REPORTS
IN
INFORMATICS

ISSN 0333-3590

End-to-End Secure Solution for Java ME-Based Mobile Data Collection in Low-Budget Settings

Federico Mancini and Khalid A. Mughal and Samson Gejibo and Jørn Klungsøyr and Remi B. Valvik

REPORT NO 401 **September 2011**

Department of Informatics

UNIVERSITY OF BERGEN

Bergen, Norway

This report has URL
http://www.ii.uib.no/publikasjoner/texrap/pdf/2011-401.pdf

Reports in Informatics from Department of Informatics, University of Bergen, Norway, is available at http://www.ii.uib.no/publikasjoner/texrap/.

End-to-End Secure Solution for Java ME-Based Mobile Data Collection in Low-Budget Settings

Federico Mancini Khalid A. Mughal Samson Gejibo

Jørn Klungsøyr, Remi B. Valvik

Department of Informatics
University of Bergen
N-5020 Bergen
Norway

September 30, 2011

Abstract

Lack of infrastructures in health care and transportation, combined with the demand for low cost health services and shortage of medical professionals, are some of the known causes for loss of life in low income countries. mHealth is an emerging and promising mobile-health technology to bridge the gap between remotely and sparsely populated low-income communities and health care providers. Information collected in remote communities can be relayed to local health care centers and from there to the decision makers who are thus empowered to make timely decisions.As sensitive information is stored, exchanged and processed in these systems, issues like privacy, confidentiality, integrity, availability, authentication, non-repudiation and authorization must be given top priority. However, many of these systems do not systematically address very important security issues which are critical when dealing with such sensitive and private information. Well-known and recommended security solutions are ruled out because of specific requirements imposed by mobile platforms (eg. whether HTTPS is available or not) and because of challenges imposed by the working environment in which the system is supposed to function (eg. mobile network coverage).. In this paper, we analyse challenges in deploying Mobile Data Collection Systems where workers in the field collect and transmit data using a mobile client. We propose end-to-end security solutions when inexpensive Java-enabled mobile phones are used for data collection and transmission in remote communities. we aslo analyze implementation challenges of a proposed security protocol based on the Java ME platform. The protocol presents a flexible secure solution that encapsulates data for storage and transmission without requiring significant changes in the existing mobile client application. The secure solution offers a cost-effective way for ensuring data confidentiality, both when stored on the mobile device and when transmitted to the server. In addition, it offers data integrity, off-line and on-line authentication, account and data recovery mechanisms, multi-user management and flexible secure configuration. A prototype of our secure solution has been integrated with openXdata, a Mobile Data Collection System that is primarily designed for data collection using low-end Java-enabled phones in low-budget settings. We use openXdata as our canonical reference for a Mobile Data Collection System.

1 Introduction

Mobile Health (a.k.a. mHealth) is an emerging technology that uses mobile devices to deliver health services and disseminate information to different stakeholders. Disease monitoring and management, clinical trials, data collection, immunizations and education programs are all examples of mHealth services. In this paper, we focused on security aspects of Mobile Data Collection Systems (MDCS). A MDCS allows the collection and transmission of data electronically from different geographical location to centrally located data storage repositories through wireless network. It is combination of a client application running on the mobile devices, wireless infrastructure and remotely accessible server databases. A number of such systems already exist that allow data collection using mobile devices, but we will focus on those used for the collection of health related data. A few such mHealth systems are [3, 6, 16, 17, 18] and most of them share identical principles and guidelines to collect data remotely. An overview of such systems can be found at [15], but we can roughly divide them into two categories: those that provide a client and a server application, but do not provide service [16, 17, 18]; and those that provide client application and a centralized server to store the collected data [2, 6]. In the first case, anyone can set up both client and server applications based on their project requirements and also decide the location of the database. In the latter, one would download the client on the mobile and subscribe to the existing server, which provides data storage. Most systems of the second type offer at least Hypertext Transfer Protocol Secure (HTTPS) [24] for secure communication with the subscribed server, but none of them provides secure storage on the client, or an alternative secure communication option to HTTPS. However, none of these systems has a complete security solution to guarantee data confidentiality, integrity, availability and privacy both on the client and on the server. The process begins by designing a form, which contains a collection of questions. The form is stored in an accessible server database. Collectors can then send HTTP requests to download forms through a wireless network and retrieve forms from the server. The forms are saved in the mobile storage on the client device. The client application uses a presentation layer to display the form in a user-friendly manner on the mobile device. Forms are used by collectors to enter data about each participator and stored in the mobile storage for further action like editing or uploading to the server.

The openXdata system [18] is an open source mobile data collection system that is primarily designed for data collection using low-end Java-enabled phones in low-budget settings. It consists of a server component that provides the tools to set up the studies, design the related forms, manage the users involved in the study and store and analyze the collected data. In the field, the data is collected by collectors which are entrusted with a mobile phone having the openXdata client installed on it. These collectors are given some area of responsibility and are in charge of downloading forms for the current study, filling them with the required data, and uploading the data to the server.

We use the openXdata system as our canonical reference for a Mobile Data Collection System. Our choice of the openXdata system as a reference model is based on the challenges from using low-end Java-enabled mobile phones, having both low battery power and processing capabilities, that are prevalent in low-income countries, and the commitment by the openXdata project to be compliant with international standards and regulatory requirements.

Currently, the openXdata system is used especially in third world countries for collecting data in clinical trials, tracing the progress of large-scale vaccinations or just monitoring student and teacher participation in schools.

In this paper, we analyse challenges in deploying Mobile Data Collection Systems where workers in the field collect and transmit data using a mobile client. We propose end-to-end security solutions when inexpensive Java-enabled mobile phones are used for data collection and transmission in remote communities. We present the challenges in implementing and integrating *openXSecureAPI* (which from now on we will refer to as simply

API) based on secure protocol proposed in [14], that can be used to add security in these systems in a flexible way, without enforcing drastic changes to the existing application.

We focus on the mobile side of the API, which is developed for Java mobile applications (Java paltform, Micro Edition) [19] and assumes that the main use of the application is the collection of data by an authorized user, through predefined forms. In other words we do not consider systems where data is fed by automated systems as sensors. The API is designed by considering several challenges in mobile data collection environment and challenges in Java ME enabled low end mobile phones and offers services such as: server authentication, user registration, secure login also when off-line, encrypted storage, secure upload and download of forms, basic access control, and password and data recovery. Furthermore, it provides two different levels of flexibility. On the first level, developers can choose, for instance, which secure services to integrate in their application (e.g., only storage if they use HTTPS), or which library to use to provide the cryptographic primitives. The second level offers a way to configure the level of security for most services in the application, like the strength of the encryption keys and the passwords, which communication protocol to use to transfer data and the level of protection of specific operations.

For this work we collaborated with openXdata [18], a Mobile Data Collection System that is primarily designed for data collection using low-end Java-enabled phones in low-budget settings. openXdata community shared with us their field experience regarding the deployment of their mobile data collection tools and various technical details of their architecture. This made it possible to evaluate different real world scenarios and come up with a ground set of security and usability requirements for an application dealing with mobile data collection. A high-level protocol satisfying such requirements, while addressing the basic security concerns mentioned previously, has been recently presented by the authors in another work, and it constitutes the foundation of the API design. The implementation itself focused also on more technical issues, keeping in mind the future integration with existing data collection systems.

The overall organization of the API and its usage is discussed in Section 6. The implementation overview are covered in Section 7.5, where, in order to make this article self-contained, we also mention how the different parts of the API reflects the underlying protocol, and which security and usability requirements they address.

Finally, we present some experimental results we obtained by testing a basic data collection client that uses our API, on various mobile devices. For the rest of the article we assume that the reader is familiar with Java ME technology and terminology.

2 Security Review of openXdata

International and national regulatory requirements in security and privacy are the major driving force in making Electronic Health (eHealth) services secure all the time. This is not the case in mHealth, as lack of standards and regulatory requirements create a comfort zone for the application providers to concentrate on the functionality of the system rather than security and privacy requirements. In contrast, openXdata is committed to following eHealth regulatory requirements and accustomed to the mHealth environments. OpenXdata presents a number of security issues that require to be systematically addressed. We present the most critical below.

2.1 User Authentication

The openXdata system uses simple, but potentially vulnerable, basic authentication to identify users. For example, it uses the same username and password to login on the client and on the server. Once the application installation is completed on the mobile client, it is required to login with a valid username and password. After successful login, user login credentials stored on the mobile device. Next time, when the user tries to login, it

will be authenticated against the user table stored in the client store. The user table maps username with password and is stored in the Record Management Store (RMS) without any protection; hence getting access to user accounts will compromise entire system. Besides, authentication credential is send to the server as clear text on an insecure channel. Moreover, the system does not provide any recovery mechanism for lost or forgotten user credentials.

2.2 Data Transfer and Sender Authenticity

The transfer of data between a handheld device and the database (server/desktop/laptop) must be secure, reliable and encrypted; this implies that only data originating from valid sources should be imported to the database. The openXdata system does not implement any security mechanisms to ensure data integrity when data is transmitted.

Data quality is the major comparison factor of electronic data collection with traditional paper-based collection systems. Data may be modified or corrupted when it passes through an insecure channel. In the openXdata system, there is no security mechanism to ensure data integrity. In addition, the openXdata system uses login credentials to make the data collector accountable for the data they send to the system. Based on username and password we cannot make the sender accountable unless otherwise we have other mechanism such as digital signature. Key-management, secure storage, device performance, and cost of certificate are some of challenges in sender authenticity. On the other hand, additional overhead, device performance, low bandwidth and weak connectivity are the major challenges for secure data transfer and integrity.

2.3 Date Storage

In the openXdata system, data is stored unencrypted and can be easily accessible through external connection such as via Bluetooth. Records can be easily copied from the phone and accessed with a simple text editor. encryption is required, and it is required so that it can support a multi user environment.

3 Working Assumptions

There are some security concerns that every system dealing with sensitive information should address. In particular, sensitive data should at all-time be protected, both when stored on a device or a server, and when transferred between localities. However, different systems might have different limitations or requirements that can prevent the adoption of standard and well-proven security solutions. Hence, tailored approaches must be created in order to provide the security needed by the system, while fulfilling such additional requirements. By collaborating with the openXdata system, we have been able to use their field experience to analyze a number of typical real-world situations in which their mobile data collection tools are used, and understand how to secure them in the best possible way. We have identified a few critical issues that can influence the design of a secure protocol for mobile data collection, which are presented in this paper.

3.1 Low Budget Projects

Many projects dealing with mobile data collection choose to deploy free open source tools like the openXdata system, also because of low budget constraints. Such projects would out of necessity also deploy mobile phones with low-end specifications that have low computational power and comparatively little memory. This limitation might preclude strong cryptography that is needed to guarantee a high level of security. Besides, a project running

on low budget would benefit from using security solutions that do not entail higher running costs.

3.2 Remote Working Locations

Many of the projects using openXdata are deployed in developing countries like Uganda [27] and Pakistan, where the infrastructure for mobile communication and Internet access are not yet fully developed. Furthermore, much of the data collection might take place in remote locations or isolated villages, where the possibility of transmitting data through a mobile phone might be very limited or even non-existing. This means that most of the data collection is done off-line, and collectors must be able to authenticate themselves on the mobile phones without connecting to the server. Besides, even when some connectivity is available, the overhead due to the use of cryptography should be minimal in order to minimize both the cost of the data transfer and the possibility of transmission failure.

