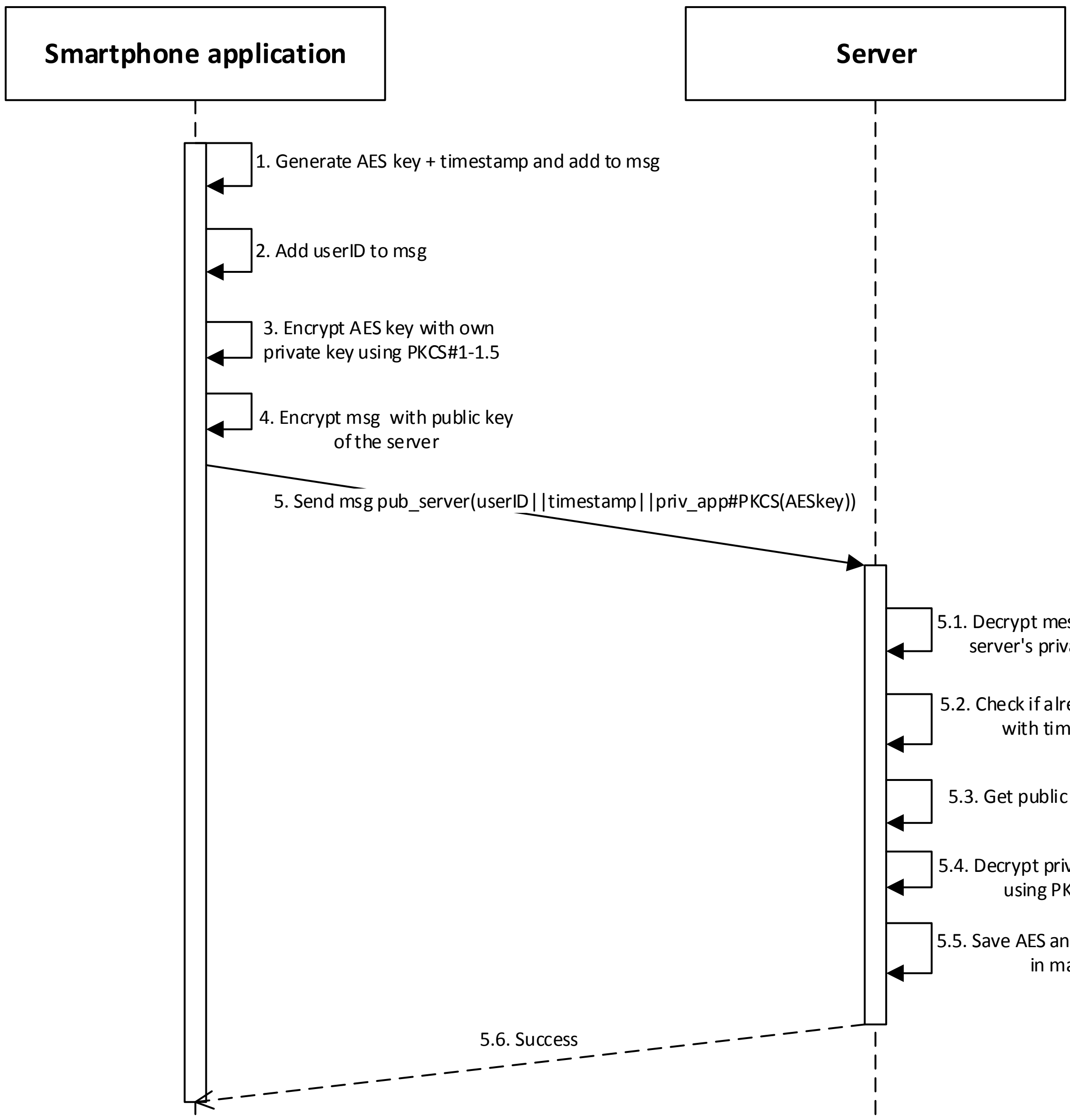
Smartphone application
Server
1. Generate AES key + timestamp and add to msg
2. Add userID to msg
3. Encrypt AES key with own private key using PKCS#1-1.5
4. Encrypt msg with public key of the server
5. Send msg pub_server(userID||timestamp||priv_app#PKCS(AESkey))
5.1. Decrypt message using server's private key
5.2. Check if already processed with timestamp
5.3. Get public key from user
5.4. Decrypt priv_app#PKCS(AES) using PKCS#1-1.5
5.5. Save AES and userID pair in map
5.6. Success

