Secure Mobile Data Collection Systems for Low-Budget Settings

F. Mancini, S. Gejibo, K. A. Mughal, R. A. B. Valvik
Department of Informatics
University of Bergen
Bergen, Norway
{federico, khalid, sge065}@ii.uib.no, remi@valvik.org

J. Klungsøyr
Centre for International Health
University of Bergen
Bergen, Norway
mihjk@cih.uib.no

***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. Mobile Health (a.k.a. mHealth) is an emerging and promising health service delivery concept that utilizes mobile communication technology to bridge the gap between remotely and sparsely populated communities and health care providers. So far, several mHealth applications have been developed and deployed in the field, but many of them do not systematically address the security and privacy issues involved. As sensitive information is stored, exchanged and processed in these systems, issues like privacy, confidentiality, integrity, availability and authentication must be dealt with accordingly. In this paper, we analyze the challenges in securing Mobile Data Collection Systems deployed in remote areas and in low-budget settings, and discuss how one can provide an adequate security solution for such systems.**

***Keywords*-mHealth, Mobile Data Collection Systems, Mobile Security, J2ME, HTTPS, secure communication protocols, secure mobile data storage, secure mobile data transmission**

I. Introduction

Mobile Health focuses on mobile devices as the main tool used to deliver health services and disseminate information to different stakeholders. In particular, 6 main areas of application were identified in the UN Foundation and Vodafone Foundation report [26] on mHealth: Education and Awareness, Remote Data Collection, Remote Monitoring, Communication and Training for Healthcare Workers, Disease and Epidemic Outbreak Tracking, and Diagnostic and Treatment Support. Here, we focus on applications for Remote Data Collection, which we will refer to as Mobile Data Collection Systems (MDCS).

MDCS allow the collection and transmission of data from remote geographical locations to data storage repositories through wireless or cellular network. It is a combination of a client application running on the mobile devices, wireless infrastructure and remotely accessible server databases. Most of these systems share identical principles and guidelines to collect data remotely. The process begins by designing a form, which contains a set of questions for collecting the relevant data. The form is stored in a server database. Collectors can then download these forms on their mobile device and use them to collect the actual data in the field. The forms that have been filled are stored on the mobile device until it is possible to upload them to the central server.

These systems clearly deal with a large amount of sensitive and private information, especially when used for medical purposes, and they should provide an adequate level of protection for data at all times: both while it is being collected and stored on the mobile device, and while being transferred and stored on the server.

The work we present here originated from a collaboration with *openXdata* [16], an open-source MDCS used to conduct health related projects in low-income countries. We primarily assessed the system from a security point of view. What we found was that most basic security concerns had not been addressed in a satisfactory manner or had been completely ignored. This led to a security review of similar MDCS, to understand whether they had similar issues or more adequate solutions. We looked especially at those systems that cater for low budget projects, and therefore use low-end mobile devices. In particular, we focused on those systems that provide a J2ME [17] client, like openXdata does.

A preliminary survey of systems like Episurveyor by DataDyne [2] and ComCareHQ by DiMagi [1] (both based on JavaRosa [15]), Nokia Data Gathering [14] and Mobenzi Researcher [12], [24], revealed that the security issues we found in openXdata were not an isolated case. We were not able to collect any information on Voxiva [27], although they claim to provide a secure solution.

However, although it is clear that the current security is inadequate, we do not think there is a simple and straightforward fix either. The reason is that standard security solutions are not always suitable for these systems. From a discussion based on the field experience of openXdata, it turned out that even the widely deployed HTTPS protocol [23] used to protect data transmission in most systems, might not always be a viable solution. The working environment in which these MDCS are deployed, the low budget settings of the projects using them, and other factors that we will discuss later, impose, in fact, heavy restrictions on their capabilities and reliability.

Therefore, the main goal of this paper is to understand the security implications of MDCS working with constrained environmental and technological resources, and identify suitable solutions. We do this by first presenting a security review of

openXdata and similar systems in Section II, where we look at whether basic security concerns have been addressed in a satisfactory manner. Then, in Section III, we discuss the main constraints that remote data collection is subject to, especially when deployed in low-budget settings in low-income countries, and what specific security issues this might entail. Using these observations as baseline, in Section IV we evaluate some available solutions that can be used to secure the mobile client in these systems and discuss how suitable they actually are. Based on all the above, we propose an alternative solution specifically designed for constrained MDCS in Section V, that can provide the following: local and end-to-end authentication, support for multiple users per mobile device, secure storage, key distribution and management, data integrity, and secure communication, even if HTTPS is not available. Finally, in Section VII, we conclude with some related and future work.

The secure communication and storage schemes we propose, are based on standard tools and techniques so that they can be easy to implement and evaluate, but they are also designed to be light and flexible enough to fit the particular requirements that we will identify throughout the rest of the paper. We have also conducted a first evaluation by implementing a J2ME API based on the approach we propose and using it to create a secure version of the openXdata client. Preliminary tests are promising, and a full field tests is being currently prepared. Some discussion on the API can be found in [5].

In the rest of the paper we assume the reader is familiar with security and cryptographic concepts.

II. Security Review of MDCS

As mentioned in the introduction, this work originated from a security review of various MDCS, that uncovered the lack of a systematic security approach to mobile data collection. We have compared the solutions adopted by some J2ME based systems to protect the data both in transit between client and server, and when stored on the mobile device. The results showed that as long as the platform used offered some standard means of protection, these were adopted, but otherwise not much was done to improve the situation. This turned out to be a major problem especially for the confidentiality of the data stored on the phone, since J2ME does not offer any libraries for encryption. We also briefly considered other security aspects like authentication and authorization.

