Secure Data Storage for Java ME-Based Mobile Data Collection Systems

S. H. Gejibo, F. Mancini, K. A. Mughal and R. A. B. Valvik,
Dept. of Informatics, University of Bergen
Bergen, Norway
Emails: {samson.gejibo, federico.mancini,khalid.mughal}@ii.uib.no, remi@valvik.org
J. Klungsøyr
Centre for International Health, University of Bergen
Bergen, Norway
Email: {jorn.klungsoyr}@cih.uib.no

***Abstract*— The falling price of mobile devices with high mobile subscriber penetration rate provides an opportunity to create an affordable mobile solution to disseminate and gather health related information in remote areas in a real time. These solutions come with data security and privacy challenges, and many existing mobile data collection systems (MDCS) do not systematically address very important security issues which are critical when dealing with sensitive and private health information. Insecure storage, in particular, is one of the major challenges in mobile security. In this paper, building on our previous work, we present a solution to storage security issues for data collection systems running on the Java ME platform. Our secure storage scheme is flexible enough to be integrated in existing mobile client applications without requiring significant changes, and offers a cost-effective way for ensuring data protection and authorization, account and data recovery mechanisms, and multi-user management. We use openXdata as our reference MDCS and the secure solution has been successfully integrated and evaluated in a prototype system.**

***Index Terms*—mHealth, MDCS, OWASP, J2ME, RMS, Secure Mobile Data Storage**

I. Introduction

The usage of mobile phones, PDAs and other mobile communication devices in the context of health is an emerging part of eHealth. A report by the United Nations Foundation and Vodafone Foundation [16] suggests that close to half the population in low-income countries own or have access to mobile phones. Healthcare in these nations can be scarce or difficult to access due to restraints such as limited resources, finances and healthcare workforce, or parts of the population living in remote locations. High mobile phone penetration makes mHealth a viable option for providing better healthcare through Mobile Health (a.k.a mHealth) systems. In this paper we are mainly focused on a specific aspect of mHealth, namely remote data collection. However, the work presented is applicable in any context where secure data storage is required.

A. Mobile Data Collection Systems

Mobile Data Collection System (MDCS) allow the collection and transmission of data from remote geographical locations to centrally located 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 an accessible server database. Collectors can then download these forms on their mobile device and use them to collect the actual data on 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 handle a lot of sensitive and private information, especially when used for medical purposes, and they should provide an adequate level of protection to the data at all times. Insecure data storage is one of the major challenges and ranks at the top of the OWASP Top Ten Mobile Security Risks. [14].

The work we present here originated from a collaboration with *openXdata* [11], an open-source MDCS used to run health related projects in low-income countries and has been deployed in different countries such as South Africa, Pakistan, and Uganda [6]. 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 they had found better 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 Java ME [12] client, like openXdata. Java-enabled phones are, in fact, more likely to be used for low-budget projects than smart-phones or PDAs.

B. Security Reviews of Java ME-Based MDCS

We have compared the solutions adopted by some Java ME based MDCS systems to protect the data both in transit between client and server, and when stored on the mobile device. 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.

II. Proposed Secure Data storage Solution

A. Security Requirements and Practical Constraints

The practical constraints of feature phones such as limited CPU power, working memory, persistent storage, battery power, screen size, and input mechanisms might preclude strong cryptography that is needed to guarantee a high level of security. Most reviewed MDCS projects have been targeted low-income countries where the infrastructure for mobile communication and Internet access are not yet fully developed.

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 a user and that most of the data collection might have to be done off-line. From a security perspective this translates in the following requirements:

1) Confidentiality (encryption)
2) Authorization (users can access only 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 reasonably 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

B. Proposed Secure Storage Solution

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

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 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.

How difficult it is to break the data encryption depends on the strength of the user password and the key derivation algorithms used. A good compromise could be to require an alphanumerical password of a given minimum length, or a pass phrase, while using algorithms that could slow down as much as possible a brute force attack, but without compromising usability. For this purpose, we recommend to use a password based encryption (PBE) algorithm as described in [5], with as many iterations as possible, based on the computation power of the phone.

