STUDY OF SECURITY MECHANISMS FOR SECURE MOBILE COMMERCE ARCHITECTURE THROUGH THE DEVELOPMENT OF A MOBILE BANKING APPLICATION.

A Dissertation
Submitted In Partial Fulfilment Of
The Requirements For The Degree Of

MASTER OF SCIENCE

In

NETWORK AND E-BUSINESS CENTERED COMPUTING,

in the

ESCUELA POLITÉCNICA SUPERIOR
UNIVERSIDAD CARLOS III DE MADRID

By
Paras Mehta

26th of March, 2008

Supervisor: Prof. Mario Muñoz Organero

Acknowledgements

I would like to express my sincere gratitude to my dissertation supervisor, Prof. Mario Muñoz Organero who was always patient in listening to my problems, giving me advice and supplying me with new ideas which were vital for the project work. I am indebted to him for his valuable guidance, encouragement and support throughout the dissertation period.

Special thanks to all the coordinators, teachers and assistants of the participating institutions of MSc in NeBCC – University of Reading, Aristotle University of Thessaloniki and Universidad Carlos III de Madrid for their contributions in the completion of this dissertation. I am also very grateful to the European Commission and the Erasmus Mundus programme for giving me the opportunity to study in this excellent course. And finally, I thank my colleagues for their constant support and motivation.

Abstract

Mobile commerce is a relatively new and evolving concept with immense potential and promise. An important and security-critical application of m-commerce is mobile banking. Despite the recent advancements, scope for improvement in this domain particularly on the security front still exists. This project is an endeavour to propose a feasible architecture ideally suited to m-commerce applications, with a focus on mobile banking. By studying and analyzing the existing m-commerce architectures, the major security threats are identified and security measures are devised based on which a new improved architecture is proposed which guarantees higher security and usability. Furthermore, to demonstrate its practical use, the proposed architecture is implemented through the design of an m-banking application.

Contents

1. Introduction

Mobile commerce, in simple terms, refers to the exchange of goods and services using handheld devices such as mobile phones and PDAs. It can be compared to e-commerce with the principal difference being the use of a mobile client. M-commerce is slowly and steadily gaining on popularity due to many reasons - the increasing ubiquity of mobile devices and the anywhere, anytime connectivity offered by mobile clients being a few of them. Mobile banking is an important application of mobile commerce and involves conducting banking transactions through a mobile client. This may include account balance and statement enquiries, fund transfers, paying bills, etc. The sensitivity of the data transferred is evident and hence, is the need for security.

The project aims at devising a software architecture for m-commerce applications which is both efficient and secure. It aims to identify the main security threats to m-commerce applications and to devise security measures to counter such threats. Hence, this involves analysis of the technologies and the architectures of the m-commerce applications currently available and functioning. These current architectures may have some security loopholes and there also may be a scope for improvement in terms of usability, interoperability or efficiency.

This project focuses on m-banking which is one of the most important applications of m-commerce. Security assumes very critical importance as the data exchanged can be quite sensitive i.e., financial data. Moreover, several technologies are available for data exchange between a mobile client and a bank server. Therefore the aim of the project would be to evaluate the most suitable of these technologies to ensure highest efficiency and security. The following section elaborates the major goals of the project.

1.1. Goals and Milestones of the Project

- Study and analysis of existing m-commerce architectures focusing on m-banking and identifying the major security threats and limitations.
- Devising security measures to address these threats and limitations and, proposing an improved secure architecture for m-commerce keeping security, efficiency and usability in mind as the deciding factors.
- Creating and deploying an application to demonstrate the feasibility and the functionality of the architecture.

1.2. Information Analysis and Research

The first milestone of the project is the study of existing m-commerce applications, finding the security threats and the tools and technologies that would be useful in this project. A summary of this background study and analysis is presented here:

1. Several leading banks such as Citibank, Wachovia, Bank of America offer mobile banking services to their clients. Hence, to study the existing m-banking applications,

these applications were analysed by gathering information from the banks' websites and the internet. For instance:

- Citibank offers its clients an application called Citi Mobile. The application can be downloaded onto mobile phones and PDAs and installed. Citi Mobile connects to the bank server and the client can view the details of his account on his cellular phone or PDA. It also has the option of making fund transfers and even to pay bills[1].
- Mobile banking service is also offered by the Bank of America to its customers. Wachovia offers Wachovia Mobile to its customers for m-banking[2]. Each of these m-banking solutions and applications offered by banks to their customers must guarantee security.
- Vipera Networks offers a mobile banking solution called MoBank which allows clients to carry out banking transactions using a smart client, SMS or WAP[3].
- .mobi is a top-level domain which is dedicated to the mobile web. With the .mobi sites, users are assured that the content of the web site is optimized to suit the constraints of mobile devices[4].
- Many other banks also provide applications for download by mobile clients or create web sites which are suitable for being viewed on the small screens of a handheld device.
- Information review about the current trends in mobile web.

2. Valuable research work is being conducted in universities and international bodies in the domains of J2ME applications, WLAN security and m-banking. This research work is also an important source of background information providing research papers and related material indicating the current trends in m-commerce security.

3. The mobile client in this project is developed using Java ME which consists of Java APIs adapted suitably to function on the limited resource environment of a handheld device such as a mobile phone. The project requires knowledge and study of J2ME and the API functionality offered by it. Java ME comes with a Wireless Toolkit (WTK) which can be used to build and simulate MIDlets before actually deploying them on the real device. In this project the J2ME WTK 2.5.2 has been used.

4. The main aim of the project is the development of a secure architecture. Hence, it is necessary to study basic cryptographic concepts and security mechanisms and requirements. This includes digital signatures, digital certificates, encryption, XML signature, etc. HTTPS is commonly used in online as well as mobile banking applications. Therefore, an understanding of the working and the functionality of HTTPS is necessary. Also, it is important to learn how to use the J2SE keytool utility or openssl which are used for managing certificates and keys and, the MEKeytool which is the version of keytool for Java ME.

5. Knowledge of XML and web services is also important which includes understanding of SOAP and WSDL and of types of web services – RPC and Document type, Encoded and Literal style and web service design with Apache Axis2 libraries. Additionally study of concepts of web services with Java ME and security in web services are essential.

6. Use of an IDE like Eclipse can facilitate programming immensely. Hence, for all the programming in Java, Eclipse IDE is used in this project. Apache Tomcat 5.5 is used as the web server. For web service development Axis2 libraries are used. Eclipse is

configured for web service development and deployment with Apache Tomcat and Axis2 and plug-ins are installed for Java ME Wireless Tool Kit.

7. The means of communication that can be used in m-commerce include SMS, WAP, HTTP, HTTPS or SOAP and XML. The actual data connection can be through GSM, GPRS, WLAN or 3G. Some basic knowledge is required about these subjects so that they can be analysed and compared.

1.3. Security Requirements of M-Commerce

After an in-depth analysis and study of the existing m-commerce and m-banking architectures, some major requirements and security threats and limitations of m-commerce solutions were found and are enlisted in the following sections.

1. **Confidentiality**: Confidentiality means that the data being exchanged by two parties can only be interpreted by the intended recipient. Confidentiality is ensured by encrypting the data being sent. Encryption itself can be of two types: symmetric and asymmetric.

2. **Integrity**: Integrity ensures that the data sent from one end is received intact without any modification at the other end i.e., the data must not be tampered with during transmission. Integrity is achieved using digital signatures and message digests.

3. **Non-repudiation**: Non-repudiation ensures the accountability of the two parties during a transaction or data transfer. The sender should not be able to deny having sent the data and the receiver should not be able to deny the receipt of data. Non-repudiation is ensured using digital signatures.

4. **Authentication**: Authentication ascertains that the other end of a transaction is what he claims to be. Authentication can be achieved by means of security tokens such as passwords, digital certificates, etc.

1.4. Security Threats and Limitations

The discovery of the chief security threats to m-commerce applications is a significant goal of the project and lays the foundations for the work in the remainder of the project. As mentioned earlier, based on an in-depth analysis and study of m-commerce technologies and architectures, some major security threats and limitations of m-commerce solutions were located and are explained below.

1. Client authentication is not offered normally by HTTPS applications.

Use of Passwords

This is the most common method for authenticating a user. The client is asked for a username and password to log in.

- This is not secure as the password can be hacked easily through brute force.

- To counter this, usually, banks can impose a limit on the number of login attempts after which the account of the client is blocked. But, placing a limit on the no. of login attempts makes blocking an account easy for a miscreant.
- People tend to write passwords down in places from where they can be easily stolen or choose passwords which can be guessed easily.

Ownership Differences between Web Apps and Mobile Apps

- Web applications are designed for access from any machine, any browser, anywhere in the world. To prove my identity to the web application, I need to supply a login name and a password through the web browser of whichever machine I am using anywhere in the world to authenticate me.
- But, mobile devices are typically owned by one person and people carry them everywhere.

Storing Secret Information on the Device

Because each MIDP device usually belongs to one person, user names and passwords can be stored on the device so the user doesn't have to enter them each time he wants to use an application.

- This significantly increases convenience, especially because it is hard to enter data on a typical MIDP device.
- However, MIDP doesn't support secure storage, so theft of the device could be a disaster for the user.
- J2ME contains the SATSA API which offers a nascent optional package for access to secure storage, among other things.
- Instead of allowing the user to choose a password, a MIDP application can embed a password in the client application and send it with requests to the server automatically.
- Storing password on the device offers an additional benefit. The password no longer needs to be a human-readable string. It can be a random array of binary data, which makes a dictionary attack virtually impossible.

2. HTTPS is not sufficient for non-repudiation. Non–repudiation can be provided by using digital signatures, timestamps and logging the sending and receipt of messages.

3. HTTPS offers Point-to-point security only. SSL secures direct connections between hosts. However, web services, the emerging topography of commerce on the internet will require multiple intermediaries to help process and deliver XML-based service requests. Hence, an end-to-end security solution is needed.

4. Peer groups and multicast applications will be a major model for the smart wireless applications of the future. Being a one-to-one protocol, SSL does not support multicast applications very well.

5. SSL indiscriminately encrypts all data with the same key strength, which can be unnecessary or even undesirable for some applications. The computational overhead for setting up and running SSL connections is simply too high for some wireless applications.

Having identified the threats to the security of data and the limitations of the existing architectures in m-commerce, the next step is to devise security mechanisms to address these problems and to propose an improved and more secure architecture.

2. Preliminary System Design

2.1. Secure M-Commerce Architecture

In this section, architecture for m-commerce, particularly m-banking, is proposed which overcomes most of the security issues identified earlier. The features and functionality of this architecture are explained in the figure and the step-wise explanation that follows:

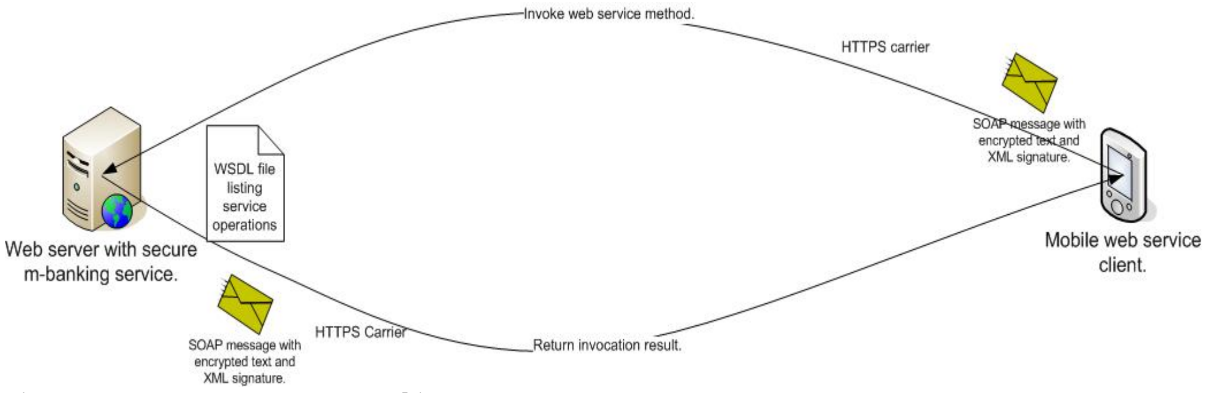

Invoke web service method.
HTTPS carrier
SOAP message with encrypted text and XML signature.
WSDL file listing service operations
Web server with secure m-banking service.
Mobile web service client.
SOAP message with encrypted text and XML signature.
HTTPS Carrier
Return invocation result.