A. Confidentiality

The first thing we checked was whether data was properly protected from unauthorized disclosure both when stored on the mobile device and when transferred to the server. In particular we verified that proper encryption was in place.

1) Communication: All MDCS we investigated use or support HTTPS to protect the transmission of data between the client and the server. In general this solution is more than adequate to ensure that data in transit between client and server is protected, under the right conditions. That is, as long HTTPS is used with certificates signed by a trusted Certificate Authority (pre-installed on the mobile device by the manufacturer), and the implementation of the SSL/TLS protocol on which it is based, is correct. In Section IV-B, we will show that these conditions are not always fulfilled or can be fulfilled, in which case no alternative is currently available to secure the communication.

2) Storage: Our findings regarding the protection of the data while being stored on the mobile device, unlike the client-server communication, give reasons for serious concern. Among all MDCS we analyzed, only one actually encrypted the data on the phone, and even in that case, we found some weaknesses in the solution that was used.

From the answers we got from DataDyne and Mobenzi Researcher customer support, we know that they do not encrypt the data stored on the phone, and their security actually relies mostly on two things. The first is that the Application Management System (AMS) on the phone prevents unauthorized applications from reading from another application memory store, and the other is that forms that are completely filled in, are automatically uploaded whenever a connection to the server is available. The AMS can work in some measure to sandbox applications on the phone, but the main problem with mobile phones, is that it is quite easy and straightforward to get a copy of all the data stored on the phone, and, once outside the phone, the data is completely unprotected [3]. In this case, uploading frequently can mitigate the problem, but in *remote* data collection, it is not unreasonable to assume that connection to the server might not be available even for days. Also, if the phone is lost or other sensitive data like user credentials are stored permanently on the phone, encryption should be in place to guarantee adequate protection.

To test the other clients, we tried ourselves to extract the data stored on the phone by means of a back-up software, and analyzed the source code where available. We found that most clients store their data in clear, but with different formats. OpenXdata stores the serialized Java objects, but data elements are still recognizable, while CommCareHQ and EpiSurveyor store all data as XML in clear text. EpiSurveyor, in addition, stores also passwords in clear text if one chooses to have the login form pre-filled. The only notable exception is Nokia Data Gathering. This client provides password-protected data encryption. However a quick look at the implementation [13] revealed that the encryption key is a direct MD5 hash of the password, and it is stored in clear on the phone. Hence, one can retrieve both the key and the encrypted data, hence rendering encryption useless.

B. Authentication and authorization

When dealing with authentication in remote data collection, there are mainly two aspects to consider: local authentication and server authentication.

Every mobile client should provide some form of local authentication in order to prevent any unauthorized user from tampering the data while stored on the phone, despite the encryption. In fact, someone authorized to use the application

would also be able to decrypt, modify and delete the data. This is even more critical if multiple users share the same phone.

Clients like openXdata, Episurveyor and CommCareHQ, have a login procedure in place that authenticates the users remotely on the server when the application is run the first time. Thereafter, openXdata and CommCareHQ store the credentials of the user on the device as a hash, so that off-line authentication is possible by computing the hash of the password entered by the user and comparing it against the stored one. Episurveyor, instead, continues to authenticate the user remotely, although it can save the credentials to pre-fill the login form as we mentioned earlier. The difference is how these credentials are stored. OpenXdata stores a SHA-1 hash of the alphanumeric password, concatenated with a salt received from the server and also stored on the phone, CommCareHQ seems to store SHA-1 hash of a numeric-only password without salt, while EpiSurveyor, as we mentioned stores everything in clear text. CommCareHQ in addition contains a local admin account with a published password, which allows for doing any manipulation on the data, submitting data, creating new users and changing any users password. Nokia Data Gathering, instead, does not seem to have users at all, but some authentication is in place if one chooses to encrypt the data on the phone, since a password must be entered to access and decrypt the data. In this case, the encryption key is the MD5 hash of the password and the authentication mechanism consists in verifying that no decryption error is thrown when some data is decrypted with the key derived form the password entered by the user. One problem with this approach is that it uses error messages from the decryption to detect a wrong key and decide the outcome of authentication, but unless some integrity check is done on the data after decryption, false positives could arise.

Among these approaches, although the security of all of them actually depend on the strength of the password itself, the openXdata one is probably the one to prefer, since it uses a salt to create the hash of the password, so that at least dictionary attacks based on precomputed rainbow tables can be prevented (given that the salt is long and random enough). Using only the hash of the password, especially with numeric-only passwords like CommCareHQ, increases considerably the possibility of success of brute force attacks, and storing a clear text password when remote authentication is performed, simply does not make sense.

When it comes to remote authentication, it is also important that not only the user, but also the server is authenticated. As long as HTTPS is used with certificates released by certified CAs, server authentication is taken care of, and if the client authenticates through username and password, those credentials are protected by the encryption provided at transport layer. However, if HTTPS is not available, only JavaRosa based systems offer an alternative authentication protocol, i.e., the Digest Access Authentication defined in [4]. This protocol is of course better than clear text passwords on an insecure channel, but provides only user authentication, and some critics found in [11] when it is used as a SASL mechanism are still valid when it is used as originally described in [4].

Regarding authorization, when multiple users are allowed to share a phone, they should not share storage, or if they do, some mechanism should be in place to prevent them from reading each other's data. This problem is independent from encryption, although personal encryption keys could help a lot. In openXdata, a bug has been recently reported that allows a user to do exactly this: if the a user has not uploaded all the forms to the server, then other users on the same phone can see those forms. Episurveyor seems to support multiple users, but when testing this feature we experienced that the last user who logged in the application, was automatically logged in again the next time we opened the client, even though the user had been logged out. Although we have not tested whether this behavior is due to a problem with the particular phone model we used, or Episurveyor in general. In any case, it is not acceptable if a mobile phone is shared by many collectors. Finally, CommCareHQ seems to have a default *admin* user with default password "234", which allows anyone to have complete control over the application.