3.3 Phone sharing

Another issue is related to the fact that we cannot assume that each mobile phone is only used by a specific collector. One phone might be shared among collectors, each with their own account, and the same collector might be registered on more than one mobile phone. Phone sharing raises problems regarding privacy and access control, since most of the work is done offline and large amounts of sensitive data can be stored on a phone. Communication using SMS cannot be regarded as private with this constraint, making it difficult to communicate sensitive information to a collector. If email is not available either, it becomes problematic to issue, for example, new passwords.

3.4 Technical Specifications of the Mobile Client

The current openXdata mobile client is developed using J2ME [19], in order to be compatible with any phone that is at least Java-enabled. This means that any security solution proposed must be compatible with any phone on which the openXdata client can run. Unfortunately unlike more advanced and high end technologies as iOs, Blackberry or Android, not all J2ME specifications provide support for cryptographic APIs. There exists a package called SATSA-CRYPTO [20] which provides basic cryptographic services, but it is not supported by many phones yet and it provides only a limited number of primitives. The most common alternative is to use Bouncy Castle [13], which is well-tested and constantly updated, supporting a wide range of cryptographic tools. However, the main drawback of Bouncy Castle is the large memory footprint of its libraries. Our aim is to provide a security solution that has an acceptable memory footprint and is flexible enough to allow a developer to choose implementations other than Bouncy Castle (if available).

Further issues when using cryptography on a mobile phone is the generation of good random keys, since mobile phones do not have good sources of entropy [4], and even if they have, J2ME might lack the necessary libraries to access them and the problem of finding an acceptable compromise between cryptographic strength and available computational power. A more extensive discussion of the security challenges related to J2ME can be found in [12].

3.5 Distributing and configuring the mobile application

If we want to allow a project to configure the security settings of the application, we have an additional problem since this should be done during or after installation on the mobile device. The original application, in fact, cannot be modified by the project using it, in order

to guarantee that the source code and therefore the security solutions in place are not compromised. In other words, the configuration cannot be hard-coded in the application itself, but it should be sent as additional information, and as such vulnerable to modification.

4 Analysis of Available Solutions

4.1 Data Transfer

The Hypertext Transfer Protocol Secure (HTTPS) is a protocol design to transfer Hypertext Transfer Protocol (HTTP) request and response through SSL/TLS [24] and usually the preferred way of securing client-server communication. However, given also the limitations mentioned earlier, we cannot rely on that HTTPS is available.

First of all, not all mobile phones have a good support for HTTPS, or more precisely for certificates, especially low-end devices which are most likely to be used in openXdata-based projects. Usually, the best way to get HTTPS to work smoothly is to buy and use a certificate signed by some well-known Certificate Authority (CA). Even then we cannot be sure that every mobile phone will support that specific CA, since the list of supported CAs is preinstalled by the vendor and is not standardized. Thus more then one certificate might be necessary. This can easily lead to significant extra costs for the project, and not all projects can afford that.

A cheaper alternative might be to use self-signed certificates, but they do not provide the same security level as CA certificates, and moreover such certificates are not accepted by all mobile phones. Additionally, different phone models accept different certificate formats, making the distribution very difficult to handle from the server side. An interesting discussion on the subject can be found in [26].

Regardless of the cost associated with certificates and the problems with their distribution, it is not given that HTTPS should be the preferred solution for a system like openXdata. The HTTPS protocol was designed to be used in large distributed systems where clients need secure communication for transactions with various unknown servers. The protocol is thus based on a long handshake where a client and a server can identify each other and agree on how to communicate. In our case we know in advance both who the server and the users are, hence we can establish in advance how client and server should authenticate each other and how information should be exchanged. This would eliminate the need for the handshake and a secure communication could be achieved with a lighter and faster protocol.

We should also point out that there is no guarantee that the implementation of the SSL protocol [21], on which HTTPS can be based, will be equally secure on all devices [23].

4.2 Secure Storage

Authenticating users, keeping sensitive data in the device storage, providing data access only to authorized user, and handling key management are daily duties of mobile device. The lack of J2ME security APIs for RMS data storage protection is one of the major challenges. External library such as Bounce Castle can provide lightweight cryptographic operations but can also increase the memory foot print of the application on the mobile device. Mobile Information Device Profile (MIDP) 3.0 specification has been released and includes password based encryption (PBE) mechanism for data storage protection. However, there is no MIDP 3.0 supporting phone in the market, and will take some time before such phones become available on the market. Smart phones address some of these security problems, but the challenge remains with low-end phones. For instance, J2ME uses record management store (RMS) to store data. There are many challenges in securing J2ME applications, and we have even more constraints as usual single user application to face. See [5] for an overview.

5 The Secure Protocol

In this section we present the protocol we designed to integrate security in mobile data collection systems, while taking into consideration all the issues presented in the previous section.

The main idea is that the secure operations defined by this protocol should not rely on any particular assumption about the application that is going to use them, except from the fact that the main type of data stored and exchanged is *forms*. In practice this will translate into an API based on the protocol, that can be used by a mobile data collection application to implement a secure layer on top of the existing application layer. Therefore we can think of the protocol in terms of the *secure services* offered to the application, which can be divided in two main types: secure communication services, and secure storage services.

We present those services in details in Section 5.2 and 7.7, respectively. However, before that, we first need to consider the context in which the application is downloaded and installed on the mobile device, in order to understand the security challenges posed by the distribution of the application itself, and therefore why some services even exist.

5.1 Download and installation

As we mentioned in Section 3.5, we assume that an application using an API based on our protocol to implement security, is distributed freely to set up customized projects. However, in the context of J2ME applications, we should assume that the application is signed by its developer, i.e., openXdata. This means that it cannot be modified, and its configuration should take place either manually after installation or automatically through some configuration file distributed together with it. The first option is clearly not recommended, as it leaves too much space for human errors and possible hacking. The second option can be realized by allowing the application user to define the configuration settings in the JAD file as extra attributes. This would allow also a dynamic generation of the settings, that can therefore be customized for each application instance at download time.

Figure 1 shows a typical scenario for the download and installation of a J2ME application.

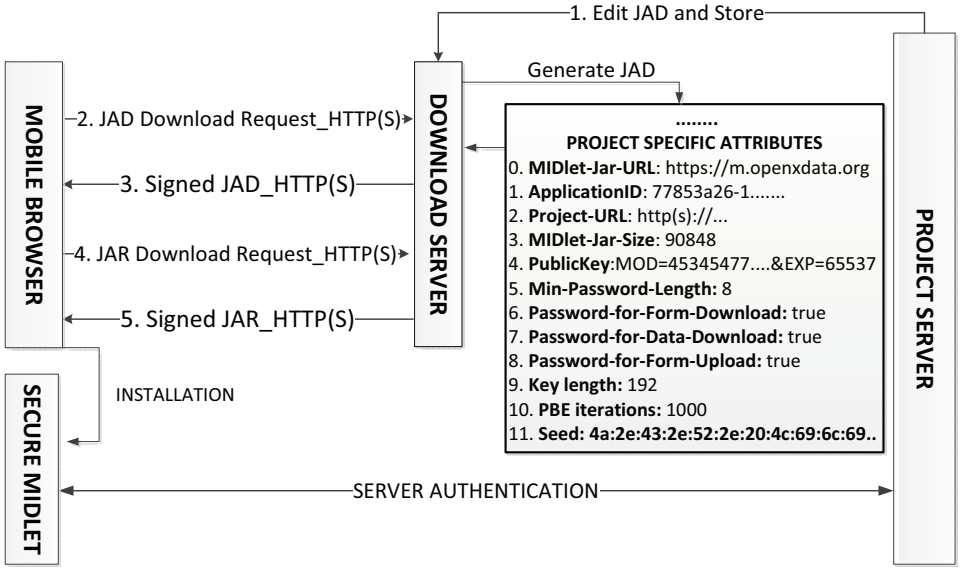


Figure 1: Downlaod and installation.

The problem lies in the fact that, although the application itself is signed and reliable,

this is not the case for the extra attributes in the JAD file. Hence they cannot be trusted unless we can be sure that the JAD file itself was downloaded through a secure channel, and it could not have been compromised after its generation.

The secure channel used for download is not necessary the same used by the mobile client to communicate with the server during the data collection phase. The download server can be, in other words, different from the application server.

To overcome this problem, our protocol offers also a Server Authentication service, that can be used to challenge the server and verify that the settings in the JAD file have not been compromised.

Typical information that can be included in the JAD file is: the server URL, the server public key, an application ID (to uniquely define an application instance and its users, since a user can have multiple account as explained in Section 3.3), the strength of the cryptographic keys and passwords used, and some strong random values to seed the mobile device random numbers generator.

5.2 Secure Communication Services

This sub-section provides an overview of all services that involve some communication between the client and the server.

We can distinguish two types of such services. The ones used to perform authentication operations which might not already be present in the application, as Server Authentication, User Registration and Password Recovery (respectively steps 1, 2 and 3 in Figure 2), and those used to secure the existing communication channel between the client and server application layer. In other words the Download and Upload operations (steps 4 and 5 in Figure 2).

All services have in any case a common aspect, e.g., they have been designed to provide encrypted and authenticated client-server communication even though HTTPS might not be a viable option as discussed in Section **??**.

Leveraging on the fact that security-related data like the public key of the server can be distributed through the JAD file at download time, we are able to use public key encryption to perform user authentication and securely exchange symmetric keys to encrypt the subsequent data traffic. Therefore a typical client request consists of a block encrypted with the server public key (indicated with the notation PK (...) in Figure 2) which contains the type of request (usually a simple string), user name, password, application-Id and a symmetric session key (denoted as SKEY). Optionally a set of blocks relaying the application layer request and encrypted with the session key might be concatenated to first one. In this context we refer to a block as the output of encryption performed by a block cipher as RSA and AES. The response from the server is encrypted with the session key received from the client if the user authentication was successful, or an error is reported. An HMAC is added at the end of every encrypted request/response to make it harder to temper with. With this approach we can implement end-to-end encryption and let both the client authenticate itself to the server though user-name and password, and trust that only the real server can communicate with the client, unless its private key was compromised.

Furthermore, the communication protocol we propose is stateless, that is, every operation requires only one request and one response to be completed. This minimizes the communication time and increases the chance for a successful completion of the transaction, as required by situations like those described in Section 3.2. Any other solution would require the use of session-ids, which would have to be exchanged in clear since, unlike SSL where the whole communication channel is encrypted, we secure only the application data. This could expose the protocol to session-hijacking attacks, reducing security. We make an exception for the Upload and Download operations as explained later.