Figure 1. Secure m-Commerce Architecture

1. **Web Services and XML**: Web services and XML are technologies of the future. They promise standardization, interoperability and facilitate transactions particularly, B2B transactions. In this architecture, communication takes place using web services and XML. This would also allow the exchange of data between different banks and between the bank and other entities to be automated by designing standard XML formats for messages. Hence, a bank server is developed and deployed as a web service so that it can be accessed by a banking client using web services. The service would publish the operations that the client can carry out using the WSDL file. The client can connect to the web service and invoke these operations. Communication between the client and the server takes place through the exchange of SOAP messages. Web Services was chosen for this project after a careful analysis and comparison with other communication technologies. Here are some of the merits and demerits of Web Services and XML with respect to other options for m-commerce.

2. **MIDP Web Service Client**: To access the bank server, a client has to be designed which can run on a handheld device. The client should be able to connect to the server and carry out banking transactions. As the bank provides a banking web service, we need to design a web service client using Java ME with an interface for the user showing all the possible transactions that he can carry out. Depending on the choice of the user an operation is invoked on the web service, the results are retrieved and are displayed to the user. In the event of the occurrence of an error, an appropriate message is displayed to the user. The client should be equipped with security mechanisms to ensure that communication is secure.

3. **Security at Application Layer**: To design a secure architecture, security at the transport layer is not sufficient. Security measures have to be added in the application also e.g., XML signature and digital encryption. This implements an additional security tier controlled by the client and the server applications in addition to the security that may be provided at the lower layers for example, through SSL. An XML signature block is appended to the body of every SOAP message to provide authentication, non-repudiation and message integrity. The actual message inside the body is not sent in clear text but, is encrypted. This provides confidentiality.

   For client authentication, the client can store its key on the device. The key itself should be encrypted with a secret known only to the client. The user therefore needs to know this secret to use the client's key for signing messages

4. **SSL**: For ensuring confidentiality of data, SSL can be used as till date it has proved to be an effective protocol for confidentiality. SOAP messages carrying the XML signature can be sent over a secure HTTPS connection to ensure security in place of an HTTP connection which is normally used. SSL is used at the session layer and its use is transparent to the upper layers. This adds to the security implemented at the application level and makes the application more secure.

5. **Standard XML Message Formats**: Another important feature of architecture is the standardization of the content and the format of the XML messages exchanged. This would facilitate interoperability immensely. The XML messages sent and their responses for a particular operation must carry data in a standard format. For example for to transfer money from the user's account to another account, we can standardise the XML document format to include the user's account number, the destination account holder's name, the destination account number and the amount transferred. This facilitates interoperability as in different application scenarios, the fields within the XML document remain the same and only the data within these fields may change.

3. Implementation Strategy

In the last section, the features of the secure architecture for mobile commerce proposed in this project were outlined. The implementation of the architecture was carried out in a sequence of phases which are explained below:

1. **Security Analyser Application:** As discussed earlier, study of the current m-commerce architectures, their key security threats and limitations and the identification of suitable security techniques and mechanisms is an important part of the project. Hence, in this direction, a 'Security Analyser' application was designed which modelled a simple client authentication application with username and password. This helped analyze the security of existing applications and to gain an insight into the actual implementation and functioning of HTTPS based security mechanisms.

   The application consists of a servlet running on a web server which asks for a username and password from every client that sends a connection request. A J2ME MIDlet was developed which connects to the servlet through an HTTPS connection and provides the user with an interface to enter the username and password. Next, the input from the client is sent to the server which connects to a database and verifies them against the values stored in the database. Lastly, it notifies the client about the result of the verification and the client displays the results on the screen to the user. Additionally, another MIDlet was designed which connects to the server and launches a Brute Force attack to recover the password. This helped understand the functioning of HTTPS which is used in several present day m-commerce applications and gave an idea about the merits and demerits of the protocol.

2. **Starting With Java ME Web Services:** In the secure architecture for m-commerce proposed in this project, Web Services is employed. Hence, in this phase of implementation, a basic web service was designed to get an idea of web service development. Subsequently, a mobile client was designed which can connect to this web service and invoke the operations published by it.

   For the development of the application, Eclipse was used. Eclipse has tools for developing and deploying web services. Java ME comes with the J2ME web services API (JSR 172) which provides functionality for developing MIDlets which can connect to web services. Furthermore, the Java ME Wireless Tool Kit is equipped with a Stub Generator utility which can generate the stubs of a web service using its WSDL file. The SOAP messages exchanged between the server and client were captured and analysed using the Network Monitor utility of J2ME WTK.

   This basic web service-client application models the basic communication model of the proposed secure architecture. It gives an insight into web service application development which would be used in the later phases of the project implementation.

3. **Securing J2ME web services:** Till the previous step, a basic web service and client application was designed wherein the server and the client exchange SOAP messages. However, these SOAP messages carry XML data as clear text which can easily be intercepted, read and modified by an eavesdropper. Hence, there is a need to secure this

communication. This again would give an idea regarding how to implement the security mechanisms such as XML signature and digital encryption which were proposed as a part of the secure m-commerce architecture.

To introduce application level security, XML signature and digital encryption will be used within the SOAP messages. XML signature provides syntax for digital signature for XML documents. Besides the signature, it carries several other pieces of information in the form of XML including message digest value, the algorithms used for calculating the digest and the signature, the transformations which were carried out on the data before calculating the signature, the part of the document that was signed, etc. XML signature provides authentication and non-repudiation along with message integrity which is provided due to the incorporation of message digest value in the XML signature. To ensure application level confidentiality, the data will be encrypted using a symmetric key algorithm. The key is shared only by the client and the server between which the communication is taking place. Therefore, in this implementation phase, the basic web service developed earlier was enhanced with the addition of XML signature and digital encryption to ensure security.

Another level of security is added to the basic enhanced web service application through the use of SSL. The service and the client were configured to use HTTPS instead of HTTP as the carrier of their SOAP messages. Use of SSL provided security at the lower layers. However, the use of HTTPS in place of HTTP is transparent to the application.

To summarise, all this gave an idea of how to practically implement the security techniques that form a part of the secure architecture proposed in this project. This made it easier to use these security mechanisms later in the m-banking application developed at the end of the project.

4. **Standardizing XML message formats:** To facilitate interoperability and exchange of data, the format of the XML data exchanged through the SOAP messages should be standardized depending on the operation invoked by the client. For every operation there should be the same fields of information exchanged but, the value contained in these XML fields may vary. Hence, first some operations that banks provide to it clients were selected e.g., enquiry of the account balance, fund transfer, statement of the last transactions in the account. Subsequently, for each of these operations the structure of the XML request message which the client would send to the server and response message which the server would send to the client were formulated. These standard message formats were later also used in the m-banking application developed to implement this secure architecture.

5. **Developing an m-banking application:** Finally, after having proposed a secure architecture for m-commerce and having acquired all the necessary knowledge and understanding of the tools and technologies that form a part of the architecture, a basic m-banking application was designed to demonstrate its actual functioning and viability.

   The secure m-banking application consists of a banking web service which offers its clients three operations: enquiry of the account balance, transfer of money to another account, display of the last ten transactions in the account. The formats for the messages used for carrying out these transactions are based on the standard XML message formats formulated in the last implementation phase. A mobile web service client was developed which provides an interface to the user to carry out transactions on his account in the bank. It can connect to the bank server and invoke an operation depending on the choice of the user by sending the necessary input data along with the security information such as signatures and ciphers. After that, it retrieves the response, decrypts it, verifies the signature and displays the results. If an error occurs or the decryption and signature verification fails, error messages are displayed to the user.

4. Detailed Software Design

The design of the software developed in this project is detailed here:

4.1. Security Analyser application

The security analyzer application is designed to study and model the username/password authentication mechanism that is used in several applications. The server and the client side of the application have been explained separately below.

4.1.1. HitServlet.java

- To replicate a common banking application, a servlet was deployed on a web server which asks for a username and a four digit numeric password upon receiving a client request. The web server used for this application was Apache Tomcat 5.5. From the request from the client, the username and password are extracted using the request.getParameter() method in the doGet() and doPost() methods of the servlet.

```
String username = request.getParameter(“uname”);
```

- The correct username and password values are stored in MySQL database. Hence, the values received from the client are verified against the values stored in the database. Depending on whether the verification is successful or not, the client is notified. The client can read the response from the server and display a message accordingly on the phone to the user.

4.1.2. Connection to MySQL

- The connection to the database from the servlet is done using the Connector/J JDBC driver. To use the driver, it has to be first loaded using Class.forName() method:

```
Class.forName("com.mysql.jdbc.Driver");
```

- Next, a connection with the database is established.

```
Connection conn = DriverManager.getConnection(url, username, password);
```

- Finally, a Statement object is created and using the executeQuery() method, an SQL query is executed on the database.

```
Statement stmt = conn.createStatement();
ResultSet rs = stmt.executeQuery(sqlquery);
```

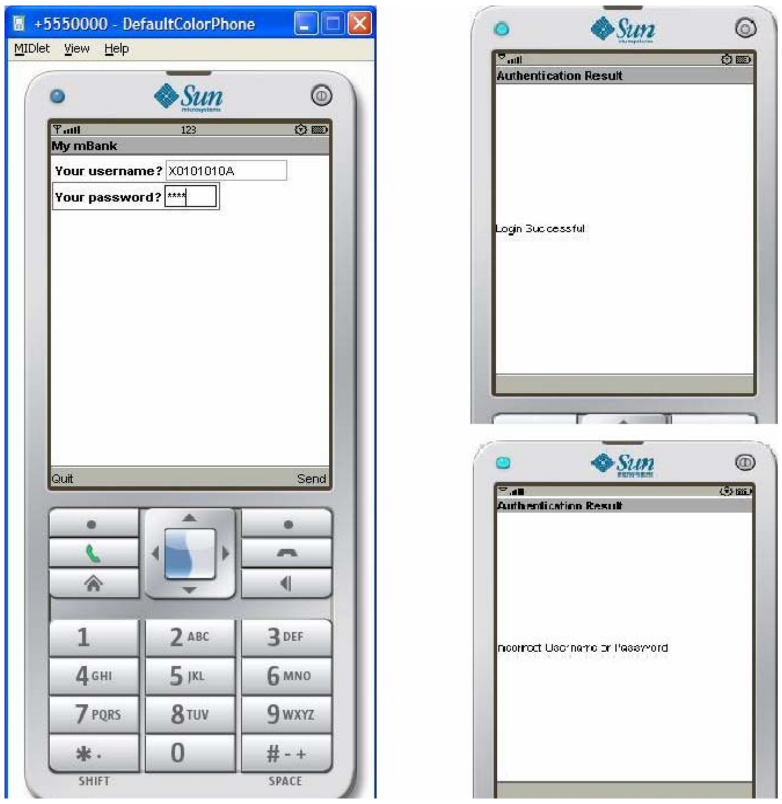
As most present banking applications use SSL or TLS, in this application also, the client connects to the server using HTTPS. First, the server was configured to enable use of HTTPS and to accept HTTPS connections.

4.1.3. Configuring Tomcat 5.5 for SSL

- A keystore was created for the server using Java keytool utility and an RSA key pair was created for the server and stored in this keystore.
- A self-signed certificate for the public key of the key pair was exported to a file. The certificate was imported to the list of trusted certificates for the mobile client using the Manage Certificates utility of the Java ME WTK which maintains a list of all the Certificate Authorities trusted by Java ME MIDlets.
- Finally, the Tomcat server.xml file was modified to accept HTTPS connections on port number 8443 and to use HTTPS and, to use the keystore and the certificate generated earlier in the SSL handshake.
- During the SSL handshake, the server sends this certificate to the client. The client accepts and verifies it as it is already in its list of trusted certificates. The data exchanged between the server and the client is encrypted and the communication is secure in HTTPS.

From the client side, the username and password parameters could be sent to the server through two HTTP request methods: GET and POST. For the client application, two MIDlets which had two different functions were designed – HitMIDlet.java and TestMIDlet.java.