C. Key and Password Management

As long as HTTPS is used to protect data transmission, only a valid certificate recognized by the mobile device is needed. Rest of the key management is taken care of by the SSL/TLS protocol on which HTTPS is based.

When passwords are used for user authentication, some distribution and recovery mechanism should be in place to guarantee availability of the service and the data, in case passwords are lost or forgotten. As long as the credentials are the same on both the client and on the server, and they are not stored on the client, it is quite easy to change centrally only on the server, although it might be challenging to distribute them securely to the collectors in the field. When a password is used to generate a cryptographic key like in Nokia Data Gathering, or it is stored on the phone for off-line authentication like in openXdata, then the change must be done both locally and remotely on the server, especially if the same password is used for both, and data availability should be preserved. The problem is how to enforce the change on the phone, when a user is locked out, without allowing unauthorized entities to reset a user's password, and how to access encrypted data again. We are not aware of any solution for these problems being adopted by any of the MDCS we tested.

The results are summarized in Table I.

III. Remote Data Collection Constraints

In this section we systematically identify the limitations and requirements the remote data collection has in practice, and how these can influence the design of a security solution for MDCS. This assessment is based on experience in the deployment of openXdata in projects like OMEVAC [9] and other field tests [8].

In the following subsections, we define the challenges for each primary aspect of remote data collection and discuss how different constraints translate into different security concerns.

	OXD	NDG	ES	MR	CCHQ
SC	HTTPS	HTTPS	HTTPS	HTTPS	HTTPS
SE	NO	YES	NO	NO	NO
OA	YES	YES	NO	N.A.	YES
RA	YES	NO	YES	N.A.	YES
SA	HTTPS	HTTPS	HTTPS	HTTPS	HTTPS
MU	YES	NO	YES	N.A.	YES
PM	NO	N.A.	N.A.	N.A.	N.A.

TABLE I
SUMMARY OF THE SECURITY REVIEW. OXD=OPENXDATA, NDG=NOKIA DATA GATHERING, ES=EPISURVEYOR, MR=MOBENZI RESEARCHER, CCHQ= COMMCAREHQ. SC=SECURE COMMUNICATION, SE=STORAGE ENCRYPTION, OA=OFF-LINE AUTHENTICATION, RA=REMOTE AUTHENTICATION, SA=SERVER AUTHENTICATION, MU=MULTIPLE USERS, PM=PASSWORD MANAGEMENT.

A. Low Budget Projects

Most of the challenges faced when trying to secure MDCS come from the fact that many projects using these systems run on very low budgets. This has repercussions on the type of hardware and software used. For example, such projects would out of necessity deploy mobile phones with low-end specifications that have low computational power and comparatively little memory. Hence the problem of finding an acceptable compromise between cryptographic strength and available computational power.

B. Remote Working Locations

Many of the projects using openXdata are deployed in low-income countries like Uganda [8], [9] 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. Thus, it is not unreasonable to assume that data collected might stay on the phone for several days before being uploaded.

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.

C. 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 off -line and large amounts of sensitive data can be stored on a phone. Different users should not be able to access other data than their own or compromise other user accounts.

D. Collectors

Collectors must usually be trained in the use of the MDCS before being able to go in the field and start collecting data. Adding security should not compromise the usability of the devices. Also, if collectors were already trained to use a specific MDCS, they might need to be trained again if the adopted security solution has led to significant changes in the user interface or the application logic.

E. Privacy and data ownership

Many projects that collect private and sensitive information, might have an issue with storing their data on databases located on third party servers. These projects might prefer to set up their own servers as allowed by openXdata, Nokia Data Gathering and CommCareHQ, rather than using systems offering everything from the client application, to data storage and management, like Mobenzi Researcher or Episurveyour.

The drawback is of course that the costs related to the set up and maintenance of the infrastructure are left to the individual projects. Therefore, the running costs should be minimized, and this could preclude the adoption of some security solutions.

F. Availability

It is paramount that the collected data is not lost or rendered unaccessible just because a password for the mobile device has been lost. Hence adequate recovery mechanisms must be in place. Such mechanisms should also take into consideration that the same account might be used to access the services provided by the server also from web-based application, not only the mobile phone. Hence, changing a user account from a given location or device, should not compromise the other means of access.

G. Technological constraints

An important factor that can influence the design of a security solution for MDCS, is the type of technology used to implement them. Such solution should be compatible with the existing systems.

1) Limitations of J2ME: As mentioned earlier, we only consider systems based on J2ME technology. This means that any security solution proposed must be compatible with this platform. Unfortunately, unlike more advanced and high-end technologies such as iOS, Blackberry or Android, J2ME supports only one optional crypto API [18] which most low-end phones do not implement. So external security providers must be used to guarantee wide compatibility.

Another issue when using cryptography on a mobile phone is the generation of good random keys, since mobile phones do not have good sources of entropy, and even if they have, J2ME might lack the necessary libraries to access them.

2) Integration: Implementing security in existing and well-established application, can be a challenge because only few would be willing to heavily refactor their application to include security and more importantly, to take the burden of maintaining the new solution upon themselves. They might even be willing to take the risk to run insecure software, rather than facing the cost of securing it. Hence, a security solution for existing MDCS should be flexible enough to be integrated into these systems with as little effort as possible. This can

translate into a trade-off between ad-hoc solutions that might be especially efficient for specific clients, and more generic approaches that guarantee wider compatibility.