Not using session-ids might result in possible replay attacks, where a request sent from the client could be re-used to gain unauthorized access to privileged resources, or critical information from the server. To solve this problem we always add a unique serial number

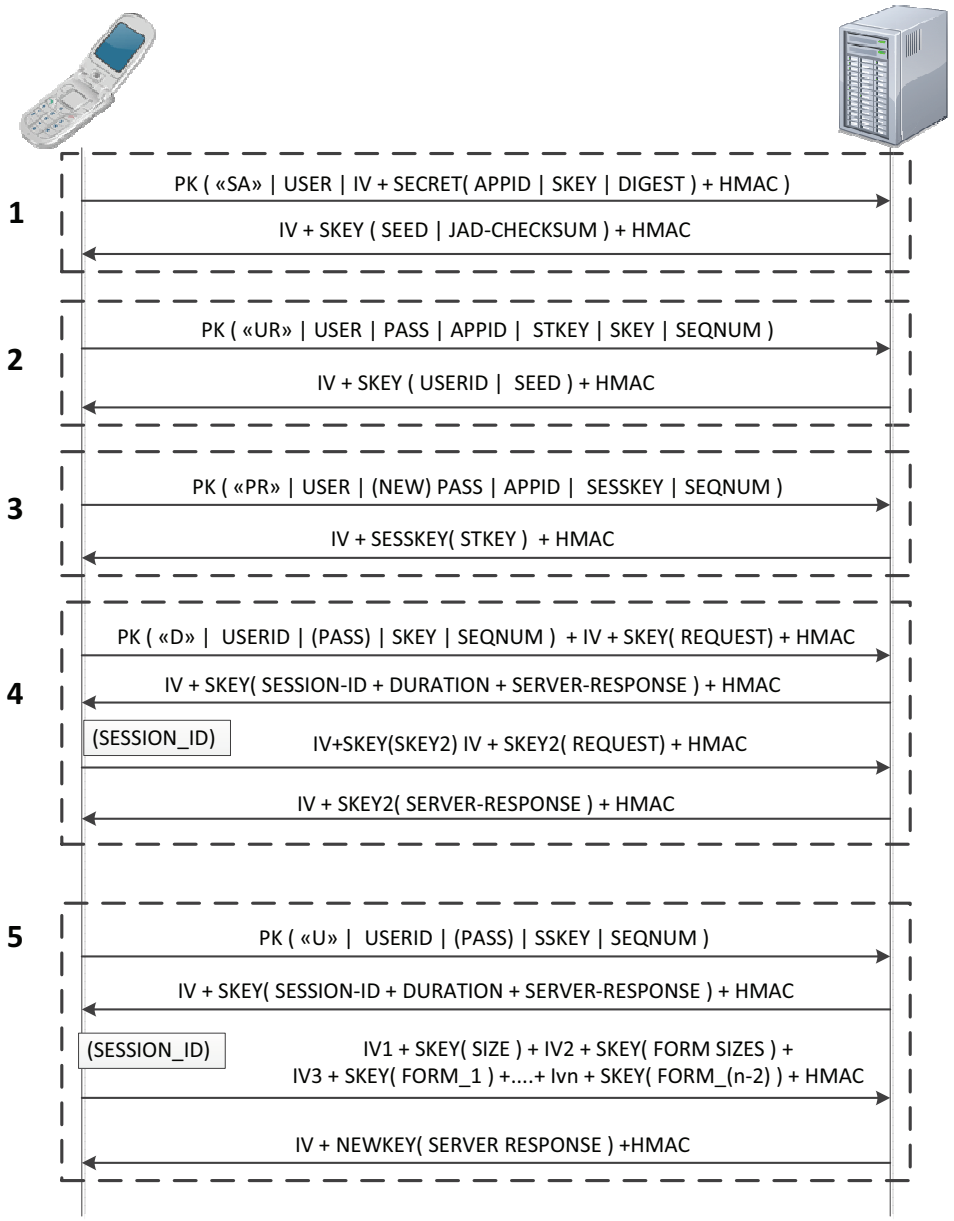


Figure 2: Overview of the storage and login process.

to each request (REQNUM in Figure 2), which is sufficient to make the request unique, and impossible to modify unless the encryption key is broken. As the request number is in practice just an increasing sequential number, in order for the server to easily validate it, it might be possible for an attacker to guess a high enough request number and try and forge a fake request. However, in order for the request to be valid, also some secret information must be provided, as for example the user's password. It would be important in this case, that the server reveals as little information as possible about why the request was not valid.

For symmetric cryptography we propose to use the AES algorithm in CBC mode, and therefore unique initialization vectors (IV) for each new encryption.

We can now describe the single operations shown in Figure 2.

1. **Server Authentication:** If the JAD file is downloaded from a insecure channel, then it needs to be verified before its attributes can be used. At the same time, if the client server communication does not use HTTPS, the server itself must be authenticated. This is done by challenging the server to decrypt a session key (SKEY) encrypted with a shared secret (SECRET) pre-distributed to the users along with their user-name and password. If the server can decrypt the secret key, it can confirm its identity, and will send an encrypted response to the client containing a digest of the original JAD file that was generated at download time. Which JAD file the client downloaded can be identified by the application-Id (APPID), also inside the challenge. A new strong seed is also sent to improve the quality of the keys generated by the random generator on the mobile phone. Note that if the public key had been tempered with by an attacker at download time, a man in the middle, this procedure will still ensure that the response received by the client is genuine, unless the attacker could also break the encryption of the challenge.

2. **User Registration:** A user who wants to use the application, needs to create an account on the mobile phone. Therefore, the first time, remote authentication on the server is required. In this step, besides user name and password, also a storage key (STKEY) that will be used to encrypt the user's data on the phone is transmitted. If the credentials are correct, the server stores the keys, and returns a unique user id (USERID) to identify the user on the specific application instance, to be used instead of the pair user-name/application-Id for further communication. Further seed material is also sent in the response. After this step is completed, the user will be asked to choose a new password, let us call it *mobile password* to distinguish it from the *server password* used for the registration, in order to login on the mobile device.

3. **Password Recovery:** This operation is very similar to the User Registration, with the exception that the storage key is not sent to the server, but retrieved from it. The main idea here is that the server password is used to reset the mobile password, without loosing the encrypted data stored on the mobile device, since a copy of the key used for encryption was saved on the server. Note that user-name and application-Id are used even though the user has a user-Id. The user-Id is, in fact, encrypted and cannot be recovered without the mobile password. More details on the actual recovery procedure on the device, are given in Section 7.7.

4. **Download:** As mentioned earlier, Download and Upload procedures are actually used to encrypt and relay requests from the application layer to the server. Therefore, besides the usual authentication block, there is always also a set of blocks encrypted with the session key that contains the actual client request being relayed. After the first request, a session-Id is used for further communication, in order to avoid repeating public key encryption on the mobile, which is a very expensive operation from a computation point of view. Although we said that this could involve some security risks, we have also to consider that hijacking this session (which still require to break the session key, besides stealing the session-Id), does not allow an attacker to

perform very dangerous actions. In fact, in the context of data collection, downloads consist often of empty form definitions. Also in the case of upload, the session can be used only to store valid completed forms on the server, but not to gain access to data stored on it (unless some injection is performed and the server is vulnerable to it). For the same reason the password field in the first block of the request is optional, as the user-Id is just as secret as any other information stored encrypted on the device. This can be an advantage for the user, who does not need to remember the server password after registration unless the mobile password is lost, in which case the server password is used for the recovery procedure. However, if the device is compromised, an attacker might be able to perform download and upload operations, if the password is not required, as all the necessary information would be available (encrypted) on the device.

5. **Upload:** The upload process is in principal identical to the download, with the exception of the client request format. In fact, unlike the Download case, an upload request might be very large, as it consists of potentially many completed forms. When considering that most of the collection is done off-line over long periods of time as discussed in Section 3.2, a lot of data might have been accumulated before any opportunity to upload it comes along. For this reason, we optimize the process by storing the form on the mobile device already encrypted with the session keys used for the communication. This reduces the cryptographic burden from the upload operation, as the encryption is done separately for each form when it is completed and stored rather than for all the forms just before upload. The forms can be simply read from the local storage and written on the communication channel. One problem is that the server will not be able to parse the forms as it sees only a stream of encrypted information. Therefore we add special blocks before the forms at upload time. One block of fix size containing the size of the next set of blocks, which in turn contain the list of sizes of the encrypted forms (in other words their offsets in the stream). These blocks are also encrypted with the session key in order to avoid revealing any information to a potential attacker.

5.3 Secure Storage Services

Security issues regarding offline usage of the mobile device dictates how data must be stored on it. These include how to authenticate a user locally on the mobile phone, how to keep the stored data secured and how to recover passwords and data if user credentials are lost. Figure 3 shows an overview of the storage architecture in an application using the API based on the protocol. The different services are explained in detail below.

1. **Data Storage:** The mobile application can require encryption and decryption services to the API. The API, given some data in the form of a byte array, will return the encrypted data encapsulated in an object, together with meta information which is available without having to perform any decryption first. The protocol does not make any assumption on where the encrypted data will be stored afterwards, hence remaining general and independent from the underlying application.

2. **Login:** After the User Registration operation, the user is asked to choose a password to log in on the mobile phone (the mobile password). This password is used to generate a key through a Password Based Encryption (PBE) scheme [11], which is in turn used to encrypt the storage key of the user on the phone. The storage key itself, instead, is used to encrypt all the other data of the user. In Figure 3, we can see that all the cryptographic information of the user created after registration are stored in a separate store to which the application has no access. The login procedure, described in Figure 3, is based on the successful decryption of the storage key. A key is generated from the password using the PBE scheme and the salt created at

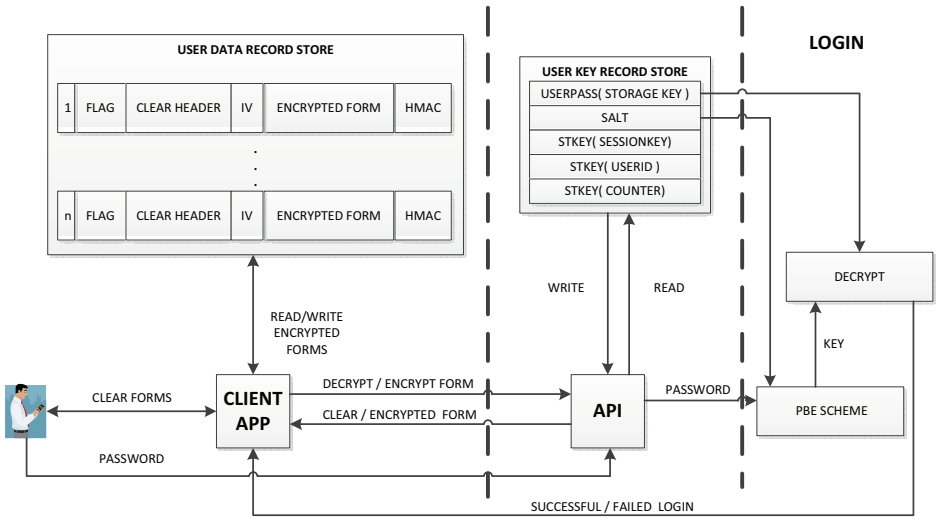


Figure 3: Overview of the storage and login process.

the registration step, and used to decrypt the storage key. Since the storage key is encrypted together with its digest, it is possible to check whether the decryption was successful, and therefore authenticate the password. Using two passwords solves some of the issues discussed in Section 3.3. It reduces the risk that if the mobile or the user password is lost, the access to the server is compromised. In fact, the server password is not stored in any form on the mobile. This limits the damages of a successful brute force attack on the device, as only the data stored on the phone are compromised, but not those on the server.