4.1.4. HitMIDlet.java



My mBank

Your username? X0101010A

Your password? ****

Authentication Result

Login Successful

Authentication Result

Incorrect Username or Password

Figure 2. HitMIDlet user interface

As shown in figure 2, HitMIDlet displays a form to the user on the phone with text fields for the username and password. The values entered in the text fields are read and an HTTPS connection is established with the server. To do this, the MIDlet uses the HttpsConnection class of the javax.microedition.io package of Java ME which carries the functionality of an HTTPS connection and allows a MIDlet to communicate using HTTPS. The creation of an HttpsConnection object is shown in the following code snippet.

```
hc = (HttpsConnection) Connector.open(url);
```

where 'url' contains the address of the server e.g., 'https://localhost:8443/HTTPSTestServer/HitServlet'. The username and password are sent inside an HTTP POST request. This is done in the following way:

```
hc.setRequestMethod(HttpsConnection.POST);
hc.setRequestProperty("Content-Type", "application/x-www-form-urlencoded");
```

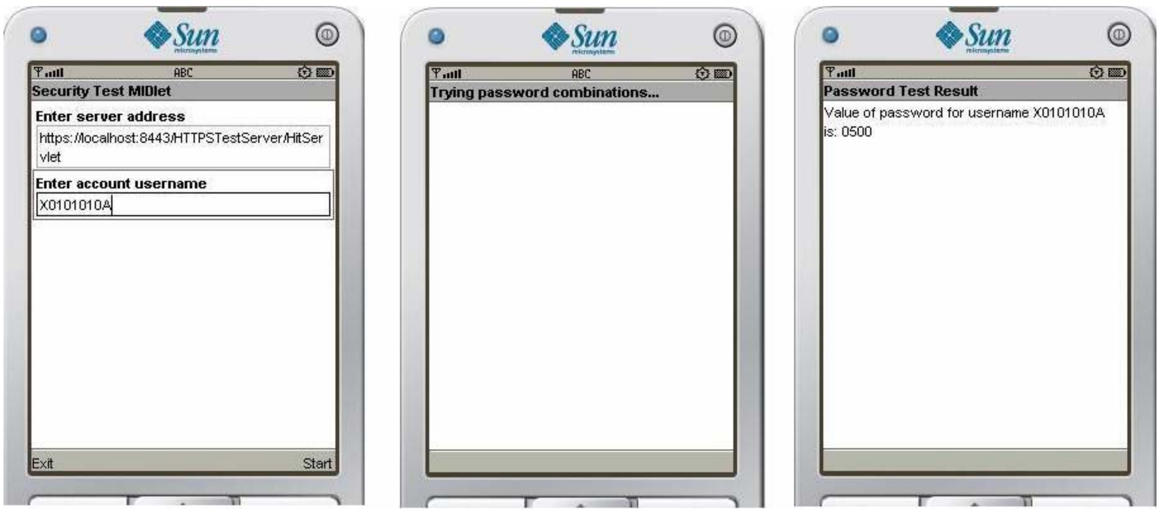
The first Java statement sets the request method to HTTP. In the POST request, the username and password are written directly into the output stream and form a part of the request body. The second statement sets the content type to url-encoded so that at the server side, the parameters can be extracted from the request using the getParameter() mehod even though they are explicitly not a part of the URL.

The server connects to the database and checks whether the given username-password combination is valid and notifies the client accordingly. If the verification succeeds, a message 'Login Successful' is shown else, the message 'Incorrect Username/Password' is displayed. Pictures of both these messages as simulated on the Sun WTK are shown in the figure above.

4.1.5. TestMIDlet.java

TestMIDlet also connects to the server using HTTPS as explained above but, by sending a GET request. The GET method is the default request method and in this case, the username and password are appended to the server URL as URL parameters. The server extracts the parameters from the request using request.getParameter() statement.

The aim of the TestMIDlet is to retrieve the password of the account knowing the username using brute force. The MIDlet tries all combinations from 0000 to 9999 opening an HTTPS connection for every try and incrementing the value of password in the URL. When the correct value of the password is found, the trials stop and the correct value is displayed to the user. Hence, in the figure below, the TestMIDlet asks the user for the username whose password is required and the URL of the server. After pressing 'Start' the password trials start and when the password that matches is found, it is displayed along with its username on the screen.



Security Test MIDlet
Enter server address
https://localhost:8443/HTTPSTestServer/HitServlet
Enter account username
X0101010A

Trying password combinations...

Password Test Result
Value of password for username X0101010A is: 0500

Figure 3. TestMIDlet user interface

4.1.6. Results and Observations

1. It was measured that it took approximately 10 seconds for the TestMIDlet to recover a password of value ‘0500’ trying all combinations from 0001 to 0500, knowing the username.
2. It shows that HTTPS alone may not be sufficient to guarantee security in m-banking applications. Hence, banks usually limit the number of login attempts to rule out a brute force attack.
3. However, this could be a potential problem. A miscreant knowing the username of a user could repeatedly get the account blocked by trying wrong password combinations. Therefore, security should not be at the cost of user convenience.
4. Although, HTTPS encrypts the parameters that are sent in the URL, it is a poor practice to include information in the URL as is done in a GET request. The URL may be stored in cleartext in the server logs or in the browser history making it quite vulnerable to attacks[5].

4.2. Using J2ME Web Services

Web services and XML are the communication technologies on which the secure m-commerce architecture would be based. This phase involves the study of J2ME web services and their use in a simple application. J2ME comes with an API for web services defined by the JSR 172 specification.

4.2.1. What is Java ME Web Services API?

The J2ME web services API (JSR 172) extends the Java 2 Platform, Micro Edition to support web services. The API has two optional packages which provide the functionality necessary for mobile clients of web services: remote service invocation and XML parsing.

The remote invocation API is based on a strict subset of J2SE's Java API for XML-Based RPC, with some Remote Method Invocation (RMI) classes included to satisfy JAX-RPC dependencies[6]. The typical steps for creating a JSR 172 JAX-RPC client are:

1. Generate a JSR 172 JAX-RPC stub class from a WSDL XML file that describes a remote web service. For this, the Sun WTK Stub Generator utility can be used.
2. In the client, create an instance of the generated stub.
3. After instantiation, invoke methods of service on the generated stub. These methods correspond to values under wsdl:operation tag in the WSDL file.

Using stubs makes remote service invocation very easy. The client instantiates the stub class, initializes the instance, and invokes one of its methods supplying input values, if any, as parameters which may return an object. The application may then use the different fields within the returned object to extract the required output.

JAXP and JAX-RPC are both optional packages; they can exist independently of each other. The JAXP XML-parsing API is based on a strict subset of the Simple API for XML, version 2 (SAX2). Even though most of the classes in the standard SAX2 API are missing in this subset, all the necessary core functionality remains.

After studying the functionality of the J2ME Web Services API, a basic web service application was designed and invoked using a mobile client. To start with, the WSDL file for the web service was designed in XML and using the Eclipse web services tools, the Java skeleton classes which contain the implementation logic of the service are generated. This is explained later. The service was deployed on the Tomcat web server. The J2ME Wireless Toolkit (WTK) comes with a stub generator utility which can be used to generate the stubs of a web service from its WSDL definition. So, the stub classes were generated by supplying the WSDL file location in the stub generator. Subsequently, a client MIDlet is developed to use these stub classes and invoke the web service. In the MIDlet, the stub class is instantiated and use to call the operations offered by the service in the WSDL file providing the relevant input parameters. The stub takes care of the communication with the web service. These steps are demonstrated in the code snippet below:

```
SecureMobileEmailService_PortType_Stub service = new
SecureMobileEmailService_PortType_Stub();

service._setProperty(SecureMobileEmailService_PortType_Stub.SESSION_MAINTAIN_PR
OPERTY, new Boolean(true));

service.getSubjects(msgString);
```

The MIDlet interacts with these stubs which in turn interact with the web service. The stubs does the job of converting the input and output parameters to the required format before sending them to the server.

As mentioned earlier, from the service WSDL file, the Java skeleton classes are produced on the server side. The skeleton plays a similar role for the server side as the stub does for the client side. The stub does not interact with the service directly but does it through the skeleton. In the skeleton the required business logic for the service is put for each operation offered by the service to its clients. It accepts the input parameters sent by the client in the method call, carries out the necessary operations and returns the desired output expected by the client. In this case, the generation of the WSDL, the deployment of the service and the generation of the skeleton classes is taken care of by the Eclipse IDE.

4.3. Securing J2ME Web Services

After development of the basic web service and gaining acquaintance with the concept of J2ME web services, comes the job of securing the architecture.

In web services, the client and the server exchange SOAP messages. In order to secure this communication, the security mechanisms proposed in the secure m-commerce architecture are as follows:

- Appending a Signature XML structure to the SOAP messages
- Encrypting the message content in the SOAP message at application level
- Use of HTTPS as carrier protocol for SOAP messages in place of HTTP to provide lower layer security.

Therefore in this phase of the software design, the simple web service is enhanced and the above security features are added and tested to understand their implementation and functioning.

4.3.1. What is XML Signature?

XML Signature is a W3C recommendation that defines XML syntax for digital signatures.
XML signatures can be used to sign data of any type, typically XML documents, but anything that is accessible via a URL can be signed.

An XML Signature consists of a Signature element in the http://www.w3.org/2000/09/xmldsig# namespace. The basic structure is as follows:

```
Signature
        SignedInfo
                SignatureMethod
                CanonicalizationMethod
                Reference
                        Transforms
                        DigestMethod
                        DigestValue
                        Reference ...

        SignatureValue
        KeyInfo

 Object
```

- The SignedInfo element specifies what was signed and using which algorithms. The SignatureMethod and CanonicalizationMethod elements are used by the SignatureValue element and are included in SignedInfo to protect them from tampering. A list of Reference elements specify which resources have been signed, using URI references; this element also specifies any transforms to apply to the resource before applying the hash, the digest algorithm (in DigestMethod), and the result of applying it to the resource (Base64-encoded in DigestValue).
- The SignatureValue is the Base64-encoded value of the signature. This value is the signature (produced according to the specification of the SignatureMethod element) of the SignedInfo element after serializing it with the algorithm specified by the CanonicalizationMethod element.
- The KeyInfo is an optional element that enables the recipients to obtain the key needed to validate the signature. Typically it can contain a set of X.509 certificates. If a KeyInfo element is not present, the recipient is expected to identify the key from context.
- The Object is an optional element used to hold the signed data in the case of an enveloping signature[7].

4.3.2. Customized XML Signature

In the project, the structure of the XML Signature block has been ‘customized’ to suit requirements and constraints. The XML Signature forms a part of the SOAP envelope and signature value is calculated by encrypting the hash of the message content with a symmetric key. It is then encoded to base-64 and placed inside the XML Signature.

The organization of various elements within this modified XML signature is depicted below:

```
Signature
        SignedInfo
                SignatureMethod
                CanonicalizationMethod
                Reference
                        DigestMethod
                        DigestValue
                        Reference
        SignatureValue
        KeyInfo
```

4.3.3. What is SATSA?

For the calculation of digest, ciphers and the signature values needed in this project, an API of Java ME called the Security and Trust Services API (SATSA) defined by the JSR 177 specification is used. SATSA is used for providing security services and communicating with a smart card security element such as a SIM card in J2ME.
It consists of four main packages:

i. SATSA-APDU Communication API
ii. SATSA-JCRMI Communication API
iii. SATSA-PKI Signature Service API

iv. SATSA-CRYPTO Cryptographic API

The first three APIs – SATSA APDU, SATSA JCRMI and SATSA PKI are for interaction with a java card like a SIM card in the device. The SATSA CRYPTO package provides security services such as message digests, encryption and signature verification and does not require presence of a java card. The SATSA-PKI package manages the functionality of the security services associated with Public Key Infrastructure like the generation of digital signature and the management of security credentials like digital certificates[8].

4.3.4. Encryption of Message Content

In the simple web service designed previously, the SOAP messages carried the message in the form of clear text. However, this message may contain very sensitive financial data like account numbers, credit card numbers, etc. Hence, to ensure security of the data within the SOAP message body irrespective of the security provisions at the lower layers, the message is encrypted with a secret shared key using a symmetric key algorithm such as DES or Triple DES and then placed within the SOAP message. Like this, only the two communicating parties which share the secret key can view the message content.