H. Battery consumption

A very important aspect of remote data collection, is that mobile devices should have long battery life, since, in some remote regions, electricity might be a scarce resource, and it might not always be possible to charge them whenever needed. Therefore, if a proposed security solution consumes too many resources and shortens considerably the battery life, it might render mobile data collection quite simply an unfeasible alternative. In this paper computation intensive operations are considered carefully for their impact on performance, and therefore on battery life.

IV. Analysis of Available Solutions

From the discussion in the previous section we can see that the main security concerns in MDCS are the confidentiality of the data and their availability, while keeping costs and hardware requirements as low as possible, and without compromising usability. Altogether, not a trivial combination to satisfy. In the next sections we discuss whether existing solutions manage to satisfy all these requirements.

A. Cryptographic APIs

As we have mentioned in Section III-G1, J2ME does provide some cryptographic APIs, but they are currently supported by very few mobile phones. The most widespread alternative are the Bouncy Castle lightweight APIs [10]. Their main drawback is their large memory footprint, but on the other hand they are free, very comprehensive and constantly updated. Therefore, although this solution is not optimal because of the footprint, it might be the only option for those who have budget constraints and do not want to implement their own cryptographic algorithms, since all other J2ME compatible crypto libraries are commercial.

B. Secure Communication

The Hypertext Transfer Protocol Secure (HTTPS) is a protocol designed to send HTTP requests and responses in a secure manner through the SSL/TLS [23] protocol and it is a standard way of securing client-server transactions. In fact, all MDCS we reviewed can actually use it. However, we have also mentioned that HTTPS might not be the best solution given all the constraints discussed in section III. Here we explain why. The security of HTTPS depends mostly on two things: valid certificates signed by a trusted Certificate Authority (CA), and a correct implementation of the protocol. Both of these conditions are not equally easy to satisfy on all mobile phones.

The first problem is the certificates. While MDCS that offer both software and service, have a centralized server and can secure the traffic to and from all the clients by using a single certificate, each project that wants to set up its own server will have to buy its own certificate. This is not cheap, especially for certificates signed by the most well-know and well-supported CAs (Certificate Authorities). Besides, the list of supported CAs preinstalled on each phone is controlled by the manufacturer and it is not standardized. Thus, in order to guarantee compatibility, certificates signed by different CAs might be necessary, thus adding to the cost. A cheaper alternative might be to use self-signed certificates, but they do not provide the same security level as CA signed certificates, and not all mobile phones accept them. See also [28] for a discussion on this subject.

Secondly, even assuming that the project is willing to take on the burden of using CA signed certificates, there is no guarantee that the implementation of the secure protocol using them will be equally secure on all devices they deploy [22].

We should also think that the HTTPS protocol was designed to be used on the World Wide Web where clients need secure communication for transactions with various unknown servers. In our case we know both who the server and the users are in advance, hence we could 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.

Such lighter protocols have been proposed previously [6], [21]. They are based solely on symmetric cryptography and are built on different assumptions than those discussed here, but some of the underlying ideas are still valid for our purposes, and we will compare them more in detail to our proposed solution in Section V.

C. Storage

J2ME does not offer any kind of encrypted storage. Hence, any solution for encrypting stored data would have to be implemented from scratch building on some external cryptographic API. Most of the examples we found in the literature, and in the MDCS we reviewed at the beginning, are very straightforward: a key is created from the user password (or worse pin code as in [6], [21]) through some hash function, and it is used to directly or indirectly encrypt the data in the RMS with symmetric encryption. Although the user password must be somehow involved in the process, we think that the storage scheme and the algorithms used should be more advanced, in order to protect data more effectively while allowing password recovery and multi-user management.

D. Authentication

There are various standardized authentication protocols available that do not require certificates and that provide mutual authentication and various protection mechanisms. However, in order to achieve this high level of security, they are also quite expensive to perform, both traffic and computational wise, and might not be adequate for the MDCS we consider.

V. Proposed solutions

In this section we outline both a scheme to secure the data on the mobile phone and a solution to secure data transmission

between client and server if HTTPS is not available, considering the limitations discussed in Section III. We also consider issues such as the secure distribution of the application, its configuration and the management of the cryptographic keys.

A. Secure Storage and Authentication

Summarizing the discussion from the previous sections, we can identify the following desired requirements that should be integrated in the design of secure storage:

1) Confidentiality (encryption)
2) Authorization (users can only access their own data)
3) (Off-line) authentication
4) Password and data recovery
5) Password changes should account for the possibility of using the same credentials to authenticate on different phones
6) Data should be adequately protected also if all information stored on the phone is available to an attacker
7) Breaking the storage encryption should not compromise the rest of the system

Here we assume that user credentials consist of a unique username and a password, and that to each user (on a given device), a different encryption key is assigned, so that a successful authentication grants direct access to this key and the data encrypted with it, but nothing else. Below we discuss some possible ways to implement this approach and discuss whether they satisfy the requirements we have identified.