3. **Recovering an account:** As we have seen in the previous section, if the password is lost, users can still authenticate themselves on the server and retrieve the storage key that was stored there at the User Registration step. The storage key can then be encrypted with a new password on the mobile phone, and replace the one encrypted with the lost password. This way all the data on the phone is accessible again. If the server password is lost, than a new server password must be issued by the server/administrator, but the specific procedure should be established by the individual projects according the available resources (let us recall that collectors might not even have e-mail accounts). Note that issuing a new server password does not affect the use of any of the mobile phones where the user is registered. Only the server needs to know about the change, and the mobile device can be normally accessed and used offline, solving the remaining issues discussed in Section 3.3.

   An interesting overview of the challenges of securing data in a J2ME device are discussed in [5].

6 API Overview

The API is designed to be transparent and flexible to establish secure layer on top of the existing application layer, both on the mobile client and on the server, while being as general as possible. Thus, it provides interfaces to perform secure operations like authentication, integrity, confidentiality without assuming any particular underlying architecture.

6.1 Proposed Secure Protocol Architecture

As we mentioned in the introduction, we use openXdata as reference system. Figure 4 shows the current architecture of the openXdata system. Note that no security mechanisms has been implemented in any of the components, apart from the user passwords in the server database, which are hashed with a salt. In addition, in order to authenticate a user on the phone while off-line, the user table which contains authentication credentials is downloaded on the mobile device and uses for user authentication locally without involving the server.

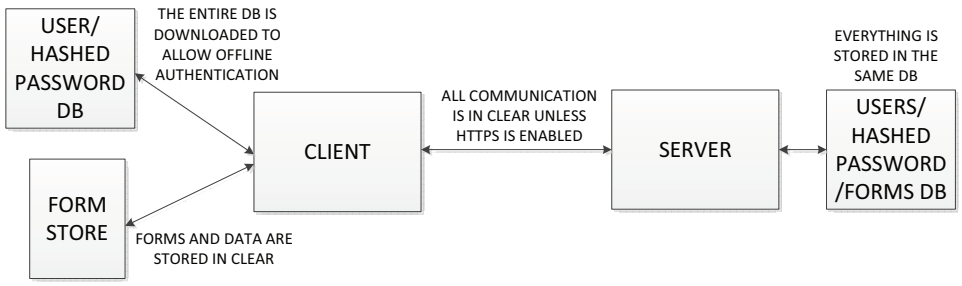


Figure 4: Current OXD architecture,

Figure 5 shows how our API can be integrated with the existing system. The application does not need to take care of the login process or the password database anymore. The API will create and manage a separate *Key Store* for each user who has registered on the mobile device. All keys and secret values needed for the cryptographic operation related to the user are kept encrypted on the device. The information in these stores will be decrypted only if the user successfully logs on the device as explained in detail in Section 7.7. In addition the API also creates and manages an *Application Store* where information common to all the users is stored (unencrypted), for instance, the server URL, the server public key, and other security settings. Thus the application need not be concerned with the technical details of the store management. The client simply create the storage and user the API to wrap it into a secure storage, so that every write/read operation will be actually performed by the API that takes care of encrypting before writing and decrypting before returning the information to the client. The API also takes care of checking whether the current user has permissions to write in that storage and handle the corresponding keys. All of this happens in a completely transparent way for the client. The communication with the server takes place in a similar manner, where the API encrypt all the traffic before sending it, and decrypts the incoming traffic before relaying it to the client. In this way, no assumption needs to be made about how or where the forms are stored, thus being able to use the API with existing applications without significant changes to the application itself.

Similarly, the API has control of the key store and user database also on the server. However, the secure layer provided by the API on the server side acts more as a proxy than a service provider. It simply relays the requests from the application layer of the server to the mobile by encrypting them first, and those from the mobile to the server, by decrypting them and extracting the original request from the mobile application layer.

7 Implementation Challenges

7.1 Server Authentication

Attributes in JAR manifest and application descriptor (JAD) are used to install MIDlet suite and MIDlet configuration respectively. JAR manifest attributes are protected by digital signature. The challenges remain with unprotected attributes on the JAD file. Here, one of

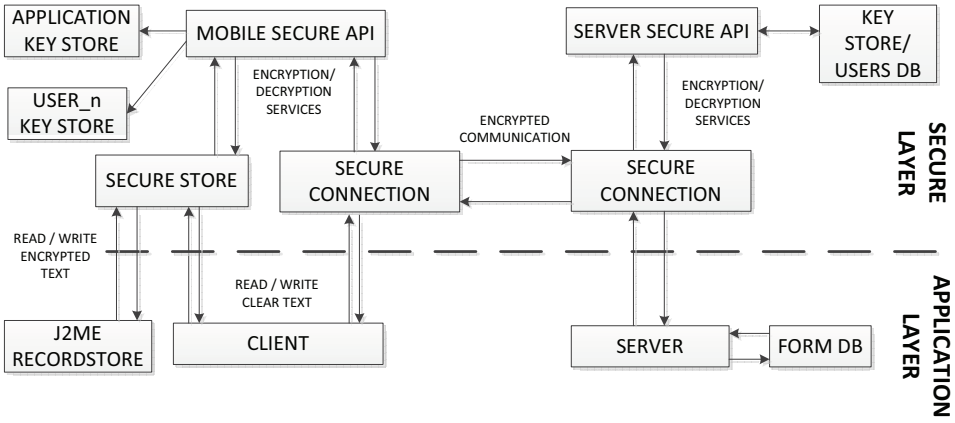


Figure 5: Proposed architecture with secure API.

the steps of secure protocol comes to the picture i.e. server authentication. There are two approaches to keep the integrity of JAD attributes. The first approach is to download the JAD file through a secure channel, which comes at a price of establishing a secure channel. Besides, downloading through https is done by the browser, and the application does not have a way to know whether it has been actually downloaded through a secure connection, so it should validate the JAD anyway. Unless the developer disable this option because they are sure everyone will download through a secure connection, or the application is pre-installed on the phones. The second approach is to check attributes integrity after application installation completed. In this paper, we have chosen the second approach and called it server authentication. This is one of the steps in secure protocol that deals with protecting the JAD attributes and making sure that installed application comes from genuine application provider.

Server Authentication implements a challenge response protocol based on public-key and symmetric encryption. Our performance test result presented in section # shows that the low-end phones compute RSA based public-key encryption [10] with key size of 1024-bit in acceptable time. Therefore, in server authentication, we use untrusted server public-key that is found in JAD file to encrypt the challenge request. The challenge request consists of a session key encrypted with PKCS#5 password-based cryptography standard citeRFC2898 generated key from a pre-shared secret between the user and the server and Advanced Encryption Standard (AES) algorithm and message authentication code (HMAC) for message integrity.

The session key is temporary persisted until the client get a response to the server. The secret can be a simple pin-code or pass-phrase. Since the users of a project are pre-configured on the server, each of them can be given such pin-code or pass-phrase in advance with their user-name and password. When the server receive the public-encrypted message and decrypts with its own private key, it gets the challenge request. The server uses the pre-shared pin-code or pass-phrase to generate decryption key using the same PKCS#5 password-based cryptography standard and decrypts the challenge request. If the server is able to decrypt the challenge request, it uses the session key within the decrypted challenge request to encrypt a response with hash of JAD attributes. Once the client gets back a response, it uses the session key to decrypt the response and verify the JAD attributes hash within the response and installed application JAD attributes hash. More importantly, the server authentication steps verify the public-key of the server that is used to secure client-server channel (the key is , in fact, one of the JAD attributes to verify).

Figure 6 represents the most general situation, when project and download server might be distinct, and the download can use either a normal or a secure connection. In any

case, after installation, the application will automatically run the server authentication step. There are also a couple of drawbacks in this approach. The first is that users might have to remember a pin-code or pass-phrase in addition to their server and mobile password. To mitigate this problem we suggest that the mobile phones used in a project should be distributed with a pre-installed application to the users. In this way, only a small group of special users will be in charge of performing the installation, and normal users will not need to even know about the pin-code or pass-phrase. The other one is that we cannot use the attributes of the JAD file to configure the application until the server identity is confirmed and the JAD file verified. Thus, also the seed parameter cannot be trusted, and the mobile device has to seed its random generator with probably a weaker seed. However, after this step, the server will regularly send new strong seeds to the device.

Note that the public encryption in this step is actually useless if the public key is not authentic, since then the attacker can easily decrypt it and read the content. In this case the security relies on the strength of the password based encryption. One might therefore think that it is more efficient to send only the challenge to the server without public-key encryption, but we can minimize the number of attackers to one who own the fake public-key on the JAD, where as if we send the challenge without public-key encryption, any eavesdropper can intercept the request and decrypt the challenges.

Also, if the application uses https instead of our protocol, this step is still necessary for checking the JAD, but no encryption or challenge is needed anymore since we rely on certificates to authenticate the server, so pin code and stuff would not be necessary any longer, and the public key wouldn't even need to be in the JAD since it would be exchanged in the ssl handshake.

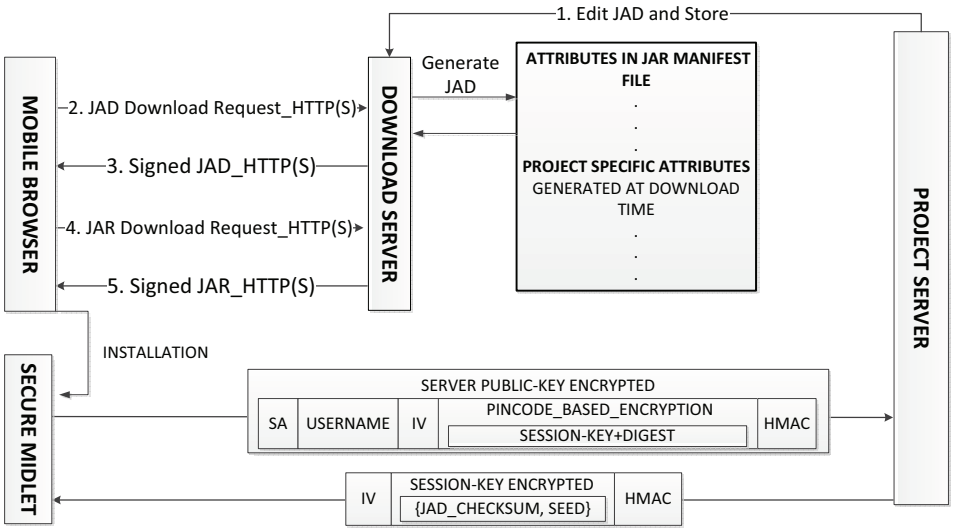


Figure 6: Client application installation procedure and JAD verification step.

7.2 User Registration

Once the application has been installed and configured, unlocking the application requires user registration to the server with pre-shared username and password. Figure 7 shows the format of the registration client request and server response. Note that, in addition to the usual parameters described in Figure 7, the request contain 128-bit (can be different key size) AES encryption key called a storage key. This key is generated using a secure random generator and seeded with a strong seed that we get on server authentication step.