Figure 3: AES key exchange process between the app and the server

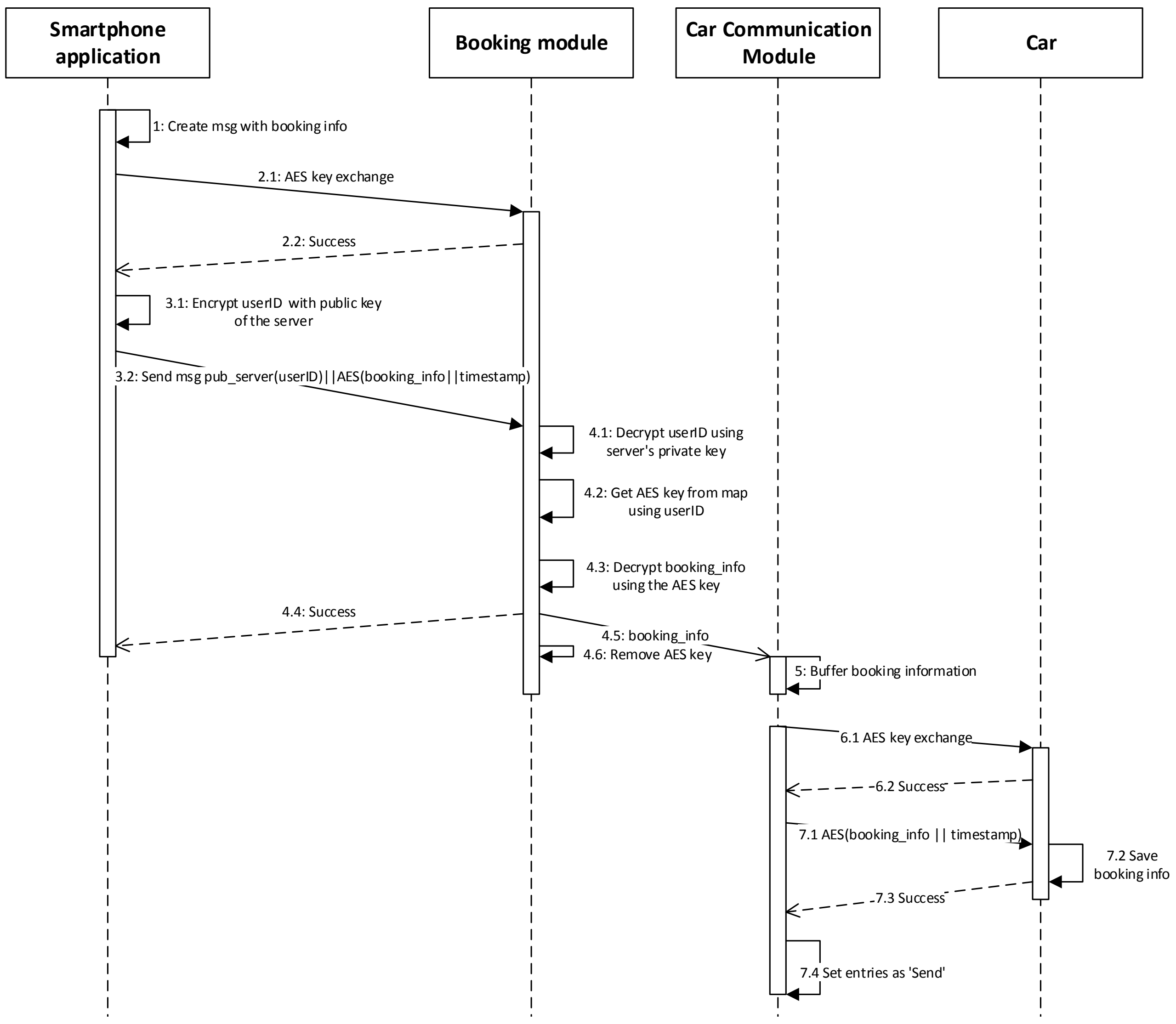
Smartphone application
Booking module
Car Communication Module
Car
1: Create msg with booking info
2.1: AES key exchange
2.2: Success
3.1: Encrypt userID with public key of the server
3.2: Send msg pub_server(userID)||AES(booking_info||timestamp)
4.1: Decrypt userID using server's private key
4.2: Get AES key from map using userID
4.3: Decrypt booking_info using the AES key
4.4: Success
4.5: booking_info
4.6: Remove AES key
5: Buffer booking information
6.1 AES key exchange
6.2 Success
7.1 AES(booking_info || timestamp)
7.2 Save booking info
7.3 Success
7.4 Set entries as 'Send'