- The user is authenticated by computing a hash of the password and comparing it to the hash stored in clear on the phone. The same hash is also used as the user's encryption key (or to derive it). This allows separate encryption for each user (requirements 1 and 2) and off-line authentication (requirement 3), but the authorization mechanism prevents unauthorized access only as long as data are accessed through the application. If an attacker can copy the phone memory, then the stored hash can be used to decrypt the data directly, just as it can be used to recover it if the password is lost (requirement 4). Also, the password can be easily brute forced if the hash is not strong enough and then it can be used to impersonate the user to the server, compromising the rest of the system (requirement 7).
- An easy improvement to the above solution is not to store the hash of the password. Authentication is then performed by verifying that the key derived from the password can correctly decrypt the data. This can be done by employing some kind of integrity check on the data, like appending a digest to the data before encryption, or adopting algorithm that can reliably detect errors caused by the use of a wrong key. This approach would satisfy also requirement 6 if the key is generated with a reasonable number of iterations of the hash process, but makes the recovery of the password and the data impossible unless a copy of the derived key or the password itself is stored somewhere and it is accessible through some other form of authentication.
- Recovering the password or the derived key is not a trivial problem. This information would have to be stored somewhere and the user should have an alternative authentication mechanism to access it, creating a circular problem. In any case, when the encryption key is derived from the password, even though we had a way to recover the data, a password change would require to re-encrypt the whole storage since also the encryption key must be changed. This problem can be solved by creating an encryption key that is not directly derived from the password, but is instead encrypted separately with a password-derived key. This would allow a decoupling of authentication and encryption, making it easier to change the password without changing the encryption key.
- Combining the previous solutions we would satisfy almost all our requirements, except for 4, 5 and 7, as long as the same password is used both for accessing the phone and the server. In fact changing a password on the server from one phone, would mean that the old password must still be used to login locally on other phones, creating potential synchronization problems. Also, if one phone encryption is broken and the password recovered, the attacker could impersonate the user to the server and gain access to the rest of the system. One approach that could give a satisfactory solution from a security perspective, is to separate completely the local authentication from the server authentication. In other words, the user should have two different passwords: a server password and a mobile password. In this way, requirement 4 would be satisfied by storing a copy of the encryption key on the server, and if the mobile password is forgotten, the user could login on the server, retrieve the key and reset the mobile password. Requirement 7 would be satisfied since no trace of the server password could be found on the mobile phone, neither explicitly (the hash of the password), nor implicitly (some keys are still derived from the password, so it is still possible to guess the password, generate a key and verify whether it is correct just as the authentication mechanism does). Also, changing the server password could be done centrally on the server, without compromising any of the local accounts of the user, hence satisfying requirement 4. The drawback is, of course, that the user has to remember an extra password.

Figure 1 shows a solution that satisfies all given requirements based on the previous discussion.

The user will have to register on the phone the first time, and this requires remote authentication on the server with the server password. If the authentication is successful, then an encryption key can be created for the specific user, either by the server or the mobile phone itself, and the user will be asked to choose a new mobile password to access the encrypted storage. In either cases, a copy of the encryption key will be stored on the server. A master key is then created from the mobile password and used to encrypt the encryption key together with its digest. Authentication can now be performed

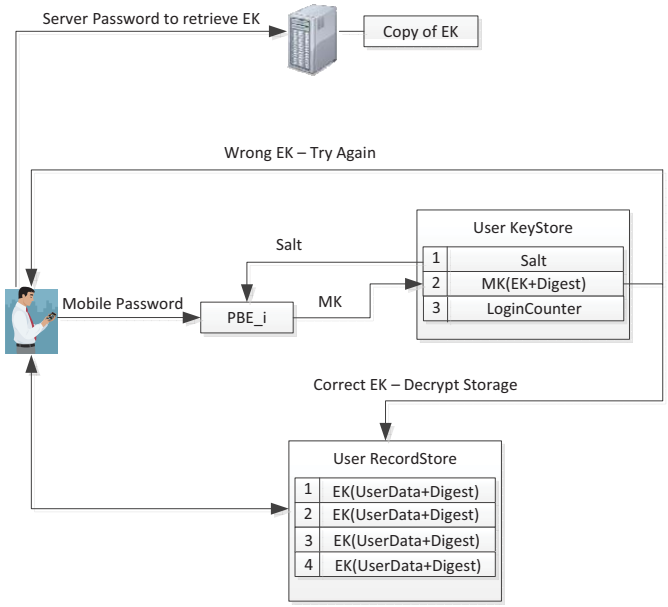

Fig. 1. Storage encryption scheme and authentication. MK=Master Key, EK=Encryption Key, PBE_i= PBE algorithm with i iterations. The notation Key(data) stands for data encrypted with Key.

by decrypting the encryption key with the master key derived by the mobile password, and checking that it matches both the decrypted key and the digest appended to it. This ensures that the authentication is also independent of the encryption algorithm.

How difficult it is to break the data encryption depends on the strength of the user password and the key derivation algorithms used. There is not much that can be done to enforce a very strong password while keeping it secure. It is common knowledge that users will just write it down somewhere or forget it. A good compromise could be to require an alphanumerical password of a given minimum length, or a pass phrase, while using an algorithm that could thwart a brute force attack for as long as possible, but without compromising usability. For this purpose, simple MD5 hashes or even a salted SHA-1 hash might not be enough, so we recommend to use a PBE (password based encryption) algorithm [7], with as many iterations as possible, based on the computation power of the phone.

If authentication fails enough number of times, the encryption key of the user should be deleted from the phone, hence forcing a recovery procedure involving remote authentication on the server. Also, keeping the keys in a store separate from the data (a keystore), and different user stores separated from each other, would add flexibility when implementing this solution.

Issues regarding the generation of keys on the mobile and how to communicate the storage key between client and server, will be discussed later in Sections V-B4.

B. Secure communication

There are many standard algorithms that can be used to encrypt the data traffic, but if we do not use HTTPS and forego the PKI infrastructure, then we have to find an alternative way to establish trust between server and client before any encryption can take place. This translates into a secure key distribution and reliable authentication. In [6], [21], this problem is solved by distributing the symmetric keys used to encrypt the communication, already embedded in the application which is distributed to the users. We do not follow this approach.

In the next section we will discuss why and how we can solve the distribution problem.