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

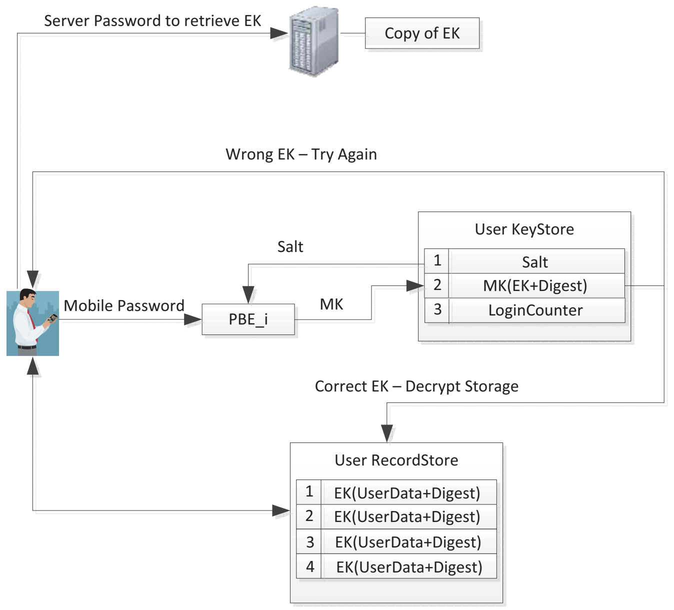

Fig. 1: Storage encryption scheme and authentication. MK=Master Key, EK=Encryption Key, PBE_i= PBE algorithm with i iterations. The notation Key(data) stands for data encrypted with Key.

III. Implementing Secure Storage Solution

A. Implementation Criteria and Challenges

The ultimate goal of this work is to create a working API implementation of the secure storage solution discussed in section II, this will be referred to as secureXdata or simply as the API. We identified some important API design criteria, which are discussed in this section.

1) Ease of Use / Transparent Design: There are multiple MDCS in use today, many of which lack proper security. If we are to be successful in getting any of these to adopt the API into their applications, integrating the API into an existing system needs to be as hassle free as possible. Trying to address this issue, we have focused on making the API design as transparent as possible.

2) Functionality Decoupling and Flexibility: The full API provides more than one functionality. Since each of these services cover different areas, they should be as decoupled as possible, preferably completely independent of each other. This makes the API very flexible since a programmer can use only the functionality that is needed, and not being forced to add unwanted functionality.

3) Low-end Device Requirements: As discussed in section I, the primary target for the API is low end (in terms of memory and CPU).

B. *Integration Approaches*

There are two main approaches for integration, these provide different degrees of control and responsibility for the programmer. If the programmer is knowledgeable when it comes to security, he or she can control how keys are stored and managed. If the programmer is not familiar with security design then this task can be handled by the API. Here, we discussed the most secure and applicable approach, i.e. Secure API control.

1) Secure API control: By extending the `AbsSecureClient` class of the API, an application can delegate the task of handling users entirely over to the API. Once the application starts, the screen control would be given to secureXdata. Depending on the configuration, the controller may prompt users to authenticate the server on the first run, ask them to register or just log in. Once the authentication is successful, control is given back to the application. At this point the `UserKeyStore`,`SecureUserRecordStore` have been initialized and are ready for use. By using this approach, the programmer is completely free from any security concerns. The API will take care of account recovery, logging users in, and initializing the different components with the correct keys. This however comes at the cost of the programmer being confined to the solution provided by the API, and having little or no control over how parts of the application works. It is believed that anything related to cryptography needs to be very flexible to be able to accommodate any requirements or shortcomings a device might have. Through the `CryptoTools` interface the API tries to address this by enabling the developer to make their own implementation using whatever algorithms or libraries they have available.

2) Bouncy Castle and the Default Implementation: The API comes with a default implementation of `CryptoTools`, this implementation uses BouncyCastle[7]. Since BouncyCastle is a third party API it should support any Java ME enabled phone with the capacity to load it. So the default implementation should work on any device. It also provides a large arsenal of cryptographic algorithms and primitives, making different implementations possible.

On the other hand, since it is not a native implementation there are down sides as well. Computational time and memory usage might be higher than what would be the case with a native implementation, this indirectly effects battery usage.