If the user successfully authenticated on the server, the server persists the storage key on the database and send acknowledgment to the client encrypted with the session key. The client uses storage key to encrypt the user key store if the registration succeeds. By sending it to the server, users will be able to recover their account in case of lost mobile password. As mentioned in Section 3.3 this is the password a user chooses after registration and that is used to encrypt the storage key and authenticate the user locally. The sequence diagram in Figure 13 shows the context in which the registration procedure takes place.

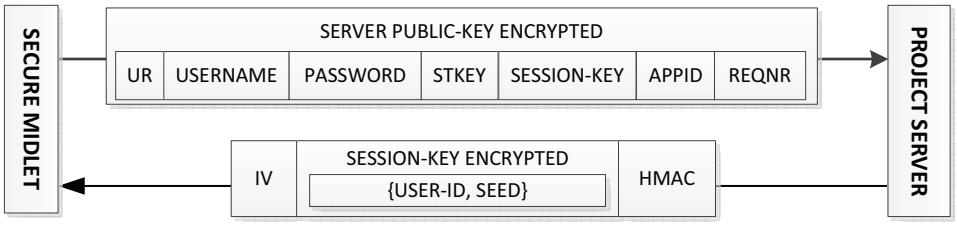


Figure 7: User registration request and response.

7.3 Password and Data Recovery

The password recovery step is initiated when the client lost or forgotten login password. The step is useful for lost password and data recovery on the client. The request (shown in Figure 8) is practically identical to the registration request, but it does not contain the storage key, which is instead received with the response from the server, and does use the username, which is accessible on the client. When the procedure is invoked, the current password is reset, which means that the encrypted storage key and salt are deleted from the user's Key Store , but not the other information. At this point, the user is allowed to choose a new password to encrypt the storage key using PKCS#5 password-based cryptography standard, and resume the work with no loss of data, and with no changes to either the other accounts or the server settings.

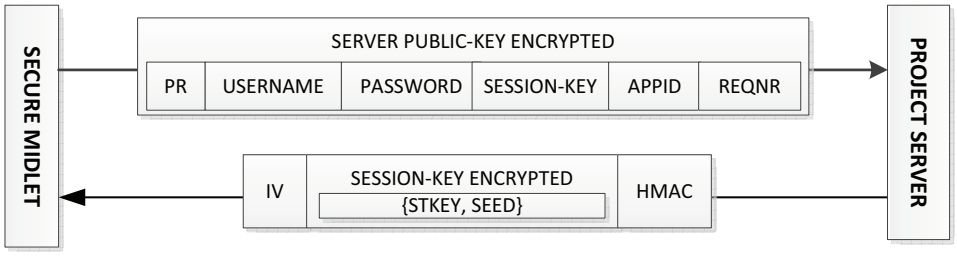


Figure 8: Request and response used for the password recovery step.

7.4 Upload and Download

While server authentication, user registration and password recovery are used by secure solution to perform services offered by the API itself, the upload and download services are used by the application to secure its own transactions with the server. In other words they are used to secure the communication on the application layer. Figure 16 shows the general structure of this type of transactions. Since we do not know the protocol used by the application to communicate with the server, we simply provide a way to encapsulate

the application requests in a secure and encrypted channel. In addition, we use session identifiers in order to minimize the user authentication requests to the server, which are encrypted with the server public key and therefore computationally expensive for the mobile device. Similar to the SSL, the secure solution encrypt the header and body of the client http application request. Thus, the session-id will be sent in clear, but the data itself will be encrypted with a AES 128-bit session key and the message authenticated with an Hash-based Message Authentication Code (HMAC). Hence, even in case of session hijacking data should not be compromised, unless the session key is guessed by brute force or other methods.

Figure 9 shows the general structure of this type of transactions.

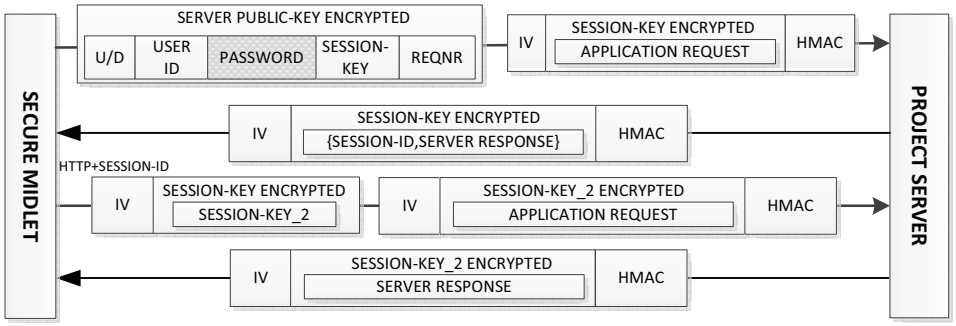


Figure 9: General request/response sequence for an upload or download session.

There is a further difference between the upload and the download request format. Since the API has been specifically designed for data collection through forms, the upload has been optimized for encrypted forms. As showed in Figure 5, forms can be stored encrypted with different keys, and the flag can indicate which key has been used according to whether they are ready or not for upload. If they are, the encryption key is used also as session-key, in order to optimize performance. In fact, this allows the application to write to the communication channel directly as forms are read from memory, without further encryption/decryption. If HTTPS is used, forms should probably be decrypted before upload, but this is something the developers of the secure application can easily decide and implement themselves. Figure reffig:uploadform shows the structure of an upload request. Since the byte stream is encrypted, the server sees only a stream of bytes and cannot know where one encrypted form starts and ends. To solve this problem without giving away any information on the stream, we added two more encrypted blocks. The first one, of fix size, contains the size of the next encrypted block. The second one, of size depending on the number of forms uploaded, contains the size (and therefore the offset) of each encrypted form in the stream. Finally, an HMAC is added at the end to guarantee data integrity and authenticity.

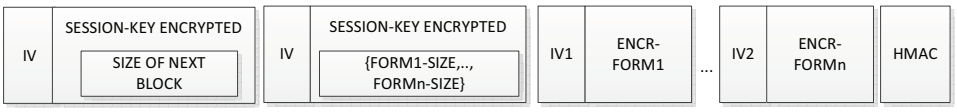


Figure 10: Format of the upload stream.

7.5 Overview of Implementation

In this section we explain the main functionality of the API in the different packages. We also show how they are used in practice through sequence diagrams and code examples.

The API is organized in five packages, which are shown Figure 11. The following three packages contain the implementation of the secure services that the API offers.

The `openXSecureAPI.communication` package contains the protocol steps dealing with mobile-server transactions, like server authentication, user authentication and registration, secure form download and upload and password recovery.

The `openXSecureAPI.storage` package contains the implementation of the key stores used by the API, both encrypted and not, used to store user keys, passwords and security settings of the application.

The `openXSecureAPI.crypto` package consists of the interfaces and implementations of the cryptographic primitives used by the API, including public key encryption, symmetric ciphers, the algorithms to perform password based encryption and normal and authenticated digests (HMAC).

The forth package `openXSecureAPI.client` is the only one actually used by an application to build the secure layer, and the only package that uses the `openXSecure-API.display` package in order to interact with the user to collect password, user-name and other required information through the device display. The content of each package is examined in detail as follow.

7.5.1 The `openXSecureAPI.client` package

This package is dedicated to create secure layer, and interact with the user to collect password, user-name and other required information through the device display. The existing application needs only to extend the abstract class `AbsSecureClient`, implement the abstract method `userMenu()` and initialize the `SecureController` as shown in the code below, in order to add a secure module that takes care of initializing the application, registering users and performing the login.

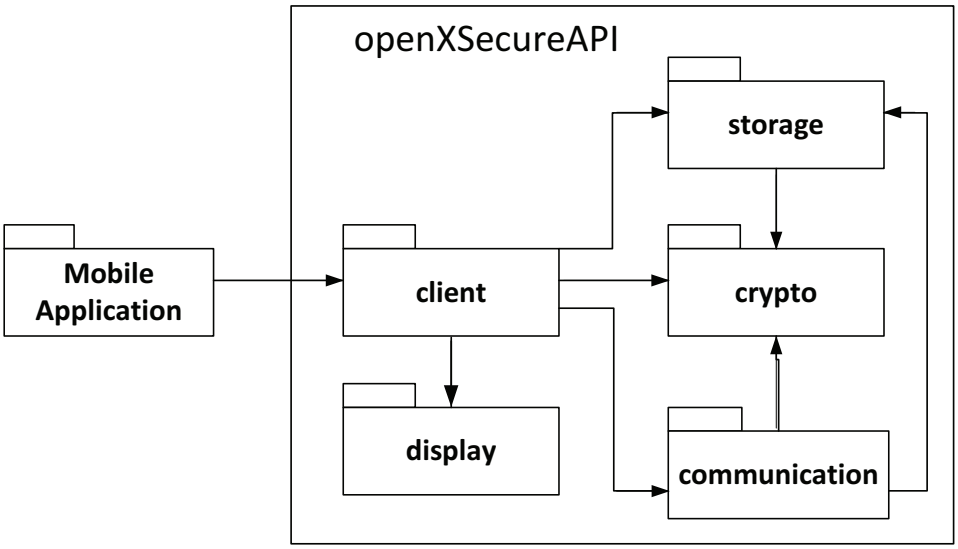


Figure 11: openXSecureAPI organization.

An existing client only needs to use this package to build the secure layer. Figure 12 shows the classes contained in the package, that are available to the client.