1) Distribution, installation and configuration: Typically J2ME applications consist of two files: a JAR file containing the application itself, and a JAD file containing a set of attributes pertaining the application. The application can be installed by sending the JAR file directly to the mobile phone, through over-the-air (OTA), Bluetooth, WiFi or cable, or it can be installed through the JAD file. In the last case, the JAD file will have to contain some mandatory attributes that must match with those in the manifest file contained in the JAR, plus optional and custom attributes. Among the required attributes is the URL of the JAR, so that the phone can download it automatically through an Internet browser. In addition, one can also sign the JAR file with a code signing certificate, and add the signature as an attribute in the JAD file. This allows to verify that the JAR file downloaded is indeed the same indicated in the JAD file, and that the entity distributing it, is a trusted one. Moreover, code signing protects the application by allowing only signed software to update installed application, so that it cannot be tampered with after installation.

If the JAR file is not signed, then there is no guarantee that the application being downloaded is genuine. Someone might have tempered with it, turned off its security features, or added a malware in it, and we do not have any way of detecting that. Thus, signing the JAR file containing the client application, is a necessary condition to guarantee any kind of security. Currently, none of the systems we reviewed has a signed client application, except for CommCareHQ.

However, the fact that the JAR file is signed by the entity who has developed the application, can prevent a project that uses a freely available client like openXdata to securely distribute application and user specific settings inside the signed JAR file as they do in [6], [21]. In fact, this is possible only if the one that distributes the application is the same entity that can sign it, so that a customized JAR can be created and signed dynamically for each download or installation. An alternative method to distribute this information, is through custom attributes in the JAD file before the application is installed or through a secure request/response mechanism after the application is installed. In the first case, the challenge is how to protect the JAD file during distribution.

One solution can be to download the JAD file through a secure HTTPS connection as shown in Figure 2. In this case, a project (the Application Service Provider) wishes to use a

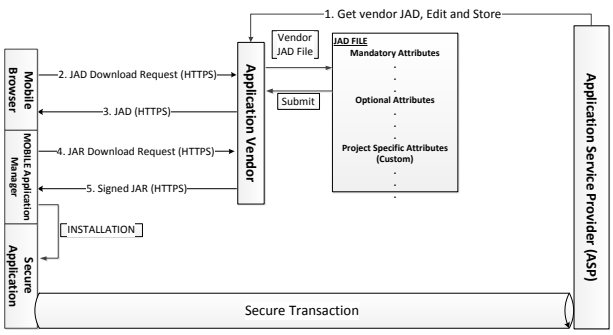

Fig. 2. Example of centralized secure distribution.

specific application developed, signed and distributed by some Application Vendor (such as openXdata). Then the project could register a project specific URL for download on the Application Vendor server and define some custom attributes that the Application Vendor should generate and put in the JAD file for each download. Thus, both JAD and JAR file can be downloaded securely from the Application Vendor server, and we would not incur many of the problems described in Section IV-B, because it is the mobile browser that downloads the JAD, not the J2ME application, and browser support for HTTPS and certificates is often better than J2ME (although some security concerns exist also in this case). Also, regarding costs, notice that the certificate used for application download can be unique for all projects and offered by the Application Vendor. After a successful installation, the client will have enough information to establish a secure connection with the project server directly.

The drawback with this approach is that the Application Vendor must be trusted with potentially sensitive information, and the URL of the JAD must be sent to the client securely, or an attacker could redirect the download to a different JAD file with different settings that could compromise security. One way to mitigate this type of threat can be to add a generic URL like `https://m.openxdata.com/` in the signed manifest file, while the JAD contains only one non-signed attribute, i.e, the project name. At this point the settings can be downloaded by the application directly after the installation at the URL `https://m.openxdata.com/-project_name`, after the user manually verified the project name.

If a centralized secure distribution is not available, one might use a pre-shared secret (PSK) approach, that allows to authenticate simultaneously both client and server. A one-time activation code could be distributed to the collectors and used to establish, after the installation, a secure connection for the download of configuration settings. In this case the keys and settings could be downloaded directly from ASP.

A drawback of the PSK approach, is the difficulty in distributing many activation codes, especially if there are thousands of collectors, and the difficulty of entering a complicated

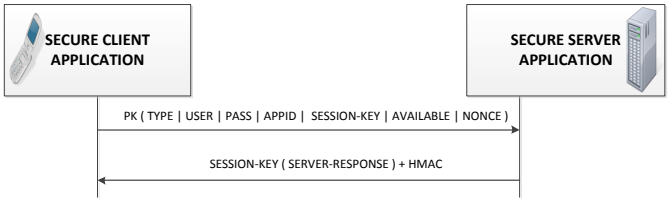

Fig. 3. Example of a generic secure service request/response

and long key on a mobile phone. So, a third option, which is also used today, is to deliver the phones to the collectors pre-configured by someone responsible for this specific task, so that installation can be done manually.

2) Public key cryptography: Given that the application has been installed and configured, and we have some key to start encrypting our traffic, the next step to consider is how the communication should be conducted from this point onwards. Given that we do not use HTTPS, we want to achieve the following:

1) Confidentiality
2) Mutual authentication
3) Minimal traffic time and traffic amount
4) Protection against replay and MITM attacks
5) Data integrity
6) Key generation and management
7) Compatibility with existing applications

If a server had a public key, then all clients could initiate a secure communication by encrypting the first request with this key. The server could, then, decrypt all incoming requests using a single private key, without knowing whom they are from or what they are about beforehand. This has multiple advantages, besides a simplified key management from the server side and the fact that practically no sensitive information needs to be communicated in clear (like the user name and application-id to identify which specific key to use). Mutual authentication can also be easily achieved by including the user's credentials in each request (given that the public key was downloaded securely with the application settings, since we do not use certificates), eliminating the need for a dedicated authentication protocol. MITM attacks would be minimized, since we do not receive the public key each time from the server, but we use the one stored on the phone. Hence, requirements 1, 2, 5, 6 and 7 would be satisfied and the format of a client request and server response could then look like the one shown in Figure 3.

