Challenges in Implementing an End-to-End Secure Protocol for Java ME-Based Mobile Data Collection in Low-Budget Settings

Samson Gejibo[2], Federico Mancini[1],
, Khalid A. Mughal[1], Remi B. Valvik[1], and Jørn Klungsøyr[2]

[1] Department of Informatics, University of Bergen, Norway
{federico,khalid}@ii.uib.no,remi@valvik.org
[2] Centre for International Health, University of Bergen Norway
{samson.gejibo,mihjk}@cih.uib.no

Abstract. Mobile devices are having a profound impact on how services can be delivered and how information can be shared. Sensitive 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. However, many of these systems do not systematically address very important security issues which are critical when dealing with such sensitive and private information.
In this paper we 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.

Keywords: Mobile Data Collection Systems, Mobile Security, secure communication protocols, secure mobile data storage, secure mobile data transmission, Java ME, openXdata, HTTPS, JAD.

1 Introduction

There are already a number of systems that allow data collection in the health sector using mobile phones and provide a server component to manage the collected data. 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 side. In this paper, we present the challenges in implementing the *openXSecureAPI* (which from now on we will refer to as simply API), based on the secure protocol proposed in [5], which can be used to add a security layer in existing Mobile Data Collection Systems (MDCS).

Here, we focus on the mobile side of the API, which is developed for Java Mobile Edition(Java ME)[7] based applications 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 gathered by automated sensing systems. The API is designed by considering several security challenges in mobile data collection where low-end mobile phones are deployed and the projects run on very constrained budgets. The details can be found in [5].

For this work we collaborated with openXdata [6], a MDCS that is primarily designed for data collection using low-end Java-enabled phones in low-budget settings. The openXdata community shared with us their field experience regarding the deployment of their mobile data collection tools and various technical details of their client and server applications.

The challenges and solutions are covered in Section 2, where, in order to make this article self-contained, we also mention how the different parts of the API reflect the underlying protocol, and which security and usability requirements are addressed. Finally, we present some experimental results we obtained by testing a basic data collection client that uses our API on various mobile devices. In this paper we assume that the reader is familiar with Java ME technology and terminology.

2 Implementation Challenges and Solutions

Most of the challenges we faced during the implementation required finding the right balance between flexibility, efficiency and usability, while not compromising security. In general, we decided to give more emphasis to flexibility, in order to create an API that is easy to use and integrate with different clients, at the cost of some efficiency. In the following sections, we discuss some of issues we consider to be highly relevant.

2.1 Cryptography API Providers

Early versions of Java ME did not support a cryptography API. However, since the introduction of MIDP 2.0, the Security and Trust Services API (SATSA) has been developed and added to the Java ME platform as an optional package that provides some basic cryptographic primitives. Besides, since it is implemented as part of the phone libraries, its use does not affect the memory footprint of the application. Unfortunately, very few low-end mobile phones actually support it. On the other hand, Bouncy Castle (BC)[4] provides a flexible lightweight cryptography API which is extensively used in Java ME applications. Since it is an external API, it allows us to develop device independent solutions, but its libraries can add a significant memory overhead.

In order to allow for future compatibility, we opted for an hybrid solution. Our API provides an interface that defines the required cryptographic operations, but leave the actual implementation open, with BC as default provider. However, if

the phone supports the SATSA package, our API can automatically switch to that implementation. So, even though memory footprint is not reduced (BC is always loaded anyway), one can gain in performance by using the phone built-in libraries. Using two different implementations, means also that we are forced to use only algorithms supported by both libraries. In particular: RSA for public key encryption, AES in padded CBC mode with initializing vector (IV) for symmetric cryptography, SHA1 digest, Hash-based Message Authentication Code (HMAC) based on SHA1 digest. Only BC provides an adequate Pseudo Random Number Generator (PNRG) and Password Based Encryption based on PKCS#5.

2.2 Key Generation

A critical 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 [1], and even if they have, J2ME might lack the necessary libraries to access them. In the proposed solution, we generate a strong seed on the server and send it securely to the client whenever possible, so that strong cryptographic keys can be generated. Every user will have their personal set of seeds stored encrypted in their key store, so that the PNRG can be seeded also at boot time, and in a different way for each user. This solution avoids putting the burden of generating the seed on the user by pressing random keys or playing a game, or turning on the camera or the microphone to collect entropy, as it has been suggested in the literature.