Figure 4: Flow of the reservation of a car using the smartphone application

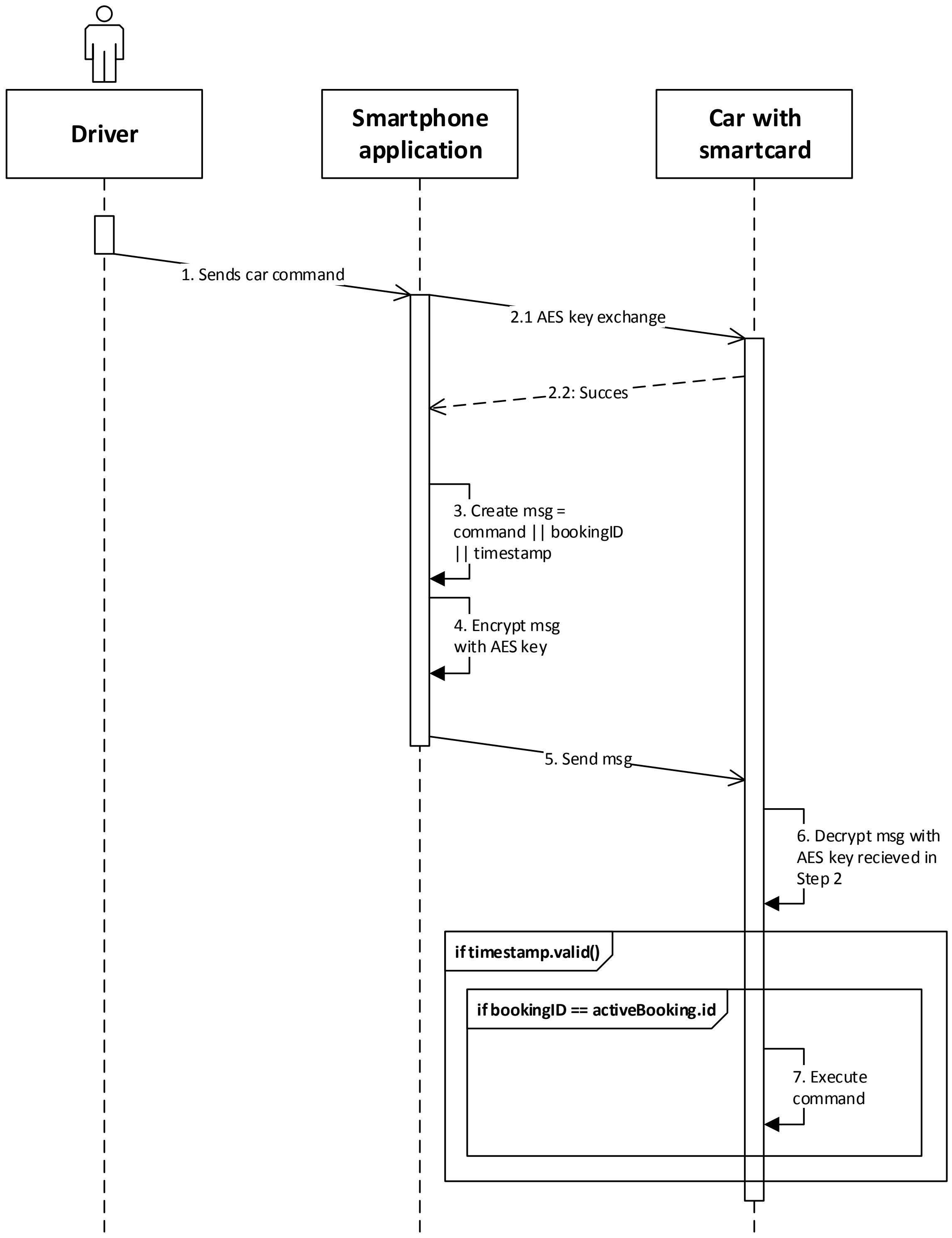
Driver
Smartphone application
Car with smartcard
1. Sends car command
2.1 AES key exchange
2.2: Succes
3. Create msg = command || bookingID || timestamp
4. Encrypt msg with AES key
5. Send msg
6. Decrypt msg with AES key recieved in Step 2
if timestamp.valid()
if bookingID == activeBooking.id
7. Execute command