The TYPE field is used to identify the type of secure service request to the proxy server. The request could be user registration, or a password recovery, a data transfer (upload or download) or a session establishment. By knowing this, in most cases the server could complete the requested operation directly, without requiring further exchange of messages, thus satisfying 3. Also, an attacker cannot distinguish different request types just by eavesdropping the traffic.

The SESSION-KEY can be used directly to encrypt the response (requirement 6), or in combination with NONCE to

derive a different key. This can be useful when sessions are used as explained in the next sub-section. NONCE can also be used to identify a specific request and buffer the corresponding response in case of retransmission.

The AVAILABLE FIELD is used for parameters that are specific for each request TYPE. For example, a User Registration request might contain a back-up copy of the key used to encrypt the user's data on the phone.

We should point out that the public key of the server is not secret and the parameters in the HTTP request are not negotiated, but retrieved from runtime memory after login and directly sent by the client. Hence the user's credentials must always be included to guarantee the authenticity of the request or anyone could forge a request and obtain services or information from the server. The fields USERNAME, PASSWORD and APPLICATION-ID are there for exactly this purpose. The APPLICATION-ID is an identifier unique for each instance of the client application, i.e., for each mobile phone. This is needed to identify the specific user account, since the same user might have activated an account on different phones. We suggest to assign this ID to the client at initialization time, so that the server can guarantee its uniqueness.

When it comes to the possibility of replay attacks (requirement 4), a request is executed by the server only the first time. After that, if the same request is received again (with the same NONCE and content), no further action should be taken, but only a buffered copy of the response should be sent back. This allows also the client to re-send a buffered response, and avoid performing public key based encryption again. Therefore, a replay attack would be identical to a legitimate client request, but it would not gain the attacker anything more than what could be eavesdropped from the network.

Notice that we use a public key only for the server, not for the clients. We do this because it would be problematic to distribute user specific private keys, and very difficult to keep them secure on the mobile phone, given the means at our disposal and the problems discussed in Section III-C. This excludes the possibility of signing the user's messages for integrity and accountability purposes, and for the server to initiate a secure communication with a specific user or device.

Dealing with the transfer of large amounts of data, like upload or download of forms, and how to satisfy requirement 7, is discussed in the next subsection.

3) HTTP Sessions and Data Transfer: A public key encrypted request can be used to establish a session, and obtain a HTTP session-Id from the server. The server maps the HTTP session object to a user session key sent by the client and sends an HTTP session ID to the client. The client uses this session-Id for further interaction with the server, by including it in a cookie with each new request as it is normally done with HTTP sessions. However, unlike HTTPS, we are encrypting the traffic at the application layer, and the cookie will have to travel in clear. Still, hijacking the session would require the attacker to know both the encryption-key and the authentication-key currently used by both client and server, and replay attacks can be prevented by adding a unique sequential number to each request.

Since a client using our encryption strategy might already have its own (unencrypted) communication protocol and use some particular HTTP request to send information, we propose to encapsulate the whole HTTP request and send it through a secure tunnel, and possibly add a standard secure HTTP header in a fashion similar to that of S-HTTP [20]. The integrity and authenticity of both the secure header and the encrypted data (requirement 5) is protected by an HMAC.

This approach satisfies also requirement 7, since it allows a separation between the application request format and the cryptographic solution adopted, although the price to pay for this flexibility is an extra HTTP header. More efficient protocols could surely be designed if tailored for a specific application, but that would be more prone to error and not interesting for securing other existing MDCS.

4) Key generation: In the protocol we propose, it is the mobile phone that has to generate all symmetric keys, since it is the one that initiates all communication and that can use the server public key for transmitting the keys securely. However, the standard pseudo-random number generator (PNRG) provided by J2ME is seeded simply by the internal clock of the phone, and it is well-known that this does not provide enough entropy to generate high quality cryptographic material [22]. Our proposed solution is to generate high quality seeds on the server and send them regularly to the client embedded in a response, and use an alternative PRNG to the one available with J2ME.

VI. Current Status of the Research

We have implemented the solution proposed in this paper as an API called secureXdata [25]. The API has been tested intensively on emulators and mobile devices like the Nokia 2330c, which fits the low budget criteria outlined by openXdata (cost below 50 USD). The benchmarks reported in [25] suggest that the overhead due to the cryptographic operations should not affect the user experience significantly, at least regarding the secure storage. The performance of the secure communication should be further tested in the field, in order to account for the possible connectivity problems mentioned in Section III. We have also confirmed the flexibility of our API by retrofitting the existing openXdata client with secure storage and communication by modifying only a few lines of code in the current application [25].

The initial tests have conviced the openXdata community to adopt our API to develop the future version of their client.To begin with, secure storage will be prioritized in accordance with OWASP Mobile top 10 mobile risks[19]. In particular, the modularity of the API faciliates integrating just the secure storage with the openXdata client, with minimal changes to the existing client code. The only modification required is that the hash of the users password currently stored on the phone to perform local authentication, is instead used to generate the key that wraps the storage encryption key. In this case, only one user password will be used, both for local and remote login, and the storage encryption key can be recovered by

retrieving the hash of the user password from the server. So, if the server password is lost, the administrator will issue a new one and communicate it to the user by any means available. The user will then use the new password to login on the server and retrieve the old password hash, that can be used to decrypt the storage encryption key on the phone. Once decrypted, this key will be re-encrypted with the new password. Although we cannot achieve complete separation between server and client authentication, recovery is still possible and the storage encryption key is never sent over the network. Besides every user has his or her own secure storage, and other services like authentication, recovery and user registration procedures will be provided directly by the API, without need for re-implementation.