2.3 Secure Data Upload and Download

The API is designed to be flexible and support both HTTPS and the protocol proposed in [5]. We offer a `SecureHttpConnection` class that can be wrapped around a `HttpConnection`. If the connection is HTTPS, the `SecureHttpConnection` will behave in the same way as a normal `HttpsConnection` object would. If however it is not HTTPS, any request headers or data written to the connections output stream will be encrypted prior to being sent to the server by using the protocol presented in [5]. The API is designed so that the client developer would use the `SecureHttpConnection` object in the same way as an `HttpConnection` object. This makes for easy and transparent integration into existing systems. We are able to create a secure tunnel by changing only two lines of code in the existing openXdata client. The following snippet shows openXdata client before the integration (no encryption is used):

```
HttpConnection  con = (HttpConnection)Connector.open(URL);
((HttpConnection)con).setRequestMethod(POST);
```

This snippet shows openXdata client after integration with the secure API:

```
HttpConnection con = (HttpConnection)Connector.open(URL);
SecureHttpConnection secCon = new SecureHttpConnection(con,
                          SecureHttpConnection.RequestType);
secCon.setRequestMethod(POST);
```

By using the `SecureHttpConnection` class, the client can now provide a secure data transfer for any project whether they can afford to use SSL certificates (and therefore HTTPS) or not.

Initially we had thought to exploit the fact that data is encrypted on the phone, and send it as it was, to avoid further encryption and decryption operations. However, that could not be done without significant changes to the existing client code, and without exposing many cryptographic operations to the developers. Not to mention that the same key used for the storage would be re-used for transmission, raising security concerns and key management issues. Hence, even if this could give better performance, it could also affect the security of the API and its usability. We chose, therefore, to simply wrap the data sent from the client in a secure connection, which, despite some extra traffic, allows also for a complete decoupling between the secure layer and the client requests.

Notice also the second parameter of the `SecureHttpConnection` constructor: `SecureHttpConnection.RequestType`. When this parameter is specified, our API can automatically generate some predefined requests that can be used for various operations: user registration; password recovery and server authentication as described in [5].

2.4 Secure Storage

The storage has been designed to accommodate typical scenarios in mobile data collection. In particular that multiple users should be allowed to use the same mobile device, that the same user can use multiple mobile devices and that Internet access might always not be 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 into the following concerns:

1. A mobile device must store some identification token to authenticate users off-line.
2. If a user loses the password, other users on the same device 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 the encrypted data stored on the mobile phone by some authorized entity.

A scheme that satisfies all the above requirements is described in [5] and implemented in the API, which offers tools to register a new user so that a

new personal secure storage is initialized according to such scheme. The client application simply needs to pass username and password to our `registerUser()` method, and thereafter use our `login()` method to get access to a user's data. The login method authenticates the user and creates a session object with a user's key, that the API will use to handle the secure stores and secure HTTP sessions. This is also independent from the authentication method used on the server. The data is encrypted with symmetric encryption, and the encryption key is protected by a password-based key. This means that losing the password does not prevent access to the data if the data encryption key has been saved, for example, on the server. Notice, however, that the overall security of the data still depends on the strength of the password, and as long as off-line local authentication is required on the mobile phone, and smart cards are not supported by the phone, this is a problem that cannot be solved. When it comes to the actual storage of data in the RMS analogue to the `SecureHttpConnection` class, we offer a `SecureRecordStore` class, that can be used to wrap the data in a secure way. Every write/read operation will, respectively, encrypt the input data before writing it in the actual RecordStore object, and decrypt it before returning it to the client. The API also takes care of checking whether the current user has permissions to write in that storage and handles the corresponding keys. All of this happens completely transparent way for the client.

The drawback of this approach is that the user has no control over the data encryption, so, every time something is read or written from the secure store, a cryptographic operation is performed. This can be a computational overhead if a search must be done across the stored data, since several decryption operations are required. This happens, for instance, when a menu must be generated to show the users which form values have been saved in the record store. To mitigate this problem, we offer to store the data with a label that describes it. All the labels are stored as a list in a single encrypted record, so that only this list needs to be decrypted to generate a menu, rather than all the records.

One advantage, instead, is that the client is not forced to pre-process the data and store it in a specific format or in a dedicated record store. The only assumption we make, is that each user has a dedicated record store, so that a unique key can be assigned to it. This makes the key and permission management much easier for the API.

An alternative solution we tested was to offer methods that took a byte stream and returned an object containing the encrypted stream plus a set of fields to manipulate it, so that the developer could have direct control over the encrypted data. However this idea was discarded because it would have required substantial refactoring in the existing client, and it could have potentially introduced security issues if the data were manipulated incorrectly.