The process of encrypting and decrypting the message content is basically the same for the mobile client and the server. The class used for encryption and decryption is the Cipher class of javax.crypto package. The encryption process is carried out through the following sequence of steps:

- First, an object of the Cipher class is created and instantiated through the getInstance() method passing the encryption algorithm as a parameter. The parameter is of the form “algorithm/mode/padding”.

  ```
  Cipher cipher = Cipher.getInstance(algorithm);
  ```

  For instance, in our m-banking application, the value is "DES/ECB/PKCS5Padding" which signifies that the encryption algorithm being used is DES in the Electronic CodeBook (ECB) mode with the PKCS5 padding mechanism. The ECB mode is the simplest mode of encryption where the text is divided into blocks and each block is encrypted with the key separately. The PKCS5 padding mechanism is used for adding padding at the end of the text to make it a multiple of the block size. At the receiver side, these padding bytes have to be removed from the decrypted text to get the message.

- Next, a secret key is constructed using the SecretKeySpec class from a byte array containing the value of the shared key. The object returned is stored in an instance of the interface Key as the SecretKeySpec class implements the Key interface.

  ```
  Key key = new SecretKeySpec(keyBits, 0, keyBits.length, keyAlgorithm);
  ```

- The cipher is initialized with this key in the encryption mode using the init() method of Cipher class.

  ```
  cipher.init(Cipher.ENCRYPT_MODE, key);
  ```

- Lastly, the data is encrypted using the doFinal() method supplying the input clear text in a byte array and a byte array to store the output.

```
cipher.doFinal(plaintext, 0, plaintext.length, ciphertext, 0);
```

- This encrypted byte array output is base64 encoded and put place of the clear text message in the SOAP message. For this, a class called Base64Coder is used which contains two methods – encode and decode for base64 encoding and decoding respectively.

```
char [] base64Cipher = Base64Coder.encode(ciphertext);
String base64Str = new String(base64Cipher) ;
```

When the receiver gets the SOAP message, it extracts the encrypted message and follows a similar procedure to decrypt the data. The cipher is instantiated with the same algorithm and initialized using the same key that was used for encryption. But, in this case, the cipher is initialized in the decryption mode by replacing the Cipher.ENCRYPT_MODE with Cipher.DECRYPT_MODE as shown below:

```
cipher.init(Cipher.DECRYPT_MODE, key);
```

This initializes the cipher for decryption. The next step is the same using the doFinal() method except that in this case the input byte array contains the encrypted data while the output is the decrypted data. However, the process is still not complete because during encryption padding bytes were added to the plaintext before encryption. Hence, to retrieve the original message the decrypted data is processed to strip off the padding bytes.

4.3.5. Calculation of Message Digest

It is necessary to calculate the digest of the message in order to draft the XML signature in the SOAP messages. The digest value is calculated through the SHA-1 digest algorithm using the MessageDigest class of the SATSA-CRYPTO API. The calculation of the message digest from the text message is carried out in three steps and is explained below.

- First, an instance of the MessageDigest class is created, instantiating it for use with SHA-1 algorithm.

```
digest = MessageDigest.getInstance("SHA-1");
```

- Next, the input message is fed in the form of a byte array into the MessageDigest instance with the update() method.

```
digest.update(message, 0, message.length);
```

- Finally, using the digest() method, the digest is calculated and stored in a byte array called output.

```
digest.digest(output, 0, output.length);
```

- Before placing this message digest value in the XML signature, it is converted to the base64 format using Base64Coder.encode() as described before. The string obtained at the end called base64Digest is placed in the XML signature.

Upon receiving the message, the value of the message digest is obtained from the DigestValue tag inside the XML signature. The actual message content is obtained by decrypting the encrypted message inside the SOAP envelope. The digest of the decrypted message is calculated and matched against the digest value inside the DigestValue tag. This helps ensure message integrity.

4.3.6. Formulation of XML Signature

As explained previously, other than the actual digital signature value, the XML signature contains several other fields. Hence, the drafting of the XML signature is explained in two steps: the calculation of the digital signature to be placed inside the SignatureValue tag and, the filling of the rest of the XML fields in the XML signature.

- **Calculation of Signature Value:** The signature value is calculated by finding the digest of the message to be signed and then encrypting it using the symmetric key shared by the client and the server. The only difference is that instead of supplying plaintext as input to the cipher, digest value is supplied. Subsequently, the signature byte array obtained is base64 encoded and put inside the XML signature.

  The calculation of message digests and encryption of data uses the SATSA CRYPTO package. This approach also offers authentication, non-repudiation and message integrity as the server is assured that the client, and the client only, possesses the secret key and hence, the correct verification of the 'signature' ascertains the authenticity of the client, the integrity of the message and non-repudiation. This value of 'signature' calculated using a symmetric key is base64 encoded and put under the SignatureValue tag.

- **Completing the XML signature structure:** The XML signature carries other tags which carry information about the signature. Before sending the SOAP message containing this XML signature structure, we have to calculate the signature and fill in the required fields in the XML signature. The digest algorithm being used is SHA-1, so, we put the value http://www.w3.org/2000/09/xmldsig#sha1 under the DigestMethod tag and the value of the digest under DigestValue. In the URI tag under the Reference tag, we put an XPath expression pointing to the part of the document that was signed. As we are not using a cannonicalization algorithm, in the CanonicalizationMethod tag we put the URI for null canonicalization as mentioned in the XML Signature specification - http://www.w3.org/1999/10/signature-core/null. Similarly, in KeyInfo, we put an identifier for the secret key being used.

  To explain the security mechanisms used in this project, an XML signature part of a message between the client and the server in the m-banking application developed at the end of this project are produced below. The Java ME WTK comes with a Network Monitor utility which can be used to capture and view messages for debugging purposes.

```
<Signature>
        <SignedInfo>
```

```
                <CanonicalizationMethod>http://www.w3.org/1999/10/signature-
                core/null</CanonicalizationMethod>
                <SignatureMethod>des-sha1</SignatureMethod>
                <Reference>
                   <DigestMethod>http://www.w3.org/2000/09/xmldsig#sha1</DigestMethod>
                   <DigestValue>VBbIWTGpe4ums67FXiPbrWmB9v4=</DigestValue>
                   <URI>//accountNumber/account</URI>
                </Reference>
        </SignedInfo>
        <SignatureValue>HV7dDro2+HojvgjeWxGEIIOKdeG14DBJ</SignatureValue>
        <KeyInfo>
                <KeyName>paras</KeyName>
        </KeyInfo>
</Signature>
```

To verify the XML signature at the receiving end, the information inside the XML tags is extracted. Next, the digest is calculated from the decrypted message content inside the SOAP envelope. Subsequently, the calculated digest value is matched with the digest value inside the XML signature and, the values extracted from all the tags with their expected values except the SignatureValue tag which is verified in the end. If all of them verify to be true, signature value is calculated by encrypting the calculated digest with the shared key and matched with the signature value in the XML signature. Only if this condition is also satisfied, the XML signature verification is deemed successful.

4.3.7. SOAP Over SSL

To provide lower layer security to the communication in our architecture, instead of using HTTP to carry SOAP messages, HTTPS was used. HTTPS involves the use of Transport Layer Security (TLS) protocol which is similar to the Secure Sockets Layer (SSL) protocol to provide security. The switch from HTTP to HTTPS does not call for any alteration in the application logic. HTTPS also offers confidentiality and server authentication through the encryption of data being sent over the network and the use of server certificate but, provides point to point security at the transport level. In the application, we are encrypting and signing the data which provides end to end security. The use of SSL makes the architecture more secure without any need to change the application logic.

The procedure to configure the web server for SSL has been described earlier and is summarized here. To use HTTPS, a public-private key pair along with a self- signed certificate has to be generated for the server. Additionally, the server’s configuration file has to be modified to allow the server to use HTTPS by providing the name and location of the keystore containing the key pair and, to accept HTTPS connections on a specified port.

In the client stub, the endpoint address of the service should be changed from the format http://address:newportno/... to https://address:portno/... and the port no. should be changed to the port on which the server is listening for HTTPS connections. Hence, no change is needed to the Java code for the client or the server for moving from HTTP to HTTPS. Therefore, the use of SSL for communication in the application is optional. The client can access the service using HTTP or HTTPS. This may be desirable for devices with low processing power and bandwidth.

4.4. Standardizing XML Message Formats

After understanding the security measures to be employed in the architecture, the next step is to define the templates for the XML request and response messages for some common operations used in banking applications. Three banking operations are chosen here:

1. Balance Enquiry: retrieves the account balance.
2. Transfer Money: transfer of money to another account in the bank.
3. Transaction History: retrieves the last ten transactions in the account.

Every operation requires two XML documents to be exchanged – one for the request from the client to the server and the other for the response from the server to the client. Below are displayed and detailed the XML document templates for each of these operations.

1. **Balance Enquiry**
   Request message contains **account number** for which the balance is being enquired and **XML Signature**.
   Response from server contains **account balance** and **XML Signature**.

   To give an idea, here is simple example of what the request and response messages for balance enquiry should look like:

   ```
   <BalanceEnquiry>
       <Account>39483948</Account>
       <Signature>
               <SignedInfo>…
       </Signature>
   </BalanceEnquiry>

   <BalanceEnquiryResponse>
       <Balance>500.00</Balance>
       <Signature>
               <SignedInfo>…
       <Signature>
   <BalanceEnquiryResponse>
   ```

2. **Transfer Money**
   Request contains the **sender's account number**, the name of the **destination account holder**, the **destination account number**, the **sort code of the destination account**, the **amount** to be transferred and **XML Signature**.
   Response contains a boolean **status** value signifying whether the transfer was successful or not and **XML signature**.

3. **Transaction History**
   Request contains **account number** for which the transaction history is asked and **XML signature**.
   On the server side, an XML type called **transaction** is created which contains various fields modelling a single transaction: **date** of the transaction, **title** of the transaction, **nature** of the transaction and the **amount** of the transaction. Hence, response contains a sequence of ten transaction objects and XML signature.

4.5. Developing the Secure M-Banking Application

In this final stage of the project, all the work done in the previous phases is put together to create a secure application for m-banking.

4.5.1. Implementation of Server

The bank server has to be developed and deployed as a web service. A bottom-up web service was created starting from the WSDL definition of the service. In the WSDL file of the service, user-defined types and elements have to be defined like the classes in Java to encapsulate the information necessary in the request and response messages of each operation. When the WSDL file is deployed and the Java skeleton classes are generated, each of these complex types corresponds to a separate Java bean class file with getter and setter methods for their inner fields or data members.

The WSDL file of the banking web service contains information about the service including the names and definitions of the data types, elements, messages, port types, binding, location, etc.

The essential steps in the functioning of the secure m-banking service are:

1. The service offers three operations to its clients. It receives a request from the client to invoke one of these operations.
2. The information including the encrypted message content and the XML signature is extracted from the request. The message is decrypted and the signature is verified.
3. In the event of decryption failure or signature mismatch, the client is notified with an error message.
4. The response is prepared by encrypting the response string, signing it and attaching the XML signature to the message. Finally, the response is sent to the client.

4.5.2. Explanation of WSDL File

To provide the content required for XML request and response documents exchanged between the client and the server, some complex types and elements have to be defined. The content of each XML request or response document is then built using these elements. Complex types and elements are similar to the idea of user-defined data types or classes in programming. For the secure m-banking service, the following elements and types were defined:

1. **Signature**: It is a complex type which models the structure of the customized XML signature that is being used in this project. The fields and their types inside this XML signature are:

   1.1. SignedInfo: Complex type element. Contains:

      1.1.1. CanonicalizationMethod: String

      1.1.2. SignatureMethod: String

        1.1.3. Reference: Complex element. Contains:
            1.1.3.1. DigestMethod: String
            1.1.3.2. DigestValue: String
            1.1.3.3. URI: String
    1.2. SignatureValue: String
    1.3. KeyInfo: Complex type element. Contains:
        1.3.1. KeyName: String

To illustrate this, the actual definition of the XML signature structure in the WSDL file is shown below:

```
<xsd:complexType name="Signature">  \\Defining Signature complex type
  <xsd:sequence>
    <xsd:element name="SignedInfo">    \\Defining SignedInfo element
      <xsd:complexType>                \\Complex type of SignedInfo
      <xsd:sequence>
        <xsd:element name="CanonicalizationMethod" type="xsd:string"/>
        <xsd:element name="SignatureMethod" type="xsd:string"/>
        <xsd:element name="Reference">
          <xsd:complexType>
            <xsd:sequence>
              <xsd:element name="DigestMethod" type="xsd:string"/>  \\string element
              <xsd:element name="DigestValue" type="xsd:string"/>
              <xsd:element name="URI" type="string"/>
            </xsd:sequence>
          </xsd:complexType>
        </xsd:element>
      </xsd:sequence>
      </xsd:complexType>
    </xsd:element>
    <xsd:element name="SignatureValue" type="xsd:string"/>
    <xsd:element name="KeyInfo">
      <xsd:complexType>
        <xsd:sequence>
          <xsd:element name="KeyName" type="xsd:string"/>
        </xsd:sequence>
      </xsd:complexType>
    </xsd:element>
  </xsd:sequence>
</xsd:complexType>
```

2. **accountNumber**:
    2.1. account:String
    2.2. signature: Signature type (defined above)

3. **transaction**: This is a complex type containing information about a transaction. It has the following fields:
    3.1. amount: String
    3.2. date: String
    3.3. detail: String

3.4. nature: String
3.5. signature: Signature

4. **transactionHistory**: It is an element which is an array of 0 or more objects of type transaction and a signature.