In our default implementation we used RSA with a 1024 bits key, and for symmetric cryptography we chose AES in CBC mode with a unique initialization vector per encryption, associated with a HMAC based on the SHA1 hash scheme.

VII. Conclusions and future works

We have presented an assessment of the security of systems for remote data collection on Java enabled low-end mobile phones, and have identified the specific challenges for their deployment and how these affect their security needs. Based on these observations we have outlined and discussed some possible solutions that are particularly appropriate to secure these systems.

The API we implemented confirmed also that the schemes we proposed are feasible in practice and quite flexible, and they can be tailored for clients with specific additional constratins by possibly dropping some of the requirements we identified. Besides, the analysis we provide of the different solutions can help assessing the security risk of dropping a requirement in favor, for instance, of a smoother integration and usability as in the case of the secure storage solution for the current openXdata client described in the previous section.

For instance, if one already uses HTTPS and it works well for the purpose, then only the storage scheme can be adopted. If a risk analysis, instead, shows that it is an acceptable risk to use the same password both for the server and the mobile phone, or we can assume that each mobile phone is personal, then it is possible to adapt the protocol to use only one password. We do not claim that these are the best or the only possible solutions, but we think that they are general and flexible enough to derive implementations that can be adapted to MDCS with a wide range of requirements.

Future work will prioritize the integration of our solutions with openXdata, and in field deployments where we will further investigate the impact of our API in low-budget remote data collection based projects.

VIII. Acknowledgments

We would like to thank Brent Atkinson for his comments and for helping in improving the presentation of the paper.

References

[1] CommCareHQ. http://www.commcarehq.org. Online, Accessed November 2011.
[2] DataDyne. http://www.datadyne.org. Online, Accessed November 2011.
[3] T. Egeberg. Storage of sensitive data in a Java enabled cell phone. Master's thesis, Høgskolen i Gjøvik, 2006.
[4] 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, 1999.
[5] S. Gejibo, K. A. Mughal, F. Mancini, J. Klungsøyrg, and R. B. Valvik. Challenges in implementing end-to-end secure protocol for java ME-based mobile data collection in low-budget settings. In *To appear in ESSoS 2012 - The Fourth International Symposium on Engineering Secure Software and Systems*.
[6] 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.
[7] B. Kaliski. RFC 2898 - PKCS #5: Password-based cryptography specification, 2000.
[8] J. Klungsøyr. 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.
[9] J. Klungsøyr, T. Tylleskar, B. MacLeod, P. Bagyenda, W. Chen, and P. Wakholi. OMEVAC - open mobile electronic vaccine trials, an interdisciplinary project to improve quality of vaccine trials in low resource settings. In *Proceedings of M4D '08 - The 1st International Conference on Mobile Communication Technology for Development*, pages 36–44. Karlstad University Studies, 2008.
[10] T. Legion Of the Bouncy Castle. http://www.bouncycastle.org/. Online, Accessed March 2011.
[11] A. Melnikov. RFC 6331 - Moving DIGEST-MD5 to Historic, 2011.
[12] Mobenzi Researcher. http://www.mobenzi.com/researcher/. Online, Accessed November 2011.
[13] Nokia. https://github.com/nokiadatagathering/ndg-mobile-client. Online, Accessed September 2011.
[14] Nokia Data Gathering. http://projects.developer.nokia.com/ndg. Online, Accessed November 2011.
[15] Open Rosa. Javarosa. http://www.javarosa.org. Online, Accessed March 2011.
[16] openXdata. http://www.openxdata.org. Online, Accessed March 2011.
[17] Oracle. Java ME reference. http://www.oracle.com/technetwork/java/javame/index.html. Online, Accessed March 2011.
[18] Oracle. Security and trust services API for J2ME (SATSA). http://java.sun.com/products/satsa/, 2006. Online, Accessed March 2011.
[19] O. M. S. Project. https://www.owasp.org/index.php/OWASP_Mobile_Security_Project. Online, Accessed January 2012.
[20] E. Rescorla and A. Schiffman. RFC 2660 - The Secure HyperText Transfer Protocol, 1999.
[21] S. M. A. Shah, N. Gul, H. F. Ahmad, and R. Bahsoon. Secure storage and communication in J2ME based lightweight multi-agent systems. *Proceedings of KES-AMSTA'08 - the 2nd KES International conference on Agent and multi-agent systems: technologies and applications, Incheon, Korea*, pages 887–896.
[22] 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.
[23] I. Society. RFC 2818 - http over tls, 2000.
[24] M. Tomlinson, W. Solomon, Y. Singh, T. Doherty, M. Chopra, P. Ijumba, A. Tsai, and D. Jackson. The use of mobile phones as a data collection tool: A report from a household survey in south africa. *BMC Medical Informatics and Decision Making*, 9:51, 2009.
[25] R. A. B. Valvik. Security API for java ME: secureXdata. Master's thesis, Department of Informatics, University of Bergen, 2012.
[26] Vital Wave Consulting. *mHealth for Development: The Opportunity of Mobile Technology for Healthcare in the Developing World*. Washington, D.C. and Berkshire, UK: UN Foundation-Vodafone Foundation Partnership, February 2009.
[27] Voxiva. http://www.voxiva.net/services/technologyPlatform.html. Online, Accessed November 2011.
[28] B. Whitaker. Problems with mobile security #1. http://www.masabi.com/2007/07/13/problems-with-mobile-security-1/, July 2007. Online, Accessed March 2011.