2.5 Modularity of the API

While designing the API we focused also on making it modular. We tried to make the different packages that constitutes the API as independent as possible, so

that a client using only the secure communication, would not import the secure storage libraries and vice-versa, thus minimizing the final size of the application.

2.6 API integration

Figure 1 shows how our API creates a secure layer on top of the existing application layer, taking as example our work with the openXdata client.

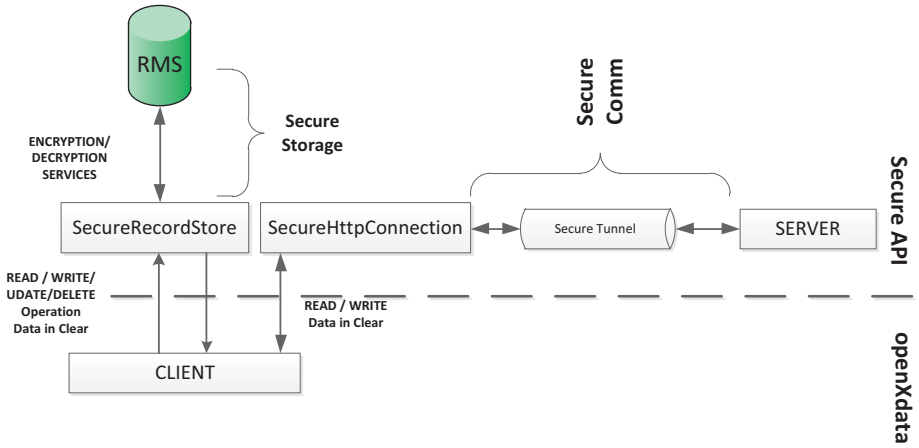

Fig. 1. openXdata - secure API integration architecture.

3 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 phones used for the benchmark are phones 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 into perspective, since they are not likely to be used due to their high cost. It is clear that with the given parameters the performance of the API is barely acceptable on the least powerful phone (2760), but it already has a more than acceptable performance compared to an equally cheap and only slightly more powerful device (2330c). It is interesting 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 of the protocol in [5] on the slowest phone. It is also clear that the bottle neck in the various transactions is the RSA encryption, 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, and, in general, it is not recommended to use less than 1024 bit anyway.

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	6	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)	28	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.

4 Related work and Conclusions

In general, all 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, 10], and we target low-end phones, so that solutions that might be adequate for high-end phone like smart phones, are not an option for our context.

The solution we implemented is based on a custom protocol developed by considering the specific constraints of MDCS [5], but it makes almost no assumptions about how or where data are 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.

Other approaches to secure applications having the Java ME platform as their target have been proposed in the literature [8, 3, 9], but it is easy to see that they are all tailored for specific target applications, and they are nowhere as extensive and flexible as our API. Besides, as far as we know, the solutions proposed in these works have not been employed in actual systems, while our API is currently being used to develop a secure client for openXdata, that next year will be deployed in the field and carefully tested in a real project. The results from this test will be used to optimize the code and develop new features for the API. For example, mechanisms for automatically configuring the security settings on each device, in order to maximize security without compromising usability, are currently being studied.

References

1. S. Crocker and J. Schiller. RFC 4086 - randomness requirements for security. http://www.ietf.org/rfc/rfc4086.txt, 2005.
2. T. Egeberg. Storage of sensitive data in a Java enabled cell phone. Master's thesis, Høgskolen i Gjøvik, 2006.
3. 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.
4. T. Legion Of the Bouncy Castle. http://www.bouncycastle.org/. Online, Accessed Mars 2011.
5. F. Mancini, K. Mughal, S. Gejibo, and J. Klungsoyr. Adding security to mobile data collection. In *Proceedings of Healthcom 2011 - 13th IEEE International Conference on e-Health Networking Applications and Services*, pages 86 –89, june 2011.
6. openXdata. http://www.openxdata.org. Online, Accessed Mars 2011.
7. Oracle. Java ME. http://www.oracle.com/technetwork/java/javame/index.html. Online, Accessed Mars 2011.
8. S. M. A. Shah, N. Gul, H. F. Ahmad, and R. Bahsoon. Secure storage and communication in J2ME based lightweight multi-agent systems. In *Proceedings of KES-AMSTA'08 - the 2nd KES International conference on Agent and multi-agent systems: technologies and applications, Incheon, Korea*, pages 887–896. Springer-Verlag.
9. Z. Wang, Z. Guo, and Y. Wang. Security research on j2me-based mobile payment. *IEEE Communication Society*, 2(2):644–648, 2008.
10. 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.