The class `AbsSecureClient` implements the methods used by the API to get and return control of the program flow to and from the application, and to handle some critical objects, like `SecureTools`, by making sure that the programmer cannot introduce security vulnerabilities in the secure layer. In this regard, this class is a black box the application

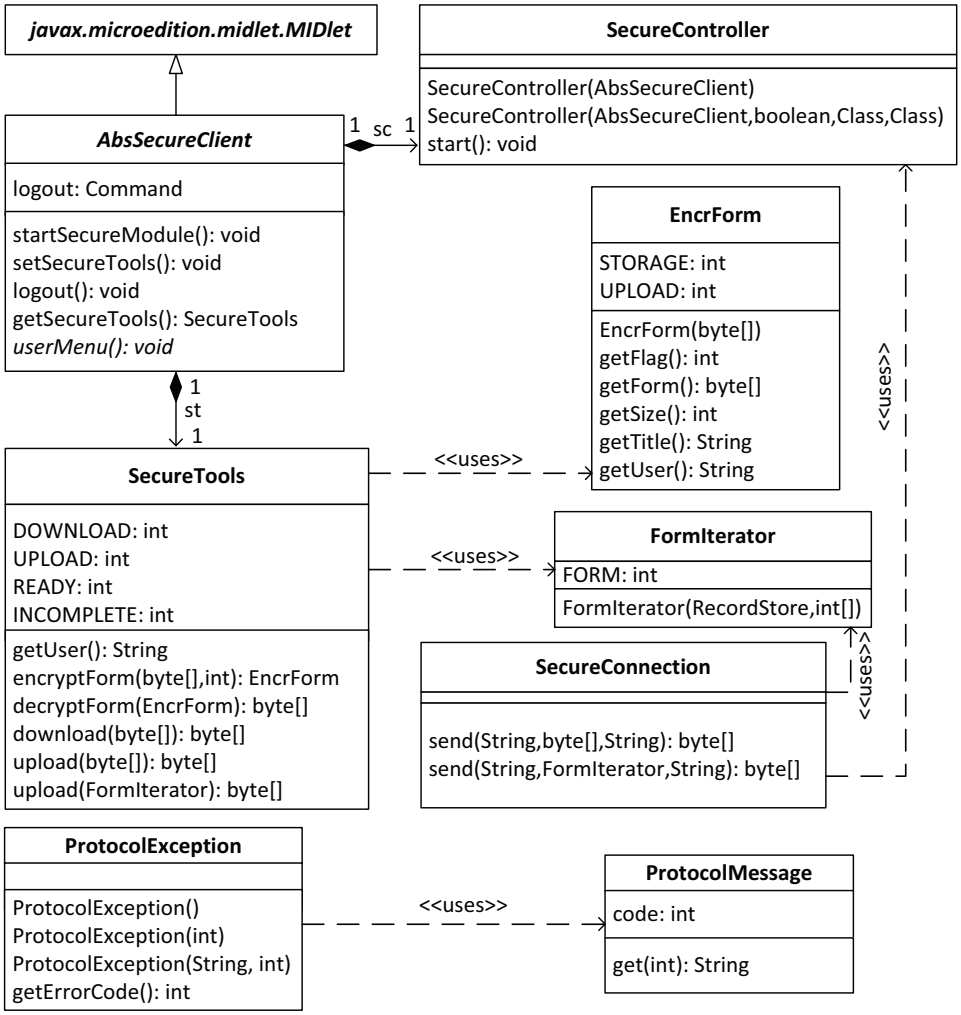


Figure 12: The classes used to add the secure module to an existing application.

cannot modify.

What actually happens is illustrated in Figure 13. The `SecureController` object `sc` takes care of initializing the application and communicating with the secure server, in order to authenticate the application itself and the users, and to perform the logging procedure. Once a user is logged in, the `SecureController` creates a `SecureTools` object initialized with all the credentials of the user who logged in, and passes it to the application by storing it into the `AbsSecureClient.st` field. The `userMenu()` method is therefore supposed to implement the first screen a user is shown after a successful login. From this point the application will have both control of the program flow again and will be able to access the API and its services through the `SecureTools` object now stored in the `AbsSecureClient.st` field.

A general recommendation when using the `SecureTools` methods is that they are run in a separate thread, to make sure that the the system thread is not locked and the screen remains available to obtain, for instance, the user password needed to perform a download or upload operation.

The `AbsSecureClient` class extends the class `MIDlet`, since otherwise the application would have to extend both the classes `AbsSecureClient` and `MIDlet`, which is not allowed in Java. Making the `AbsSecureClient` into an interface was also not an option, as it would leave the responsibility of implementing classes critical for security to the application developer.

Finally, the two classes `ProtocolException` and `ProtocolMessage` are used by the client to catch the API specific exceptions, and by all classes in the API to encapsulate various type of standard Java ME exceptions and re-throw them as more meaningful exceptions of type `ProtocolException`, with a custom error message defined in the `ProtocolMessage` class.

The existing application needs only to extend the abstract class `AbsSecureClient`, implement the abstract method `userMenu()` and initialize the `SecureController` as shown in the code below, in order to add a secure module that takes care of initializing the application, registering users and performing the login.

```
public class SecureApp extends AbsSecureClient {

  /** The object used to control the display */
  private Display display;
  /**The object containing the screen to be shown */
  /* to the user after login*/
  private Displayable userMenuScreen;

  /**Standard MIDlet method called to*/
  /* start the application*/
  protected void startApp() {
    //Initialize the display object
    display=Display.getCurrent(this);
    //Initialize the inherited field with a
    //(possibly customized) controller
    this.sc=new SecureController(this);
    //Call the inherited method that starts
    //the secure module
    this.startSecureModule();
  }

  /**Implement the abstract method inherited*/
  /* from AbsSecureClient */
  public void userMenu() {
```

```
    //Create the screen to display.
    //The name of the user
    //can be retrieved by using the SecureTools
    //object stored in the inherited field st.
  userMenuScreen= new TextBox(
   "User Menu","Welcome "+st.getUser(),25,0);
   display.setCurrent(userMenuScreen);
}
}
```

7.6 The `openXSecureAPI.crypto` package

This package consists of the interfaces and implementations of the cryptographic primitives used by the API, including public key encryption, symmetric ciphers, the algorithms to perform password based encryption and normal and authenticated digests (HMAC).

The package consists of the class `CryptoToolsFactory`, the interface `CryptoTools` and the class `BCCryptoTools` implementing this interface. The class `PasswordManager` uses a `CryptoTools` object in order to perform user authentication and password registration.

Since the cryptographic methods are accessed only through the interface `CryptoTools`, the API is independent from the actual implementation of the cryptographic primitives, allowing the developer to use his/her own implementation through the constructor of the `SecureController` class.

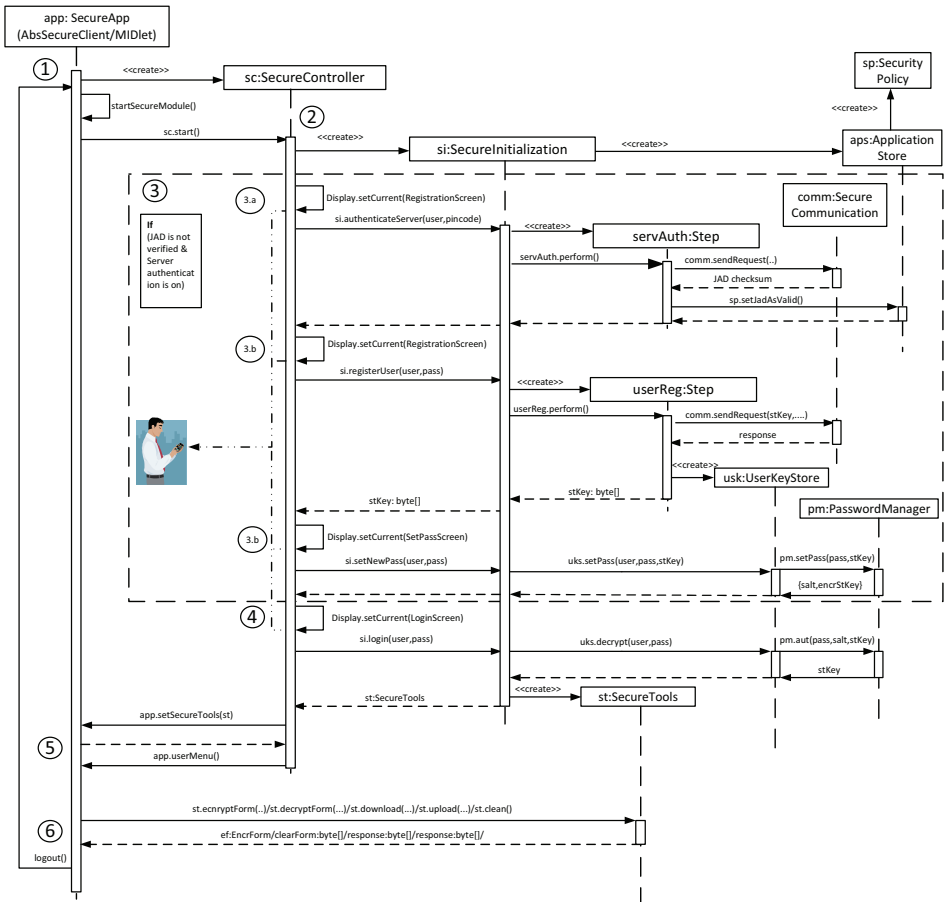
The default implementation we provide uses Bouncy Castle as provider [13], and the following cryptographic algorithms:

- RSA for public key encryption.
- AES in padded CBC mode with initializing vector (IV) for symmetric cryptography.
- SHA1 digest.
- HMAC based on SHA1 digest.
- Password Based Encryption based on PKCS#5.
- Random generator based on the `SecureRandomGenerator` class provided by Bouncy Castle.

The `CryptoTools.init(byte[] pk)` method is used to initialize the asymmetric cipher with the public key, which unlike the symmetric keys, is permanent. The parameter is a byte array, since different implementation might need different formats. For instance, Bouncy Castle requires the modulus and exponent separately as `BigInteger`.

7.7 The `openXSecureAPI.storage` package

The package handles most of the key management part of the protocol . It contains the implementation of the key stores used by the API, both encrypted and not, used to store user keys, passwords and security settings of the application. The storage has been designed to account for some typical scenarios in mobile data collection. In particular that multiple users should be allowed to use the same mobile device and the same user can use multiple mobile devices and that Internet access might not be always available. This means that mobile devices can no longer be considered private or personal to an user and that most of the data collection might have to be done off-line. From a security perspective this translates in the following concerns:



1. As shown in Section 7.5.1, the application simply creates and starts the `SecureController` object `sc`.
2. The `sc` creates a `SecureInitialization` object `si` that first extracts all parameters from the JAD file, and stores them in an `ApplicationStore` and a `SecurityPolicy` object.
3. If the application is run for the first time, the JAD file needs to be authenticated, otherwise we can skip directly to step 4. The `sc` controls the screens to obtain data from the user, while the `si` controls the actual protocol steps. Each `Step` creates the requests and handles the responses that are sent and received through the `SecurityCommunication` object.

3.a If the server authentication step this succeeds, then the security policy is set to valid, and the first user can be registered.

3.b If the user authentication performed by the `userReg` object is successful, it means that the server has a copy of the user storage key (`st`), and the `UserKeyStore` object can be created and initialized for the user.

3.c The user needs to enter a new password for the `UserKeyStore`, after which the login process can be started.

4. The login process is performed, and if it is successful, a `SecureTools` object for the user is created.
5. The `st` is set in the `app` object, and the control is return to the application by invoking `app.userMenu()`.
6. The application can now either use the `SecureTools` object or invoke the predefined `logout()` method, that will clean up the `SecureTools` object and restart the `SecureController` object.

Figure 13: A typical execution of an application using the API.

1. A mobile device must store some identification token to authenticate users off-line, but it should not store information that can be used to access the server.
2. If a user looses the password, the other users on the same devices and their data should not be affected.
3. If users change their password on the server, possibly from a web application, the access to the mobile device should not be compromised.
4. Even if the password is lost, it should always be possible to recover encrypted data stored on the mobile phone by some authorized entity.

The implementation discussed in this paper satisfies all these requirements. Figure 14 shows the classes in this package.