5. **transferMoney**: This is an element containing information that the server expects from the client to transfer money from the client's account to another account in the bank. It contains:
 5.1. receiveraccount: String
 5.2. receiverName: String
 5.3. receiverSortCode: String
 5.4. senderAccount: String
 5.5. amount: String
 5.6. signature: Signature

6. **transferMoneyResponse**: This is an element which contains the information returned by the server in response to a request for transfer of money. It contains:
 6.1. status: boolean
 6.2. signature: Signature

7. **balanceEnquiryResponse**: The information that the server needs to send when the client requests to check account balance is present in a balanceEnquiryResponse element. It contains:
 7.1. balance: String
 7.2. signature: Signature

For invoking an operation defined in the service the client sends a SOAP message and the server responds with another SOAP message. These messages contain data in the form of the elements and types defined above. Which elements and types are present in the request and response messages of each operation is defined in the WSDL file of the service. For instance, for calling the balanceEnquiry method, the client has to send a request containing the accountNumber element defined above which contains the account number and signature. The server responds with a message containing a balanceEnquiryResponse element containing the current balance and signature. This is described below:

1. **Balance Enquiry**
   Request message contains accountNumber element.
   Response contains balanceEnquiryResponse element.

   Hence, for balance enquiry the basic structure of the soap message body for the request and response SOAP messages should look something like shown below. Later in this section the actual messages captured using Java ME Network Monitor are also displayed for reference.

   Request:

```
<soap:Body>
  <tns:accountNumber>
    <account>15009090</account>
    <Signature>
```

```
            <SignedInfo>
               <CanonicalizationMethod>http://www.w3.org/1999/10/signature-
core/null</CanonicalizationMethod>
               . . .
            </SignedInfo>
            <SignatureValue>Dro2+HojvgjeWxGEIIOKdeG</SignatureValue>
            <KeyInfo>
              <KeyName>paras</KeyName>
            </KeyInfo>
        </Signature>
    </tns:accountNumber>
</soap:Body>
```

Response:

```
<soap:Body>
   <balanceEnquiryResponse>
        <balance>90.00</balance>
        <Signature>
           . . .
        </Signature>
      </balanceEnquiryResponse>
</soap:Body>
```

2. **Transfer Money**
   Request contains transferMoney element.
   Response contains transferMoneyResponse element.

3. **Transaction History**
   Request contains accountNumber element.
   Response contains transactionHistory element.

4.5.3. Generation of Service Skeleton

After writing the complete WSDL file, the web service is constructed from the WSDL definition. This is done using the Eclipse IDE which automatically generates the Java skeleton class files which contain the implementation of the complex elements defined in the WSDL and where the business logic for each operation has to be put. After putting the business logic in the skeleton, it is deployed on a web server like Apache Tomcat 5.5.

The files auto-generated by Eclipse from the WSDL definition contain a skeleton class and one Java bean class each for each of the complex type or element that was defined in the WSDL file. These files are listed and explained below:

1. **AccountNumber.java:** Java bean implementation of the type corresponding to the element accountNumber in the WSDL definition of the service. It has two inner fields – a String account and a Signature object. Getter and setter methods are present for each of these fields in the java bean.

2. **BalanceEnquiryResponse.java**: Java bean implementation of the type corresponding to the element balanceEnquiryResponse in the WSDL definition. It has two inner fields – a String amount and a Signature object. Getter and setter methods are present for each of the fields in the java bean.

3. **KeyInfo_type2.java**: Java bean implementation of the type corresponding to the element KeyInfo inside the Signature type in the WSDL definition. It has one inner field – a String keyName. Getter and setter methods are present for each of the fields in the java bean.

4. **Reference_type0.java**: Java bean implementation of the type corresponding to the element Reference inside the SignedInfo type in the WSDL definition. It has three inner fields – String digestMethod, String digestValue and String URI. Getter and setter methods are present for each of the fields in the java bean.

5. **SecureMBankingService1Skeleton.java**: The service skeleton class where the business logic of the service is put. It has the implementation of the three operations offered by the secure m-banking service to the client - balance enquiry, money transfer and transaction history. The client interacts with the service through the skeleton.

6. **Signature.java**: Java bean implementation of the complex type Signature defined in the WSDL file. It has the following fields – a SignedInfo_type1 object, a String SignatureValue and a KeyInfo_type2 object. Getter and setter methods are present for each of the fields in the java bean.

7. **SignedInfo_type1.java**: Java bean implementation of the type corresponding to the element SignedInfo inside the Signature type in the WSDL definition. It has three inner fields – a Reference_type0 object, String canonicalizationMethod, and String signatureMethod. Getter and setter methods are present for each of the fields in the java bean.

8. **Transaction.java**: Java bean implementation of the complex type transaction in the WSDL definition. It has the following inner fields – String date, String detail, String nature, String amount. Getter and setter methods are present for each of the fields in the java bean.

9. **TransactionHistory.java**: Java bean implementation of the type corresponding to the element transactionHistory in the WSDL definition. It contains an array containing zero or more objects of type Transaction and, a Signature object. Getter and setter methods are present for each of the fields in the java bean.

10. **TransferMoney.java**: Java bean implementation of the type corresponding to the element transferMoney in the WSDL definition. It has five inner fields – String senderAccount, String receiverName, String receiverAccount, String receiverSortCode, String amount and a Signature object. Getter and setter methods are present for each of the fields in the java bean.

11. **TransferMoneyResponse.java**: Java bean implementation of the type corresponding to the element transferMoneyResponse in the WSDL definition. Its inner fields – Boolean

status and a Signature object. Getter and setter methods are present for each of the fields in the java bean.

4.5.4. Business Logic in Skeleton

The skeleton of the secure m-banking service contains the three operations offered by the service. Explained below is the business logic that was put later in these functions:

Balance Enquiry

We start with the implementation of the balanceEnquiry() method of the service. This method accepts an argument of type AccountNumber which is implemented in the auto-generated Java bean AccountNumber.java explained earlier and returns a BalanceEnquiryResponse object which is implemented by the Java bean BalanceEnquiryResponse.java. Hence, the method signature of the balanceEnquiry method in the skeleton is:

```
public BalanceEnquiryResponse balanceEnquiry(AccountNumber accountNumber)
```

The work done inside the balanceEnquiry() method is explained in brief in the steps below and is elaborated later:

1. **Extracting values from the request:** The values of the encrypted account number and the XML Signature are extracted from the 'accountNumber' argument.
2. **Decryption:** The actual account number is obtained by decrypting the encrypted string.
3. **XML Signature Verification:** Using this plaintext account number, the XML signature is verified.
4. If any of the above processes fail, the client is notified with an error.
5. **XML Signature Formulation:** The current balance string is signed and an XML Signature is created for the response.
6. **Encryption:** The current balance string is encrypted.
7. A BalanceEnquiryResponse object is created using the encrypted current balance and the XML signature and sent to the client.

The above steps are in general followed by the other two banking operations as well with minor modifications. Each of these steps is now elaborated in detailed here.

Extracting Values from the Request

The AccountNumber argument in balanceEnquiry() call contains two fields – a string account and a Signature object according to its WSDL definition. Signature type is defined in the auto-generated file Signature.java. As defined earlier XML signature contains three complex inner types or fields – SignedInfo, Reference and KeyInfo. The corresponding auto-generated Java bean classes are called – Reference_type0, SignedInfo_type1, and KeyInfo_type2. Hence, for every complex type and element in the WSDL file, a java bean implementation class is generated along with the skeleton.

To begin with, we extract the information sent by the client inside the AccountNumber argument using the java bean getter methods. This is done as follows:

```
Signature signature = accountNumber.getSignature();
String encAccount = accountNumber.getAccount();
```

The string encAccount contains the account number for which the balance is being asked. However, this account number is encrypted with the secret key shared by the client and the server. Hence, to retrieve the account number, the string encAccount has to be decrypted. To do this, the verifyCipher() method is called.

Decryption Using verifyCipher()

The verifyCipher() method is created to decrypt encrypted strings. It has the following method signature:

```
private String verifyCipher(String base64CipherToVerify, String algorithm, byte[] keyBits,
String keyAlgorithm, byte[] ivBits)
```

The string to be decrypted is passed to the verifyCipher() method through the argument ‘base64CipherToVerify’ and the name of the encryption algorithm in the format "algorithm/mode/padding" as the string argument ‘algorithm’. The byte array ‘keyBits’ contains the secret shared key to be used for decryption, ‘keyAlgorithm’ contains the algorithm used for key generation and ‘ivBits’ has an initialization vector which is required by some encryption algorithms. As in this application, DES in ECB mode is being used, ivBits is null. It is however, still included to make the implementation more general and adaptable to the use of other encryption algorithms and modes.

The work done by verifyCipher() is outlined in steps below:

1. The encrypted string is in base64 format. It is first decoded using the Base64Coder.decode() method.
2. Then, decryption is carried out using the javax.crypto.Cipher class. Hence, the cipher is instantiated by passing the name of the algoithm, initialized using the key in the decryption mode and then decryption is carried out using the doFinal() method.
3. In case any exception is thrown during the decryption process, an error message is displayed and the method exits returning an error string to notify the client that decryption failed.
4. If the decryption is successful, the padding bits at the end due to the use of PKCS5 padding scheme are removed and the decrypted plaintext is returned.

Hence, in the case of balanceEnquiry() method, the verifyCipher() method returns the account number for which the current balance is required. The next step is to verify the signature attached to the message sent by the client to ensure that the client indeed sent the request and that the message was not tampered with. This is done by calling the verifySignature() method.

XML Signature Verification Using verifySignature()

The verifySignature() method is created to verify the signature supplied with a message from the client. It has the following method signature:

```
private boolean verifySignature(String text, Signature signature, String elementID)
```

The signature to be verified is passed to the verifySignature() method through the argument 'signature' along with the plaintext that was signed as the string 'text'and an XPath expression pointing to the part of the message that was signed in 'elementID'. If the signature is verified successfully, a boolean value 'true' is returned otherwise 'false' is returned.

The work of the verifySignature method is explained in the steps below:

1. The information in the tags in the signature passed to the method is extracted. This includes digestValue, digestMethod, URI, canonicalizationMethod, SignatureMethod, keyName and signatureValue.
2. Using the plaintext string passed to the method, digest is calculated using the java.security.MessageDigest class and encoded in Base64 format using the Base64Coder.encode() method.
3. This digest is compared with the digest value from the signature, the digestMethod from the signature is compared with the expected digest algorithm i.e., SHA-1, the URI with the elementID passed, cannonicalizationMethod with the identifier for null canonicalization method "http://www.w3.org/1999/10/signature-core/null", signatureMethod with the identifier for signature algorithm being used i.e., "des-sha1", digestMethod with the URI for the message digest algorithm being used http://www.w3.org/2000/09/xmldsig#sha1. This comparison is demonstrated in the following code snippet:

```
if(base64Digest.equals(digestValue) && keyName.equals("Paras") &&

     canonicalizationMethod.equals("http://www.w3.org/1999/10/signature-core/null")
          && signatureMethod.equals("des-sha1") &&
               digestMethod.equals("http://www.w3.org/2000/09/xmldsig#sha1") &&
                                        uri.equals(elementID) )
```

4. If any of these conditions is not satisfied or any exception is thrown, the method exits returning 'false' which notifies the client about failure of signature verification.
5. If all conditions are satisfied, the signature value is calculated from the digest by encrypting it with the shared key and base64 encoding and, is compared with the signature value obtained from the passed Signature object.
6. If they match, signature verification is successful and true is returned else false is returned indicating that the signature sent by the client is not correct.