The default implementation uses RSA for public key encryption, AES in padded with CBC mode and initializing vector (IV) for symmetric cryptography, SHA1 digest, HMAC based on SHA1 digest, Password Based Encryption based on PKCS#12. A random generator based on the `SecureRandomGenerator` class provided by Bouncy Castle.

IV. Integration With OpenXdata

There are two things that we wish to accomplish: first and foremost we want to keep the data safe on the device, and secondly we wish to separate data from different users. OpenXdata has an issue with their storage system, if a user logs out, the next user logging in has access to the other users data. By using the `SecureUserKeyStore` class of the API both these issues can be solved. Since openXdata uses a centralized class for all storage and stores no data that are shared between users, very few changes were needed to secure storage.

In the current version of openXdata, they no longer store the entire user database on the device, only the registered users. Leveraging on this, we can use the `UserKeyStore` for storing the user credentials instead of storing them in a normal record store. When a user tries to authenticate, if a user's `UserKeyStore` exists, we try to unlock it using the supplied password and stored salt. If the password is entered three times incorrectly, the user is locked out and needs to recover the account by contacting the server. If a user's `UserKeyStore` does not exist, the user name and password are sent to the server for authentication, and, if successful, the salt is returned and the `UserKeyStore` is created.

The integration of the `SecureUserRecordStore` would be done in the same way as described in the prototype integration section III. Server communication would still be insecure, however providing confidentiality locally on the device is a step in the right direction. The client can make a secure connection to the server through HTTPS or some other secure protocol.

V. Evaluation of performance

While designing the API, the main focus has been to make it easy to integrate with existing systems and easy to use in general. Furthermore, decoupling between the storage and communication parts has also been important. In this section we will evaluate the cryptographic performance of the current design. That means using the default `CryptoTools` implementation that is implemented using BouncyCastle.

A. *Cryptographic Benchmarking*

We primarily used the openXdata model phone, Nokia 2330c-2 [9] for benchmarks and testing. Based on preliminary performance testing that can be found in [4], this phone has fair performance with processor speed of 4.7MHz while still being inexpensive (less than 50$). We will be benchmarking the encryption and decryption speed of the same setup as used in the default implementation of `CryptoTools`. Once we have the encryption and decryption times, we can calculate the throughput speed of both operations, and the combined speed for decrypting followed by encrypting. Decryption followed by encryption is what happens if data is read from a secure record store and then sent to the server through a secure channel, this is a result of the strong decoupling between storage and communication. However, the ideal solution should be to read the encrypted data from the store and perform the decryption on the server side inorder to reduce the computation.

If the speed for the cryptographic operations are much lower than the speed at which data can be transferred to or from the server, then this means that the cryptography is a bottleneck in the system, which we want to avoid.

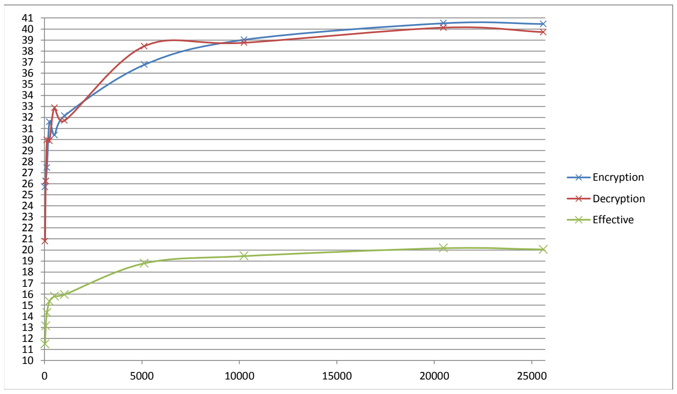

Fig. 2: A chart displaying the cryptographic speed results. The X axis is data in Bytes and Y axis is throughput in Kb/sec.

If the figure 2 results are acceptable or not, depends on the context in which the API would be used. How large are the data sets that are stored? What connectivity conditions will the application operate under? In the end it comes down to the individual project to determine if these results are acceptable. One thing we can however conclude is that the throughput speeds on the used device (Nokia 2330c) are reasonably high. Under optimal conditions a GPRS/EDGE [3] as of 2007 speeds up to 22 kb/s can be reached for upload. On a 3G network [1] higher speeds could be expected. It is however likely that the areas where the API and device would be used (low-income countries) would not have optimal conditions.