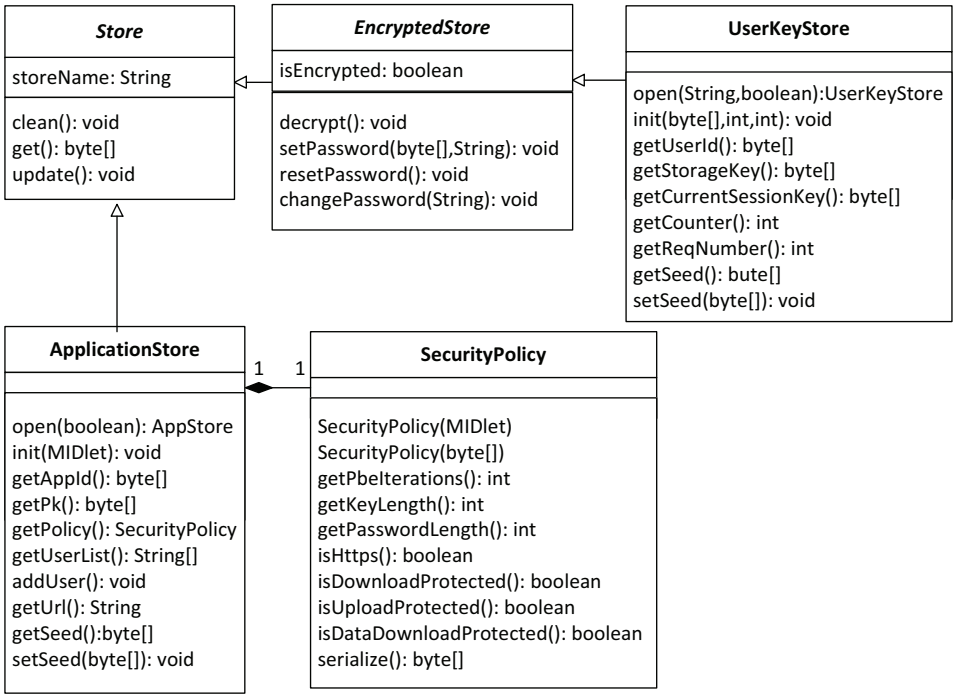


Figure 14: The class diagram of the `secureMobileAPI.storage` package.

First of all, each user has its own `UserKeyStore` object, where their secret keys are kept encrypted. As every user has a different password and an encryption key (which we call *storage key* or `stKey` in the code), compromising or blocking a user account does not affect other users on the same device. Besides, the password used to decrypt an `EncryptedStore` object is not the same as the one used to encrypt all the data in the storage, as explained in Figure 15.

Hence, each user has two passwords, but none of them is in practice used to encrypt *all* their storage. The one they choose themselves after the registration process, the *mobile* password (see Figure 13), is used to log in on the device and encrypt/decrypt their personal storage key, which in turn encrypts their storage (see Figure 15). The second password, the *server* password, is not used or stored at all on the mobile device, but it is used only for server related services. This means that if the mobile password is lost, the server password can be used to retrieve the storage key from the server and therefore reset the mobile password with no loss of data (see the recovery procedure in Section **??**). Also, if the server password is lost or changed on the server, this does not affect the mobile device in any way.

Finally, if the mobile device is compromised, only the data currently stored on it can be stolen, but the server is still safe as the password to access it is nowhere to be found on the device.

Note, that if someone gets hold of the device, it will not be difficult to access the encrypted data stored on it. In this case, an attacker could apply brute force to try and guess the encryption key. The storage key has been produced by a secure random generator that is properly seeded, while the password based key depends on the password chosen by the user. The security of the data depends, therefore, on the user's password. Unfortunately there is no real solution to this problem, as long as we want to have off-line authentication, and a chance to recover the data on the device if the password is lost. Therefore we can only try and educate the users to use strong passwords, even though this is an even bigger problem on mobile phones than on desktop computers.

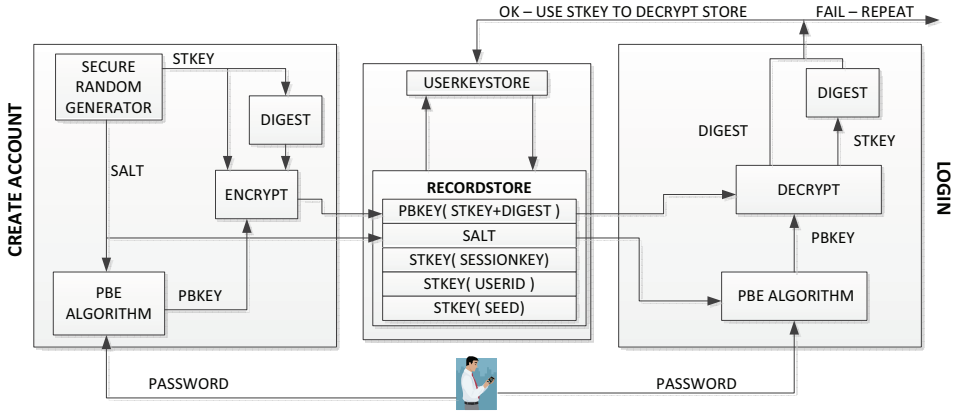


Figure 15: Login Procedure and password setting

The API will take care to clean up any data on the field members and variables when the user logs out.

7.8 The `openXSecureAPI.communication` package

This package contains the protocol steps dealing with mobile-server transactions, like server authentication, user authentication and registration, secure form download and upload and password recovery. The transactions can be divided into two categories: the ones used to initialize the application, and those used to transfer user's forms and collected data. The first type includes the server authentication at installation time, the user registration and the account recovery procedure. The second type consists of the upload and download of the forms.

The main problem we needed to solve when designing the communication protocol is that HTTPS might not always be a viable option for secure communication, due to constraints on the budget or the infrastructure. Hence alternative methods have to be used to certify the server/client authenticity and enforce end-to-end encryption. Such an alternative must deal with distribution issues caused by the lack of a recognized Certification Authority and of a secure channel for the download of the application. Besides, both the added complexity and the overhead due to the cryptography must be minimized in order to keep costs down and have a reasonable guarantee that the transactions will be completed even if connectivity is not optimal.

The general approach we propose when HTTPS is not available, consists of using a single request-response step to implement each transaction, with a simple hybrid cryptography system. Thus, a request sent by the mobile device will always consist of the user's

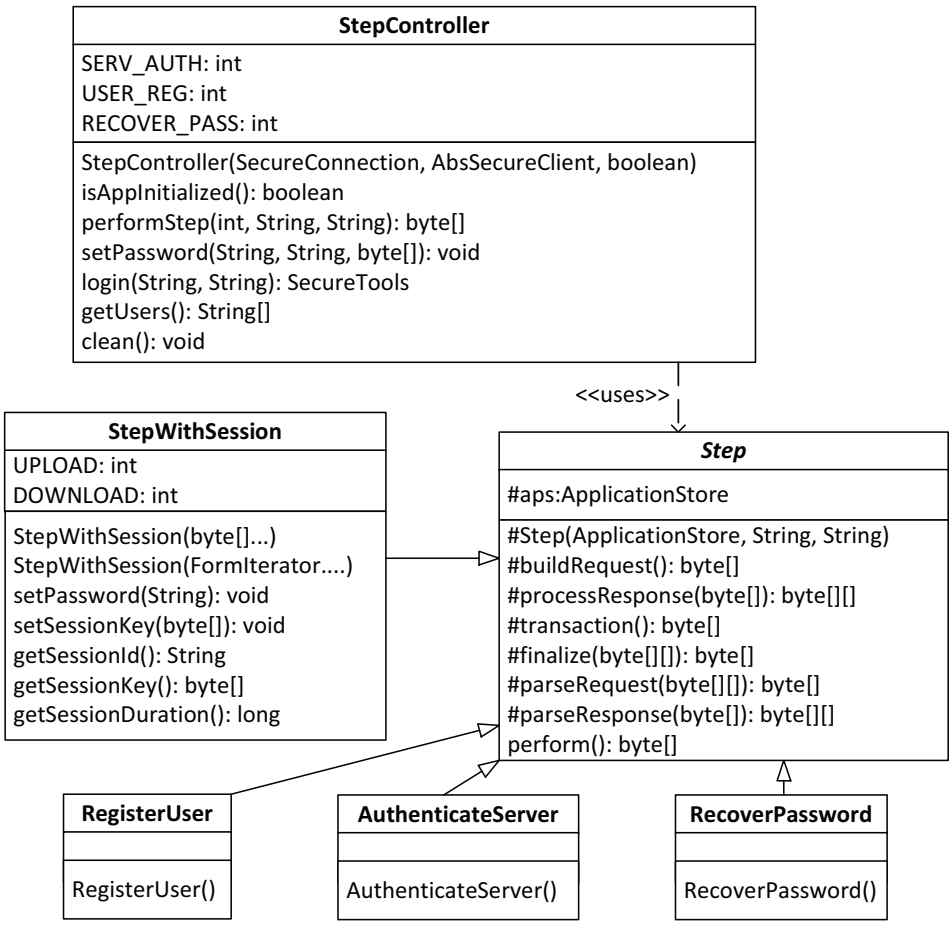


Figure 16: Overview of the `openXSecureAPI.communication` package

credentials and a symmetric session key, all encrypted with the server public key. Once the server receives this request, it can authenticate the user and both parties can use the session key to encrypt the rest of the communication. In figure 17 we show the general request-response format. Note that the application-id (APPID) is needed to identify the user-device pair uniquely, since a user might be registered on multiple devices. The session key will be used to encrypt the response, and we are guaranteed that only the real server will be able to use it, unless its private key has been compromised. The initialization vector (IV) and Hashed-based Message Authentication Code (HMAC) will strengthen the security by making it more difficult to temper with the response and break the encryption. The request number (SEQNR) is used to make each request unique and minimize the possibility of replay attacks.

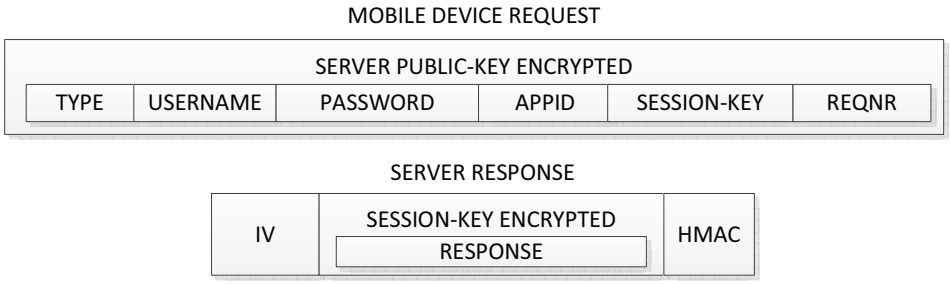


Figure 17: A typical client request and server response.

In the following subsections we describe the differences between different types of requests, but first we need to understand the underlying assumptions we made about the use of the application.

7.8.1 Client configuration

First of all, we assume that an application developed using our API is released by the application provider such as openXdata, so that it can be used out of the box to set up new projects for mobile data collection. This means that a project can configure it for its own purposes, but not change the code and recompile it. In other words, specific customer project settings cannot be contained in the JAR file of the midlet. The configuration can be done, instead, by extending the JAD file provided with the application with some predefined attributes as those showed in Figure 18.

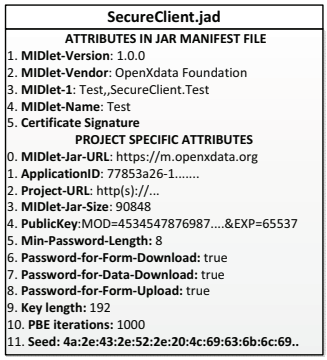


Figure 18: A sample Jad file.