Figure 5: Sequence diagram of the car commands use case

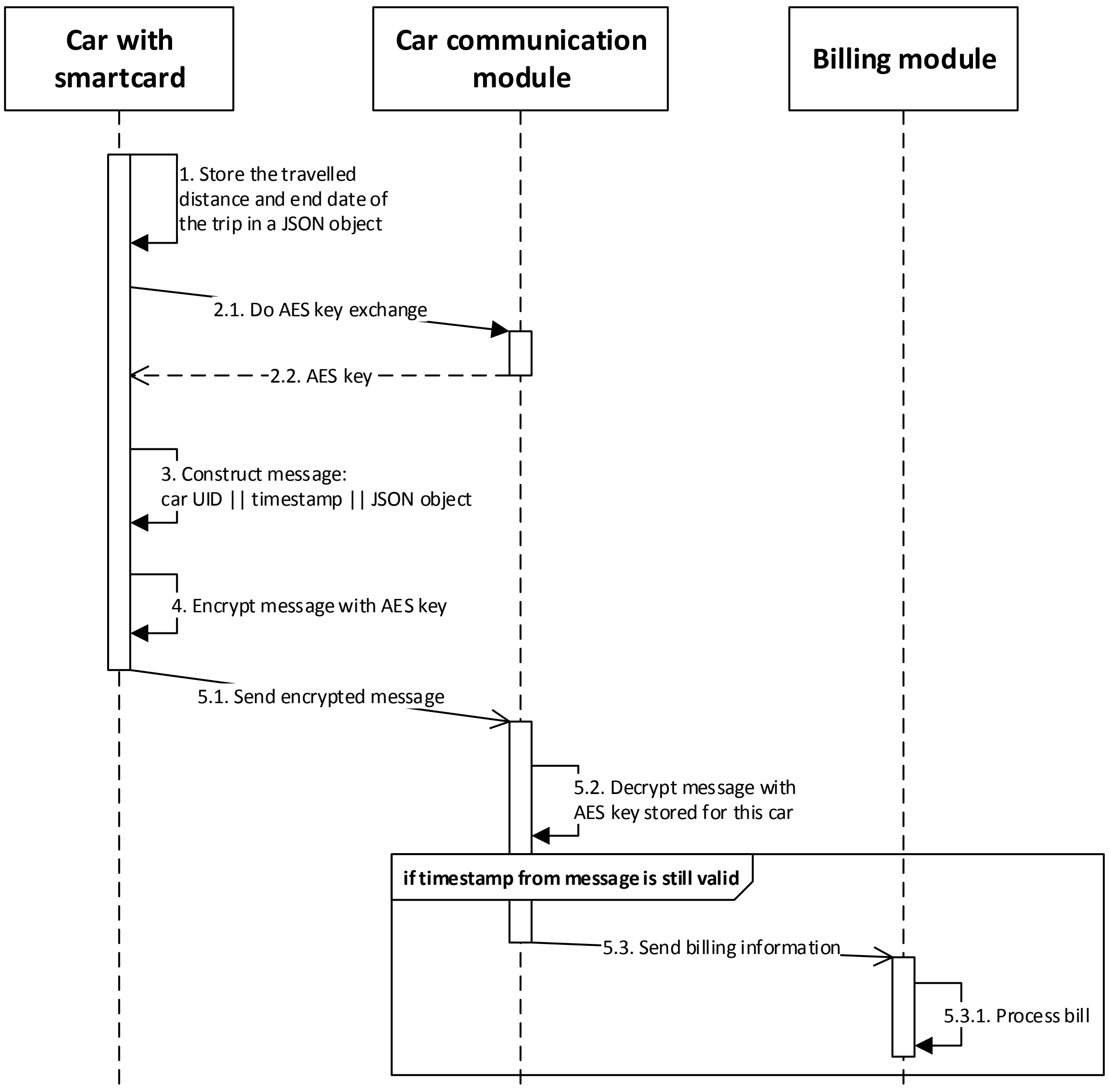
Car with smartcard
Car communication module
Billing module
1. Store the travelled distance and end date of the trip in a JSON object
2.1. Do AES key exchange
2.2. AES key
3. Construct message: car UID || timestamp || JSON object
4. Encrypt message with AES key
5.1. Send encrypted message
5.2. Decrypt message with AES key stored for this car
if timestamp from message is still valid
5.3. Send billing information
5.3.1. Process bill