Now that decryption and signature verification are complete, the server prepares its response to the client. It has to return a response of type BalanceEnquiryResponse. We first create an object of this class. The BalanceEnquiryResponse needs two values – a String containing the account balance and a Signature. But, the account balance being crucial data needs to be encrypted before putting it in the BalanceEnquiryResponse object and sending it to the client. In this application, pre-defined values have been used for server responses as the main aim of the application is to demonstrate the security of the architecture. Connection to a database may be added to the application to make it more practical. The procedure for connection to a MySQL database has been detailed earlier. To encrypt the balance string it is sent to a function called authorCipher() which is designed to encrypt plaintext.

Encryption Using authorCipher()

The authorCipher() method encrypts cleartext. It has the following method signature:

```
private String authorCipher(String algorithm, byte[] keyBits, String keyAlgorithm,
                                byte[] ivBits, byte[] plaintext)
```

The cleartext is sent as the byte array argument ‘plaintext’ along with the encryption algorithm in the format "algorithm/mode/padding" as the string argument ‘algorithm’. The byte array ‘keyBits’ contains the secret shared key to be used for decryption, ‘keyAlgorithm’ contains the algorithm used for key generation and ‘ivBits’ has an initialization vector which is required by some encryption algorithms. As in this application, DES in ECB mode is being used, ivBits is null. It is however, still included to make the implementation more general and adaptable to the use of other encryption algorithms and modes.

As explained previously, encryption and decryption is carried out using the javax.crypto.Cipher class. For encryption, a Cipher object is first created and instantiated by supplying it with the algorithm argument and initialized in the encryption mode with the key. Data is encrypted using the doFinal() method and base64 encoded to give the final result which is returned.

Having encrypted the current balance which has to be sent to the client in a balanceEnquiry call, the XML signature which is also a part of balanceEnquiryResponse has to be created. The calculation of the XML signature is done in a separate method called authorSignature().

Xml Signature Formulation Using authorSignature()

The authorSignature() method creates the Signature object attached to SOAP messages. It has the following method signature:

```
private Signature authorSignature(String text, String elementReference)
```

The cleartext to be signed is passed as the parameter ‘text’ along with an XPath expression indicating the portion of the message being signed as ‘elementReference’. To facilitate the explanation of the creation of the XML signature, the hierarchy of elements in the XML signature is shown below for reference:

```
Signature
        SignedInfo
                SignatureMethod
                CanonicalizationMethod
                Reference
                        DigestMethod
                        DigestValue
                        Reference
        SignatureValue
        KeyInfo
```

The work done by authorSignature() is detailed in the following steps:

1. The plaintext passed to the method is used to calculate SHA-1 digest and then encoded to base64 format.
2. A Signature object is created.
3. A Reference_type0 object is first created and its three string fields are set using the setter methods provided by its java bean implementation. The digestMethod is set to 'http://www.w3.org/2000/09/xmldsig#sha1' as SHA-1 algorithm is being used, digestValue is set with the digest calculated earlier and the URI is set with the XPath expression passed to authorSignature(). The following code excerpt shows this:

```
Signature signature = new Signature();
Reference_type0 reference = new Reference_type0();
reference.setDigestMethod("http://www.w3.org/2000/09/xmldsig#sha1");
reference.setDigestValue(base64Digest);
reference.setURI(elementReference);
```

4. Similarly, a SignedInfo_type1 instance is created and the values 'http://www.w3.org/1999/10/signature-core/null' and 'des-sha1' are set in the canonicalizationMethod and signatureMethod fields using the setter methods as shown above. The Reference_type0 object created above is used to set the 'reference' parameter of SignedInfo_type1.
5. Likewise, a KeyInfo_type2 object is created and its keyName parameter is set with a value which identifies the secret shared key.
6. Lastly, the digest that was calculated earlier is encrypted and converted to base64 format. This gives the signature value.
7. The SignedInfo_type1, KeyInfo_type2 and the signature value are used to set the parameters of the Signature object which is returned by the method. This is shown here:

```
signature.setKeyInfo(keyInfo);
signature.setSignatureValue(signString);
signature.setSignedInfo(signedInfo);
return signature;
```

Now, after encryption and signature calculation, we set the encrypted account balance and the signature that has been calculated in the balanceEnquiryResponse object and send it to the client.

```
BalanceEnquiryResponse resp = new BalanceEnquiryResponse();
resp.setBalance(encCurrentBalance);
resp.setSignature(authorSignature(currentBalance, "//balanceEnquiryResponse/balance"));
return resp;
```

Using the Network Monitor of Java ME WTK, the request and response messages between the client and server for the balance enquiry operation were captured and are displayed here:

Request:

```
<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:tns="http://www.parasmehta.net/securembanking">
```

```
   <soap:Body>
      <tns:accountNumber>
         <account>ME9OxAU3uuI9ItE8PU1fHg==</account>
         <Signature>
            <SignedInfo>
               <CanonicalizationMethod>http://www.w3.org/1999/10/signature-
core/null</CanonicalizationMethod>
               <SignatureMethod>des-sha1</SignatureMethod>
               <Reference>
                  <DigestMethod>http://www.w3.org/2000/09/xmldsig#sha1</DigestMethod>
                  <DigestValue>VBbIWTGpe4ums67FXiPbrWmB9v4=</DigestValue>
                  <URI>//accountNumber/account</URI>
               </Reference>
            </SignedInfo>
            <SignatureValue>HV7dDro2+HojvgjeWxGEIIOKdeG14DBJ</SignatureValue>
            <KeyInfo>
               <KeyName>paras</KeyName>
            </KeyInfo>
         </Signature>
      </tns:accountNumber>
   </soap:Body>
</soap:Envelope>
```

Response:

```
<?xml version="1.0" encoding="utf-8"?>
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/">
   <soapenv:Body>
      <ns1:balanceEnquiryResponse xmlns:ns1="http://www.parasmehta.net/securembanking">
         <balance>bPSJ0m0fOI8=</balance>
         <Signature>
            <SignedInfo>
               <CanonicalizationMethod>http://www.w3.org/1999/10/signature-
core/null</CanonicalizationMethod>
               <SignatureMethod>des-sha1</SignatureMethod>
               <Reference>
                  <DigestMethod>http://www.w3.org/2000/09/xmldsig#sha1</DigestMethod>
                  <DigestValue>fy9qFc+NorJ+Wkr0e1jnrXHAs9k=</DigestValue>
                  <URI>//balanceEnquiryResponse/balance</URI>
               </Reference>
            </SignedInfo>
            <SignatureValue>f44r7V+xqCyMroEAwUp9W5vFBGgC/wo/</SignatureValue>
            <KeyInfo>
               <KeyName>paras</KeyName>
            </KeyInfo>
         </Signature>
      </ns1:balanceEnquiryResponse>
   </soapenv:Body>
</soapenv:Envelope>
```

Transfer Money

The next method in the skeleton that has to be implemented is the transferMoney() method. This method accepts an argument of type TransferMoney which is implemented in the auto-generated Java bean TransferMoney.java explained earlier and returns a

TransferMoneyResponse object which is implemented by the Java bean TransferMoneyResponse.java. Hence, the method signature is:

public TransferMoneyResponse transferMoney (TransferMoney transferMoney)

The steps involved in the working of transferMoney() are quite similar to those explained in balanceEnquiry(). TransferMoney argument contains five string fields – sender’s account number, destination account holder’s name, destination account name, destination sort code and amount – and, a Signature object.

To begin with, the information inside the TransferMoney argument is extracted. The information in the strings is encrypted. Hence, verifyCipher() method is called and the strings are decrypted. Subsequently, the decrypted strings are used to verify the XML signature. The five strings are concatenated to form a single string and sent to the verifySignature() method to check whether the signature matches. If yes, a TransferMoneyResponse element is created and prepared. It consists of two fields – a boolean status field and a signature. If any exception occurs during the decryption or signature verification or, if the signature verification fails, the status is set to false otherwise the status is set to true to indicate that the fund transfer took place successfully.
The signature is calculated using authorSignature() method by passing the string value of the boolean status variable. Once the response is created it is sent to the client.

Transaction History

The third and the last method that the service offers the client is transactionHistory(). It takes an argument of type AccountNumber which was also used by the balanceEnquiry() method. AccountNumber is implemented in the auto-generated Java bean AccountNumber.java explained earlier and TransactionHistory in the Java bean TransactionHistory.java. Hence, the method signature is:

public TransactionHistory transactionHistory (AccountNumber accountNumber0)

The AccountNumber argument contains a string account number for which the transaction record is required and a signature.

First, the encrypted account number and the signature are extracted from the argument ‘accountNumber0’. Then, verifyCipher() is called to retrieve the account number in plaintext. This account number is passed to verifySignature() to verify the XML signature. In the event of occurrence of an exception during decryption and signature or signature mismatch, an error response is returned.

If the decryption and signature verification are both successful, TransactionHistory element has to be created and sent to the client. TransactionHistory is an array of Transaction objects. The type Transaction is defined in the auto-generated Java bean Transaction.java. In this application the transactionHistory() method returns the last ten transactions in the account. Hence, the TransactionHistory object would be an array of ten Transaction objects. To start with, a TransactionHistory object has to be created.

TransactionHistory resp = new TransactionHistory();

Next, a Transaction object is instantiated.

```
Transaction tr1 = new Transaction();
```

The Transaction object’s four inner fields are set with encrypted content using the setter methods and by calling authorCipher().

```
String amount = “10.00”;
String date = “01-09-2007”;
String nature = “Debit”;
String detail = “Metro Madrid”;
tr1.setAmount(authorCipher("DES/ECB/PKCS5Padding", kDESKey, "DES", null,
amount.getBytes()));
tr1.setDate(authorCipher("DES/ECB/PKCS5Padding", kDESKey, "DES", null,
date.getBytes()));
tr1.setNature(authorCipher("DES/ECB/PKCS5Padding", kDESKey, "DES", null,
nature.getBytes()));
tr1.setDetail(authorCipher("DES/ECB/PKCS5Padding", kDESKey, "DES", null,
detail.getBytes()));
```

Lastly, this Transaction instance is added to the TransactionHistory object.

```
resp.addTransaction(tr1);
```

Similarly, ten Transaction objects have to be put be created and added to the TransactionHistory object.

To calculate the signature, we append the string values of all fields in each Transaction in the TransactionHistory object and calculate the signature of the resulting concatenated string using authorSignature(). This signature is added to the TransactionHistory object which is returned to the client.

4.5.5. Implementation of Client

The mobile client is developed in the J2ME Wireless Tool Kit (WTK) 2.5.2. The first step to develop the mobile client to access the secure m-banking service is to create a new project and MIDlet in the WTK. For accessing the web service, the client uses stub classes. Stubs are generated using the Stub Generator tool in WTK by supplying the location of the WSDL file of the secure m-banking service. The stub classes are also stored inside the project folder so that the client MIDlet can access them.

4.5.6. Generation of Client Stub

The Stub Generator generated the following stub classes:

1. **AccountNumber.java**: Java bean implementation of the type corresponding to the element accountNumber in the WSDL definition of the service. It has two inner fields – a String account and a Signature object.

2. **BalanceEnquiryResponse.java**: Java bean implementation of the type corresponding to the element balanceEnquiryResponse in the WSDL definition. It has two inner fields – a String amount and a Signature object.

3. **SecureMBankingService1_PortType.java**: It is a local interface for the service containing the declarations of the operations offered by the service. The stub implements this interface.

4. **TransferMoneySignatureKeyInfo.java**: Java bean implementation of the type corresponding to the element KeyInfo inside the Signature type in the WSDL definition. It has one inner field – a String keyName.

5. **TransferMoneySignatureSignedInfoReference.java**: Java bean implementation of the type corresponding to the element Reference inside the TransferMoneySignatureSignedInfo type in the WSDL definition. It has three inner fields – String digestMethod, String digestValue and String URI.

6. **SecureMBankingService1_PortType_Stub.java**: The stub class used by the client to access the web service. The MIDlet instantiates the stub class and invokes the operations on the stub like local methods. The stub handles the job of invoking the operations remotely on the service.