In particular, the project is free to use HTTPS to secure its communication channel (by setting the attribute `Project-Server` to an HTTPS address), but if not, then they

should provide a public key for their server through `Public-Key` attribute. In addition the `ApplicationID` and `Seed` attributes should be dynamically generated at download time, in order to be unique for each client. The server-side API offers a module that takes care of this.

Being able to configure parameters like the public key, application-id and the seed through the JAD file, lifts the burden of configuration from the protocol itself, and solves both the key distribution issues, and the problem of collecting enough entropy on the mobile device to generate strong random numbers.

Finally, one security requirement we need to assume, and which is responsibility of the application developer, is that the JAR file containing the application itself is signed with a valid certificate issued by a CA recognized by the device. This guarantees at least that the application has not been tempered with, and the API works as it is supposed to. However, this alone is not sufficient to trust the information in the JAD file, as we explain in the next section.

8 Preliminary Performance Test

In this section we report the results of some preliminary tests we ran in order to analyze the performance of the API on devices with different hardware specifications and price categories. The results are summarized in Table 1.

Note that the phone used for the benchmark are phone that are most likely to be deployed on the field by openXdata. No smart phones are therefore considered. Also what we define as "powerful" phones, are only there to put the other results in perspective, since they also are not likely to be used due to their high cost. Remember that a typical openXdata phone should cost below 50USD.

It is clear that with the given parameters the performance of the API is barely acceptable on the least powerful phone (2760), but it has already a more than acceptable performance on an equally cheap and only slightly more powerful device (2330c). Note that the processor speed (3rd row in the table) is not always the most important factor. The most expensive and powerful mobile phone (E-63) we used in the test has very poor performance due to the high amount of time used to create new records in the record store (4th row in the table).

We have not tested our protocol when a HTTPS connection is used, but a simple SSL handshake took on average 12 seconds on all the devices tested, which is comparable with a complete Server Authentication step on the slowest phone.

It is also clear that the bottle neck in the various transactions is the RSA encryption [9], but no much optimization can be done in this regard. The key cannot be reduced to less then 960 bits, i.e., the smallest size required to guarantee that all protocol requests can be encrypted.

9 Related Work and Conclusions

Currently most mHealth systems for data collection do not systematically address security issues (and even SMS is widely used). There are mHealth systems that use HTTPS for the communication between mobile devices and server e.g. EmitMobile by Cell-Life (based on openXdata) [2] and EpiSurveyor [6] by DataDyne (based on JavaROSA mobile [16]). These represent commercial solutions, which use a centralized proprietary server where users who installed the client can register, but there is still no encryption of data stored on the mobile device. The OpenRosa consortium [15] is working to define general standards for mobile data collection practices and a particular reference J2ME implementation [16], including security. However, at the present time, the only adopted standard proposed as alternative to HTTPS is the Basic and Digest Access Authentication defined in [7].

Phone model (Nokia)	**2760**	**2330c-2**	**2730c**	**3120c**	**E-63**
Price ($)	50	<50	89	120	180
Processor speed (Mhz)	0,8	4,6	67,7	68,8	125,7
Time to create 20 records of 100 bytes on the phone (ms)	120	49	16	16	2573,9
RSA Encryption with 1024 bits key (ms)	3702	562	92	79	265
16 bytes AES encryption of a 100 bytes form (ms)	58	19	5	5	315
16 bytes AES decryption of a 100 bytes form (ms)	128	22	5	11	333
PKCS-5 password-based-encryption (100 iterations) (ms)	878	151	18	18	344
Processing time for Sever Authentication (ms)	11611	6306	5470	5021	11297
Processing time for User Registration (ms)	9226	6153	5392	3523	4186
Uploading 356 bytes of forms (ms)	3895	4989	5038	3295	1588
Downloading 2880 bytes of forms (ms)	4347	2868	4424	3023	1513

Table 1: Test results. Note that the total time used to perform a complete step (measured from the time when the OK button is pressed until the next screen is shown), includes the time taken by the user to allow an Internet connection since the application is not signed.

Related work in this area is sparse. Except for a protocol with some similarities to ours presented in [9], but with completely different working assumptions, and a client authentication protocol based on [7] proposed by the OpenRosa Consortium [15], we are not aware of any other work. In this paper, the authors also consider the problems of authentication and confidentiality if HTTPS is not available, but starting from different working assumptions, e.g., a banking application. In their scenario, the application would be customized and packaged for each user, so that all the necessary shared secrets and keys would be already bundled with the JAR file of the application. The application would also be installed directly on the user phone by an administrator. This eliminates the issues we discussed in Section 3.3 and 3.5. In addition only symmetric encryption is used. Finally, the secure storage problem is reduced to simply encrypting the user keys that came with the application, and no real password based encryption is used.

In general, all modern smart phones provide crypto API to develop secure application. Especially Balckberry, Andoid and iOs [1, 22, 8]. But as we mentioned several time, we are developing a secure solution for the Java ME platform, which does not provide any security, especially for the storage [26, 5, 14].

The protocol we implemented is relatively standard, if it is compared to well-know key distribution and encryption strategies. Also other works having the Java ME platform as target have been proposed[25], but it is easy to see that they are all tailored for the specific target applications. We are no exception. This implementation considered the challenges face by data collection in remote working location and with low-end equipment, ant tried to come up with ad-hoc solutions for this scenario, but also trying to create an API general enough to be easily integrated with different existing systems in this area.

The API presented in this paper is based on a protocol developed specifically for typical scenarios and usage patterns of mobile data collection tools in low-income countries, and offers a complete range of security services in addition to ease of use and flexibility. We have also developed a prototype of a data collection MIDlet using the API, and tested it on various phones with different settings in order to collect experimental data on the performance of the API. The results are encouraging, since the performance with the default security settings was acceptable also on low-end phones, and it is clear that high-end phones can easily use even higher security settings. Besides, considering that the version tested is still a prototype, we can hope for significant improvements.

More test has been done in order to optimize the code and the modularity of the API. The test improved both the performance and the memory footprint. Currently, the size of the JAR file containing the API is around 45 Kb, although almost half of it is due to the external Bouncy Castle libraries. More attention will also be given to mechanisms

for automatically configuring the security settings on each device, in order to maximize security without compromising usability.

The API has been integrated smoothly with the openXdata client and server with less amount of effort without changing the existing client and server code. The API provides independent and transparent solution for securing storage, communication link and authentication. The new secure openXdata client and server will be then deployed in the field in a real study to further analyze its performance and usability and reducing even more the interaction between user and security, hence minimizing the risk for human error in the configuration which might lead to security vulnerability.

References

[1] B. S. K. Base. http://us.blackberry.com/ataglance/security/knowledgebase.jsp. Online, Accessed Mars 2011. 9

[2] Cell-Life. Emit. http://www.emitmobile.co.za/. Online, Accessed Mars 2011. 1, 9

[3] O. Clinica. http://www.openclinica.org/. Online, Accessed Mars 2011. 1

[4] S. Crocker and J. Schiller. RFC 4086 - randomness requirements for security. http://www.ietf.org/rfc/rfc4086.txt, 2005. Online, Accessed March 2011. 3.4

[5] T. Egeberg. Storage of sensitive data in a Java enabled cell phone. Master's thesis, Hgskolen i Gjøvik, 2006. 4.2, 3, 9

[6] Episurveyor. http://www.episurveyor.org/. Online, Accessed Mars 2011. 1, 9

[7] J. Franks, P. Hallam-Baker, J. Hostetler, S. Lawrence, P. Leach, A. Luotonen, and L. Stewart. RFC 2617 - HTTP authentication: Basic and digest access authentication. http://www.ietf.org/rfc/rfc2617.txt, 1999. Online, Accessed January 2011. 9

[8] A. iOS Security. http://support.apple.com/kb/HT4456. Online, Accessed April 2011. 9

[9] W. Itani and A. Kayssi. J2ME application-layer end-to-end security for m-commerce. *Journal of Network and Computer Applications*, 27(1):13–32, January 2004. 8, 9

[10] J. Jonsson and B. Kaliski. Public-key cryptography standards (PKCS) #1: Rsa cryptography specifications version 2.1. http://www.ietf.org/rfc/rfc3447.txt, 2003. Online, Accessed March 2011. 7.1

[11] B. Kaliski. RFC 2898 - PKCS #5: Password-based cryptography specification. http://www.ietf.org/rfc/rfc2898.txt, 2000. Online, Accessed April 2011. 2

[12] A. N. Klingsheim, V. Moen, and K. J. Hole. Challenges in securing networked J2ME applications. *IEEE Computer*, 40(2):24–30, 2007. 3.4

[13] T. Legion Of the Bouncy Castle. http://www.bouncycastle.org/. Online, Accessed Mars 2011. 3.4, 7.6

[14] F. Mancini, K. A. Mughal, S. H. Gejibo, and J. Klungsøyr. Adding security to mobile data collection. Proceedings of IEEE HEALTHCOM 2011 Conference-13th International Conference on E-Health Networking, Application and Services, Columbia, MO, USA. June 2011. 1, 9

[15] Open Rosa. http://www.openrosa.org. Online, Accessed Mars 2011. 1, 9

[16] Open Rosa. Javarosa. http://www.javarosa.org. Online, Accessed Mars 2011. 1, 9

[17] OpenDataKit. http://www.opendatakit.org. Online, Accessed Mars 2011. 1

[18] OpenXData. http://www.openxdata.org. Online, Accessed Mars 2011. 1

[19] Oracle. Java ME reference. http://www.oracle.com/technetwork/java/javame/index.html. Online, Accessed Mars 2011. 1, 3.4

[20] Oracle. Security and trust services API for J2ME (SATSA). http://java.sun.com/products/satsa/, 2006. Online, Accessed March 2011. 3.4

[21] E. Rescorla. *SSL and TLS: Designing and Building Secure Systems*. Addison-Wesley Professional, 2000. 4.1

[22] A. Security. http://developer.android.com/guide/topics/security/security.html. Online, Accessed May 2011. 9

[23] K. I. F. Simonsen, V. Moen, and K. J. Hole. Attack on sun's MIDP reference implementation of SSL. *Proceedings of NORDSEC 2005 - 10th Nordic Workshop on Secure IT systems, Tartu, Estonia.*, 2005. 4.1

[24] I. Society. RFC 2818 - http over tls. http://www.ietf.org/rfc/rfc2818.txt, 2000. Online, Accessed October 2010. 1, 4.1

[25] Z. Wang, Z. Guo, and Y. Wang. Security research on j2me-based mobile payment. *IEEE Communication Society*, 2(2):644–648, 2008. 9

[26] B. Whitaker. Problems with mobile security #1. http://www.masabi.com/2007/07/13/problems-with-mobile-security-1/, July 2007. Online, Accessed Mars 2011. 4.1, 9

[27] J. K. yr. Handheld computers for data collection in field research in Uganda. development of EpiHandy and field tests. Master's thesis, Centre for International Health, University of Bergen, 2004. 3.2