Figure 6: Sequence diagram of the billing use case

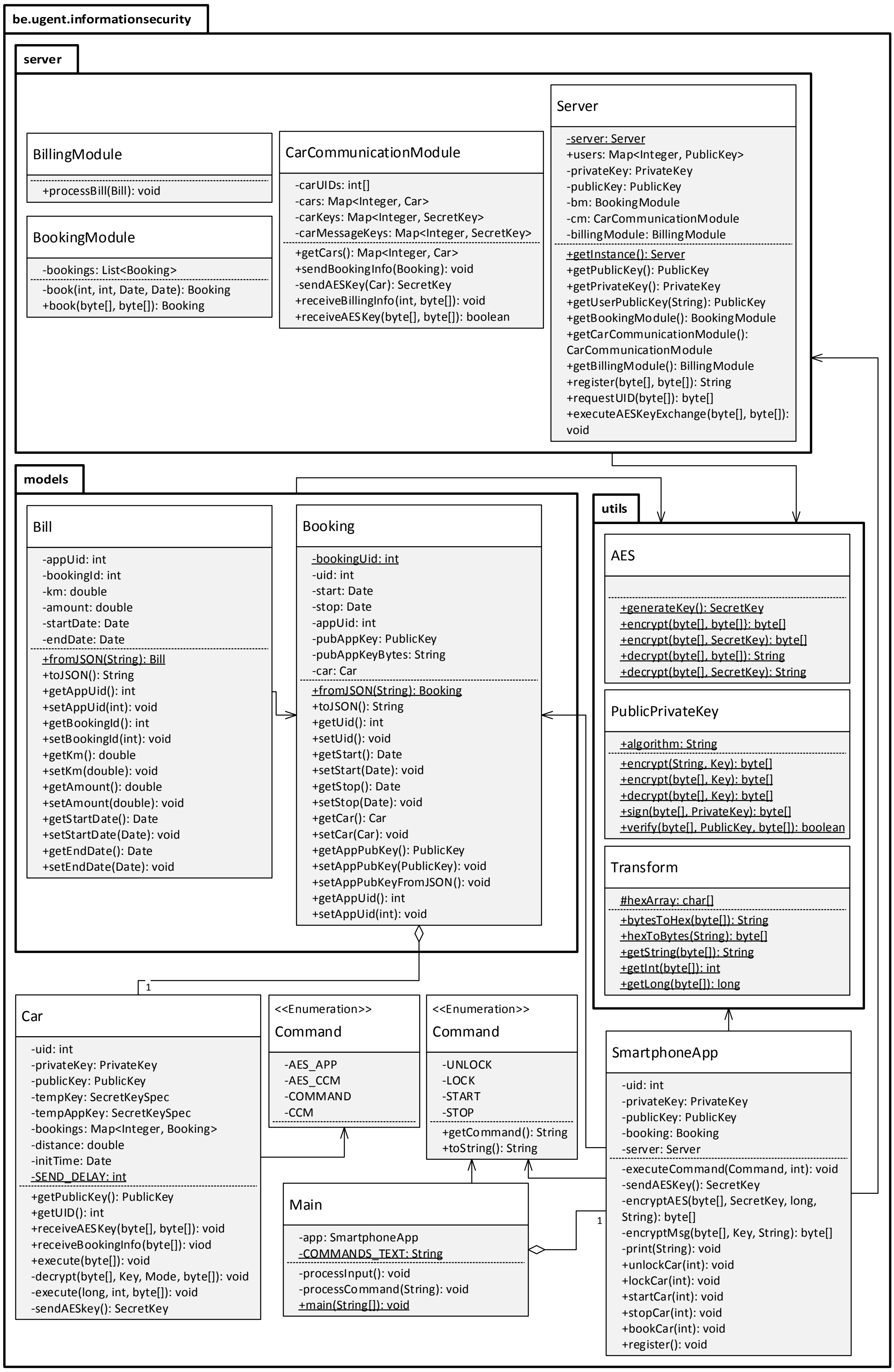

Figure 7: UML class diagram