7. **Signature.java**: Java bean implementation of the complex type Signature defined in the WSDL file. It has the following fields – a TransferMoneySignatureSignedInfo object, a String SignatureValue and a TransferMoneySignatureKeyInfo object. Getter and setter methods are present for each of the fields in the java bean.

8. **TransferMoneySignatureSignedInfo.java**: Java bean implementation of the type corresponding to the element SignedInfo inside the Signature type in the WSDL definition. It has three inner fields – a TransferMoneySignatureSignedInfoReference object, String canonicalizationMethod, and String signatureMethod. Getter and setter methods are present for each of the fields in the java bean.

9. **Transaction.java:** Java bean implementation of the complex type transaction in the WSDL definition. It has the following inner fields – String date, String detail, String nature, String amount.

10. **TransactionHistory.java:** Java bean implementation of the type corresponding to the element transactionHistory in the WSDL definition. It contains an array containing zero or more objects of type Transaction and, a Signature object.

11. **TransferMoney.java**: Java bean implementation of the type corresponding to the element transferMoney in the WSDL definition. It has five inner fields – String senderAccount, String receiverName, String receiverAccount, String receiverSortCode, String amount and a Signature object.

12. **TransferMoneyResponse.java**: Java bean implementation of the type corresponding to the element transferMoneyResponse in the WSDL definition. Its inner fields – Boolean status and a Signature object.

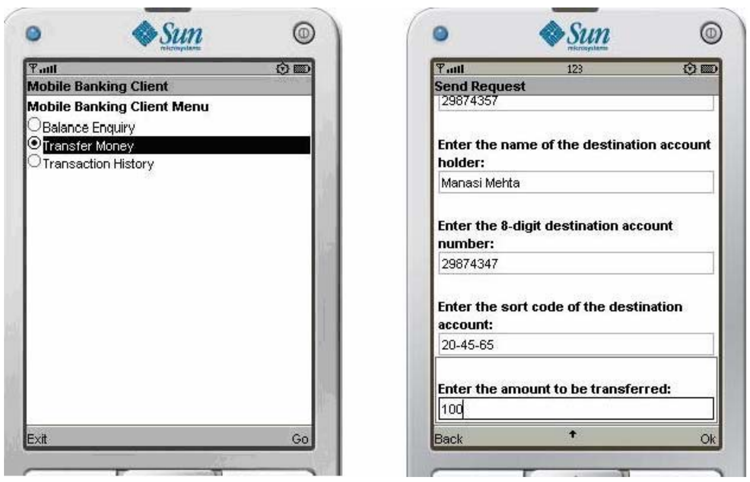
Finally, a MIDlet is designed which contains the necessary application code of the secure m-banking mobile client and uses the stub and java bean classes generated to interact with the secure m-banking service.

4.5.7. The Secure M-Banking MIDlet

The mobile client has to provide the user with an interface for carrying out transactions. It has to work as a web service client to request the operations invoked by the user. The design of the client is similar to that of the server.

The essential steps in the functioning of the MIDP client are:

1. The mobile client is equipped with an interface for the user. It displays a menu containing the list of three operations that he can carry out in his account in the bank. The user chooses one operation depending on which another screen is displayed with text fields to input data necessary to create a request to be sent to the client e.g., account number. This is illustrated in the figures below:

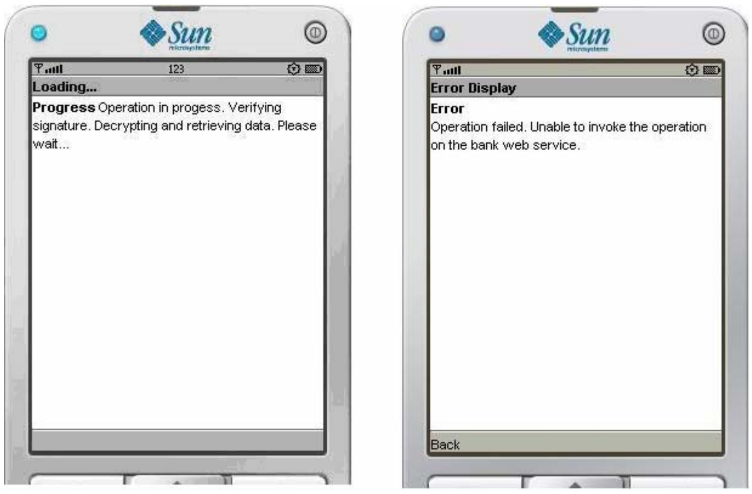

Mobile Banking Client
Mobile Banking Client Menu
○ Balance Enquiry
● Transfer Money
○ Transaction History
Exit Go

Send Request
29874357
Enter the name of the destination account holder:
Manasi Mehta
Enter the 8-digit destination account number:
29874347
Enter the sort code of the destination account:
20-45-65
Enter the amount to be transferred:
100
Back Ok

Figure 4. Creating the request

2. An instance of the stub class is created. The data entered by the client is read and is used to create a request to the server. It is encrypted like it was done in the server, by calling a method called authorCipher(). The method has nearly the same implementation as the server as the Cipher and SecretKeySpec classes are a part of both Java SE and Java ME. Secondly, the data is used to design XML signature using a method called authorSignature() which is also very similar to the authorSignature() in the server. This encrypted message and the signature are used as the input and the operation is invoked using the stub. An error message is displayed if connection to the server or remote method invocation fails. This is illustrated here:

```
stubObj.transferMoney(input);
```

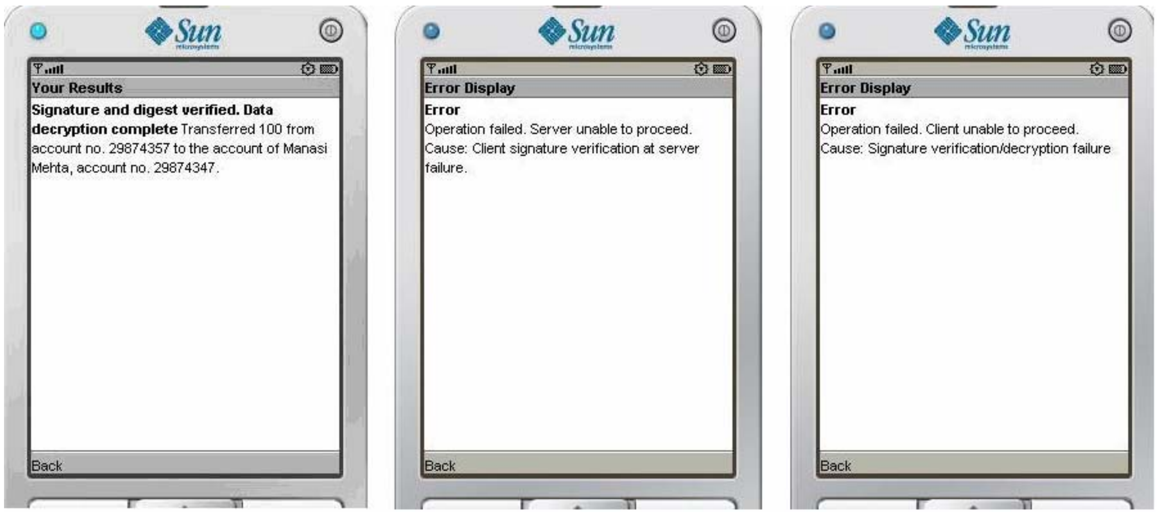


Loading...
Progress Operation in progess. Verifying signature. Decrypting and retrieving data. Please wait...

Error Display
Error
Operation failed. Unable to invoke the operation on the bank web service.
Back

Figure 5. Invoking the banking operation

3. After method invocation, the response from the server is received. The required data is extracted from the response and decrypted by calling a function called verifyCipher() resembling the verifyCipher() method of the server. If the decryption fails, the client is displayed an error message otherwise, the decrypted data is used to match the XML signature using the verifySignature() method which is again quite similar to verifySignature() in the server side. If verification fails, an error is displayed on the screen else, the results of the operation are shown on the screen to the user.



Your Results
Signature and digest verified. Data decryption complete Transferred 100 from account no. 29874357 to the account of Manasi Mehta, account no. 29874347.
Back

Error Display
Error
Operation failed. Server unable to proceed. Cause: Client signature verification at server failure.
Back

Error Display
Error
Operation failed. Client unable to proceed. Cause: Signature verification/decryption failure
Back

Figure 6. Displaying the results

4.5.8. Deploying the Client on a Mobile Device

The secure m-banking client has been developed with the Java ME WTK and tested on the simulator. Till now, both the server and the client were functioning on the same terminal – localhost. However, this is not usually the case in real applications and working on a simulator and on a real device may actually be quite different. Hence, to better demonstrate the functioning of the architecture and the application, the client was deployed on an actual smartphone – Nokia E61. A picture of the mobile client on the phone is shown below.

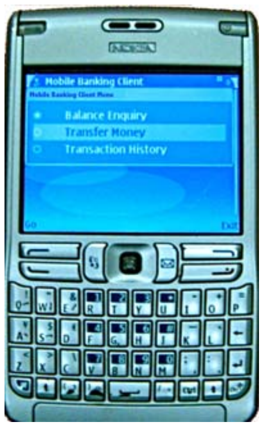


Figure 7. Secure m-banking client on Nokia E61

The deployment was carried out in the following steps:

1. The client application had to be altered to be developed with MIDP 2.0 and not MIDP 2.1. This was changed in the project properties in the WTK.
2. The endpoint address of the service in the stub was changed from localhost to the IP address of the service.
3. The service was compiled and packaged into a JAR file with a JAD file using the WTK.
4. The phone was connected to the PC (using a data cable, Bluetooth or infra-red connection). The JAR and JAD files were transferred to the phone using the Nokia Application Installer tool in the Nokia PC Suite application present on the PC.
5. On the device, installation started automatically and was completed in a few steps. The secure m-banking client was installed on the device and is ready for use. The communication between client and server was done using WLAN.

5. Conclusion

5.1. Evaluation against Goals and Milestones

The project work has fulfilled all the goals and milestones that were laid at the outset shown in the ‘Introduction’ part. The important achievements of the project are the following:

- Initially, an in-depth background study and analysis of the existing m-commerce architectures and, the tools and concepts necessary for the project work was done concentrating on m-banking.
- A Security Analyser application was developed to model and evaluate the security offered by the username-password authentication mechanism.
- Subsequently, key security threats to m-commerce applications were found and security mechanisms were devised to address the threats and limitations in current application architectures.
- Finally, a novel and improved architecture was proposed that guarantees higher security using the advantages of Web Services and modern security concepts like XML signature and, is also viable.
- To demonstrate the implementation strategy and feasibility of this proposed architecture, a secure m-banking application was designed and deployed on a real smartphone – Nokia E61.
- Additionally, functioning of different styles of web services – RPC based and Document based – , XML parsing using SAX, addition of XML signature and XML encryption using Apache Rampart module was also experimented with to gain a better insight into the subject.
- A significant amount of research was also done in attempting to use Java ME SATSA-PKI package for signing strings. Several approaches were tried and although in due course it was concluded that it is not possible to use SATSA PKI without Java Card functionality, it gave a better understanding of concepts such as the openssl tool, the PKCS standards, digital certificate encodings and formats, functionality of SATSA PKI classes and the SATSA Reference Implementation, etc., which proved quite useful in the remainder of the project.
- As Java Card is not being used in the project, an alternate approach to the above problem was devised. The signature was calculated by encrypting the digest of the message with a symmetric shared key. This met all the security requirements of the project and had the advantage of not using SATSA PKI classes.
- The subject of the research and development work carried out in the project is quite modern and emerging. The solutions proposed and developed are innovative and practical and, can prove to be a very useful advancement both in the research and the practical development work in the field of m-commerce, m-banking and Java ME web services security.
- For instance, JSR 177 (SATSA), JSR 172 (J2ME Web Services), XML Signature, etc. are relatively new specifications with scope and use in the future.
- However, on several occasions this also hindered the progress of the project work as often the software and tools lacked adequate documentation, support and functionality or suffered with bugs.