Since reliable data regarding what connectivity conditions to expect are scarce, we try to put the speeds into perspective using a different factor: the maximum size of a given record store on the device. This also provides some insight in how the throughput is when looking at the other specifications of the phone. The size can be found easily by using the `getSizeAvailable()` method on an empty record store. Doing so on Nokia 2330c shows that the record store can hold a maximum of 131072 bytes, or 128 kilobytes. These bytes would include any bookkeeping or structure overhead from the RMS system itself. Comparing this to the speeds of the larges benchmarked data set, we get the following times: 3,3s encryption, 3,2s decryption, or 6,5s for both. Waiting 3,3s to store data might be noticeable, but then we are talking about a full record store. The average English word length is somewhere around 8 characters [15]. Lets assume that due to spaces or other formatting characters the average length is 10. This means that a full record store corresponds to roughly 13100 words of text, that is a lot of text for a low-end mobile device. OpenXdata estimates a form being between 1 and 50 kb in size, usually a couple of kilobytes.

VI. Conclusions

Most modern smart phones equipped with operative systems like Blackberry, Android and iOS provide a crypto API to develop secure applications. However, we are developing a secure solution for the Java ME platform, which lacks support for any kind of data security [2], [17], and we target low-end phones, so that solutions that might be adequate for high-end phone like smart phones, are not an option in our context. The solution we implemented is based on a custom protocol developed by considering the specific constraints of MDCS [8], but it makes almost no assumptions about how or where data is stored, or how the communication layer of an existing application is implemented. This guarantees wide compatibility. Besides, the different secure solutions that it offers are very modular, and can be used independently to fit the needs of MDCS with different security requirements. We have also developed our own prototype MDCS 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 very low-end phones, and the openXdata integration is proceeding smoothly.

References

[1] 3rd generation mobile telecommunications(3G)., http://en.wikipedia.org/wiki/3G, online, Accessed December 2011.

[2] T. Egeberg, "Storage of sensitive data in a Java enabled cell phone," Master's thesis, Høgskolen i Gjøvik, 2006.

[3] Enhanced Data Rates for GSM Evolution(EDGE), http://en.wikipedia.org/wiki/Enhanced_Data_Rates_for_GSM_Evolution, online, Accessed December 2011.

[4] 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 *ESSoS*, ser. Lecture Notes in Computer Science. Springer, 2012, pp. 38–45.

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

[6] 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*. Karlstad University Studies, 2008, pp. 36–44.

[7] T. Legion Of the Bouncy Castle, http://www.bouncycastle.org/, online, Accessed March 2011.

[8] F. Mancini, K. Mughal, S. Gejibo, and J. Klungsoyr, "Adding security to mobile data collection," in *Healthcom 2011 - 13th IEEE International Conference on e-Health Networking Applications and Services*, june 2011, pp. 86 –89.

[9] Nokia 2330c classic, http://www.developer.nokia.com/Devices/Device_specifications/2330_classic, online, Accessed September 2011.

[10] Nokia, Nokia Data Gatherings(NDG), https://github.com/nokiadatagathering/ndg-mobile-client, online, Accessed September 2011.

[11] openXdata, http://www.openxdata.org, online, Accessed March 2011.

[12] Oracle, "Java ME reference," http://www.oracle.com/technetwork/java/javame/index.html, online, Accessed March 2011.

[13] Oracle Inc., "Security and Trust Services API for J2ME(SATSA)," http://java.sun.com/products/satsa/, online, Accessed March 2011.

[14] OWASP, "Mobile Security Project," https://www.owasp.org/index.php/OWASP_Mobile_Security_Project, online, Accessed March 2012.

[15] C. Z. G. N. W. unit based multilingual comparative analysis of text corpora., http://speechlab.tmit.bme.hu/publikaciok/, online, Accessed January 2012.

[16] 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.

[17] B. Whitaker, "Problems with mobile security #1," http://www.masabi.com/2007/07/13/problems-with-mobile-security-1/, MASABI, July 2007, online, Accessed March 2011.