5.2. Strengths and Weaknesses

Some major strengths of the work done in the project are listed here:

1. The Secure M-Commerce architecture is based on open, interoperable model of Web Services. The client uses the relatively new and upcoming J2ME Web Services API to access the world of Web Services.
2. The project combines various up-to-date security technologies and concepts to build an improved secure architecture. This includes XML Signature, encryption, J2ME SATSA, SSL.
3. As in the Secure M-Banking application, the key can be embedded in the client. This saves the user of the hassle of entering a username and password over and over again. Also, dictionary attacks become difficult as the key can be just illegible binary data.
4. Authentication and non-repudiation are provided by the customized XML Signature used in the architecture. Moreover, due to the presence of message digest in the XML Signature, message integrity is also ensured.
5. Encryption of message content provides confidentiality.
6. An additional tier of security is added through the use of SOAP over SSL.
7. XML message templates were designed to standardise the structure and content of the messages exchanged between the client and the server. This facilitates standardisation and interoperability.
8. The architecture allows flexibility in encryption. Instead of encrypting the entire message, specific parts can be encrypted. Also, different keys may be used for encrypting different parts of the same message to target them to different receivers. This also ensures end-to-end security.
9. As SSL is transparent to the application logic, its use is optional. To use HTTP, the endpoint address in the client stub should be of the form http://address:portno1 while to use HTTPS, of the form https://address:portno2. For meeting very low bandwidth and processing power constraints of some mobile devices, this may be useful.

Important weaknesses of the project work are:

1. Due to the presence of a large number of tags, XML uses some extra bandwidth and requires processing. For mobile devices, bandwidth and processing power are usually limited.

5.3. Future Work

- Java Card functionality can be incorporated into the project. This would allow the mobile client to store and access cryptographic data and credentials such as certificates, keys, etc. on a smart card such as a SIM card.
- The use of Java Card would also allow use of PKI signatures using RSA or DSA algorithm. This can be done using the SATSA PKI package as explained before.
- On the server side, a database can be added to better simulate a bank server. The bank server would connect to the database and fetch the details of the bank accounts and credentials.

6. References

[1] “Citi Mobile”, Citibank, [Online] Available http://www.citibank.com/citimobile, Last accessed November 2007
[2] “Wachovia Mobile”, Wachovia Bank, [Online] Available http://www.wachovia.com/mobile, Last accessed November 2007
[3] “Vipera MoBank”, Vipera Networks [Online] Available http://www.viperanetworks.com/content/view/48/107, Last accessed December 2007
[4] “dotMobi Website”, mTLD Top Level Domain Ltd. [Online] Available http://mtld.mobi/, Last accessed February 2008
[5] “The SSL Protocol Version 3.0 Internet Draft”, IETF [Online] Available http://wp.netscape.com:80/eng/ssl3/draft302.txt, Last accessed Jan 2008
[6] “J2ME Web Services APIs (WSA)”, Sun Microsystems, Inc. [Online] Available http://java.sun.com/products/wsa/, Last accessed December 2007
[7] “XML Signature Syntax and Processing”, W3C Recommendation [Online] Available http://www.w3.org/TR/xmldsig-core/, Last accessed February 2008
[8] “Security and Trust Services API for J2ME (SATSA); JSR 177”, Sun Microsystems, Inc. [Online] Available http://java.sun.com/products/satsa/, Last accessed January 2008

Knudsen, Jonathan. “MIDP Application Security 1.” [Online] Available http://developers.sun.com/mobility/midp/articles/security1/, September 2002
Knudsen, Jonathan. “MIDP Application Security 2.” [Online] Available http://developers.sun.com/mobility/midp/articles/security2/, October 2002
Knudsen, Jonathan. “MIDP Application Security 3.” [Online] Available http://developers.sun.com/mobility/midp/articles/security3/, December 2002
Knudsen, Jonathan. “MIDP Application Security 4.” [Online] Available http://developers.sun.com/mobility/midp/articles/security4/, September 2005
Knudsen, Jonathan. “Understanding MIDP 2.0's Security Architecture.” [Online] Available http://developers.sun.com/mobility/midp/articles/permissions/, Feb 2003
Mahmoud, Qusay. “Secure Java MIDP Programming Using HTTPS with MIDP.” [Online] Available http://developers.sun.com/mobility/midp/articles/https/, June 2002
Mahmoud, Qusay. “Wireless Java Security.” [Online] Available http://developers.sun.com/mobility/midp/articles/security/, January 2002
Ortiz, C. Enrique. “The Security and Trust Services API for J2ME, Part 1.” [Online] Available http://developers.sun.com/mobility/apis/articles/satsa1/, March 2005
Ortiz, C. Enrique. “The Security and Trust Services API (SATSA) for J2ME: The Security APIs” [Online] Availablehttp://developers.sun.com/mobility/apis/articles/satsa2, Sept 2005
Oritz, C. Enrique. “Web Services APIs for J2ME, Part 1." [Online] Available http://www.ibm.com/developerworks/wireless/library/wi-jsr/, 20 Jul 2004
Oritz, C. Enrique. “Web Services APIs for J2ME, Part 2." [Online] Available http://www.ibm.com/developerworks/wireless/library/wi-xmlparse/, 02 Nov 2004
Rui, Shu Fang. “Designing mobile Web services." [Online] Available http://www.ibm.com/developerworks/wireless/library/wi-websvc/, 03 Jan 2006
Sankaran, Suraj. “Mobile Banking Architecture” [Online] Available http://palisade.plynt.com/issues/2007May/mobile-banking/, May 2007

Siddiqui, Bilal. “Building a secure SOAP client for J2ME, Part 1." [Online] Available http://www.ibm.com/developerworks/edu/ws-dw-ws-soa-securesoap1.html, 16 Jun 2006

Siddiqui, Bilal. “Building a secure SOAP client for J2ME, Part 2." [Online] Available http://www.ibm.com/developerworks/edu/ws-dw-ws-soa-securesoap2.html, 04 Aug 2006

Siddiqui, Bilal. “Building a secure SOAP client for J2ME, Part 3." [Online] Available http://www.ibm.com/developerworks/edu/ws-dw-ws-soa-securesoap3.html, 19 Jan 2007

Yuan, Michael Juntao. “Developing J2ME applications with EclipseME." [Online] Available https://www6.software.ibm.com/developerworks/education/wi-nokia, 18 Jun 2002

Yuan, Michael and Long, Ju. “Securing wireless J2ME." [Online] Available http://www.ibm.com/developerworks/wireless/library/wi-secj2me.html, 01 Jun 2002

Yuan, Michael. “Securing your J2ME/MIDP apps." [Online] Available http://www.ibm.com/developerworks/library/j-midpds.html, 18 Jun 2002

“Apache Axis2 Website”, The Apache Software Foundation [Online] Available http://ws.apache.org/axis2/, Last accessed February 2008

“Apache Tomcat Website”, The Apache Software Foundation [Online] Available http://tomcat.apache.org/, Last accessed January 2008

“Eclipse Web Tools Platform Project”, Eclipse Foundation [Online] Available http://www.eclipse.org/webtools/, Last accessed November 2007

“Forum Nokia”, Nokia [Online] Available http://www.forum.nokia.com/, Last accessed March 2008

“Getting Started With Web Services, JSR 172.”, Sony Ericsson Developer World, [Online] Available http://developer.sonyericsson.com, September 2006

“Sony Ericsson Developer World”, Sony Ericsson Mobile Communications AB [Online] Available http:// developer.sonyericsson.com/, Last accessed December 2007

“Sun Java Wireless Toolkit for CLDC”, Sun Microsystems, Inc. [Online] Available http://java.sun.com/products/sjwtoolkit/, Last accessed January 2008

7. Appendices

7.1. Appendix 1: Security Analyser Application User Manual

Preparing the Username-Password Authentication Server

1. Use of the Eclipse IDE is recommended. Eclipse Web Tools Platform (WTP) is equipped with tools and plug-ins for web service and web development. Eclipse WTP can be downloaded from http://www.eclipse.org/webtools/

2. Download Apache Tomcat 5.5 web server from http://tomcat.apache.org/ and Apache Axis2 from http://ws.apache.org/axis2/ . Apache Tomcat 5.5.25 and was used in during the development and testing of the application. Specify Tomcat runtime location in Eclipse.

3. Configure Tomcat for using HTTPS on port 8443 by editing the server.xml file and putting the .keystore file in the user's home directory.

4. Create a new Java project named HTTPSTestServer in Eclipse specifying the location of source directory as the SecureMBanking1 directory supplied with this application.

5. Run the project on the Apache Tomcat Server. Once deployed, the servlet can be viewed by typing 'https://localhost:8443/HTTPSTestServer/HitServlet' in the web browser.

Preparing the Mobile Client

1. Start the Sun Wireless Toolkit. Open the project MIDletTut.

2. Run the project. A new emulator window pops up.

Using the Application

1. Start the MySQL database server. Deploy and run the servlet on the web server. Execute the client in Sun WTK.

2. The client shows two MIDlets: HitMIDlet and TestMIDlet. Select one of them and press 'Launch'. Depending on your choice, following the corresponding set of instructions below:

HitMIDlet

3. Enter the username and password in the respective fields and press 'Send' or else press 'Quit'.

4. The message that follows shows whether the username and password combination was correct or not. After a few seconds, the display automatically returns to the username – password form.

TestMIDlet

5. Enter the URL of the servlet running on the web server and the username for which the password has to be retrieved. The URL should be of the form: ‘https://address:portno/Servlet’.

6. Press ‘Start’ and wait while the client tries password combinations. The result is displayed in the next screen.

7.2. Appendix 2: Secure M-Banking Application User Manual

Preparing the Secure M-Banking Service

6. Use of the Eclipse IDE is recommended. Eclipse Web Tools Platform (WTP) is equipped with tools and plug-ins for web service and web development. Eclipse WTP can be downloaded from http://www.eclipse.org/webtools/

7. Download Apache Tomcat 5.5 web server from http://tomcat.apache.org/ and Apache Axis2 from http://ws.apache.org/axis2/ . Apache Tomcat 5.5.25 and Axis2 version 1.2 was used in during the development and testing of the application. Specify Tomcat and Axis2 runtime location in Eclipse.

8. Configure Tomcat for using HTTPS on port 8443 by editing the server.xml file and putting the .keystore file in the user's home directory.

9. Create a new Java project named SecureMBanking1 in Eclipse specifying the location of source directory as the SecureMBanking1 directory supplied with this application.

10. Run the project on the Apache Tomcat Server. Once deployed, the service can be viewed by typing https://localhost:8443/SecureMBanking1/ in the web browser.

Preparing the Secure M-Banking Mobile Client

3. Start the Sun Wireless Toolkit. Open the project SecureMBankingClient1.

4. Run the project. A new emulator window pops up.

Using the Application

7. Deploy and run the web service on the web server. Run the client on Sun WTK.

8. Launch the application in the client. Press Connect to go to the main menu of the client or Exit to quit.

9. The subsequent screen shows a list of three operations:

   Balance Enquiry: To show the current balance in the account.
   Transfer Money: To transfer money to another account in the bank.
   Transaction History: To display the last ten transactions in the account.

10. Select one of the options and press 'Go'. To quit, press 'Exit'.

11. According to the operation chosen refer to the corresponding section below:

Balance Enquiry

12. Selecting Balance Enquiry option shows another screen asking for the account number. Enter the account number of the account whose balance is sought and press ‘Ok’ or press ‘Back’ to return to the main menu.

13. The account balance is shown in the result screen. Press ‘Back’ to return to the main menu.

Transfer Money

14. In the main menu browse to the ‘Transfer Money’ option, select it and press ‘Go’.

15. In the next screen enter the details of the money transfer as asked and press ‘Ok’ or press ‘Back’ to return to the main menu. Press ‘Back’ to return to the main menu.

16. The status of the transfer is shown in the next screen.

Transaction History

17. In the main menu browse to the ‘Transaction History’ option, select it and press ‘Go’.

18. Next, enter the account no. for which the last ten transactions are required and press ‘Ok’ or press ‘Back’ to return to the main menu..

19. The subsequent screen shows the details of the last ten transactions of the specified account. Press ‘Back’ to return to the main menu.

Error Messages

Error Message	Explanation
1. Unable to invoke the operation on the bank web service.	This shows an error in communicating with the server. Make sure the service is deployed and running before using the client.
2. Server unable to proceed. Signature verification/decryption failure.	It indicates that the decryption of the encrypted text and/or verification of the XML signature received by the server from the client failed. This may be possible due to damage or tampering of data during transit from the client to the server.
3. Client unable to proceed. Signature verification/decryption failure.	It means that the decryption of the encrypted text and/or verification of the XML signature received by the client from the server failed. This may be possible due to damage or tampering of data during transit from the server to the client.