A Possible Secure Mobile Web Service

Goran Đorđević, *The Institute for Manufacturing banknotes and coins NBS*, Milan Marković, *Banca Intesa ad Beograd*

***Abstract* — In this paper, a design and programming of JAVA applications on mobile phones that securely connect to Web services are described. We considered a Web service scenario where mobile phone user produces a cryptographic signature in the JAVA application using the smart card. This smart card is generally considered in the paper but a main intention is that this should be a PKI SIM smart card. Data is encrypted using a crypto MIDlet JAVA application installed on mobile phone with CLDC configuration. The user uses XML signature to wrap a cryptographic signature into the SOAP request and sends the request over to the remote Web service endpoint implementation. Web service performs request processing and sends SOAP response back to the WSA (Web Service API) framework. WSA processes the SOAP response and display the status to the mobile user on his mobile phone. The work presented is related to the EU IST FP6 SWEB project (Secure, interoperable cross border m-services contributing towards a trustful European cooperation with the non-EU member Western Balkan countries) [1].**

***Keywords* — Java mobile application, mobile phone with CLDC configuration, MIDlet, smart card, SOAP protocol, SWEB, XML Signature, Web service.**

I. INTRODUCTION

Java 2 Micro Edition (J2ME) is a runtime environment for resource-constrained environments. J2ME includes specific virtual machines, configurations and profiles for various environments and needs. With an appropriate configuration and profile, J2ME applications could be executed within pagers, mobile phones, PDAs, set-top boxes and automobile navigation systems, just to mention some [2].

The Java Specification Request 172 (JSR 172) specifies standardized client-side technology to enable J2ME applications to consume remote services on typical web services architectures, as Fig. 1 illustrates [3].

JSR 172 defines a standardized API that J2ME clients can use to invoke SOAP and XML-based Web services. This API is in the form of an optional package for J2ME, and is referred to as Web Services APIs (WSA) for J2ME. WSA is actually a subset of the Java API for XML-based Remote Procedure Call (JAX-RPC) defined by JSR 101 [4].

JAX-RPC uses the popular concept of Web service "endpoints" and "clients". Endpoints expose Web services and JAX-RPC clients invoke – access, consume or make use of – the services exposed by endpoints.

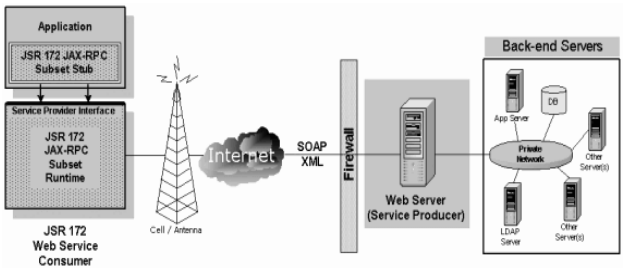

Figure 1. J2ME Web Services in a Typical Web Service Architecture

WSA, as a subset of JAX-RPC, only includes the set of interfaces that are used to define Web service clients. This makes sense because J2ME devices are not likely to expose their own Web service endpoints. J2ME devices are only expected to consume Web services exposed by service endpoints. This scenario is depicted in Fig. 2 [5].

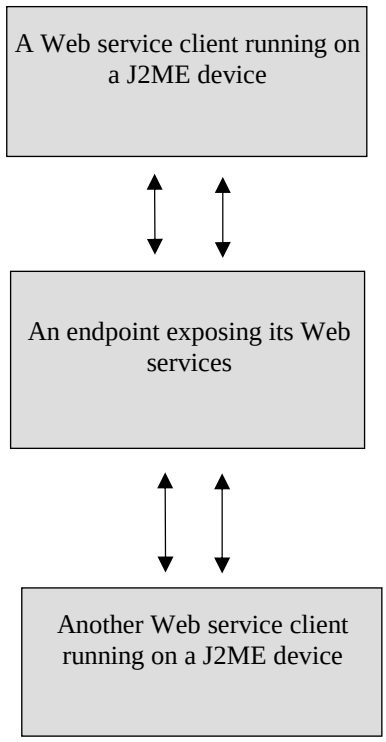

Figure 2. Web service consuming services exposed by a Web service endpoint

Goran Đorđević, The Institute for Manufacturing banknotes and coins NBS, Pionirska 2, 11000 Beograd

Milan Marković, Banca Intesa ad Beograd, Bulevar Milutina Milankovića 1c, 11070 Novi Beograd

Before we focus on the Java ME subset of JAX-RPC, let's review the general concepts and elements of a typical JAX-RPC application (see Fig. 3).

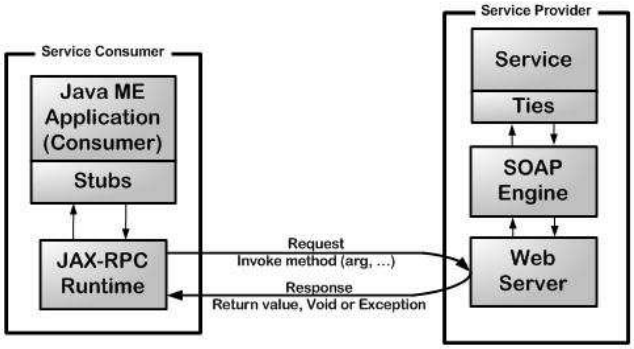

Figure 3. Elements of a Typical JAX-RPC Application

When examining a JAX-RPC application, in our case an application based on Java ME, we should focus on three main areas of interest: the service consumer, the service provider, and the network.

The **Service Consumer** consists of the application, the JAX-RPC stubs for the service of interest, and the JAX-RPC runtime. On Java ME the JAX-RPC stubs and runtime are based on the JSR 172 subset. The JAX-RPC stubs are proxies created from a web service descriptor that represent the remote service and that hide the complexity of data marshalling and interfacing with the JAX-RPC runtime. The JAX-RPC runtime is responsible for managing the remote invocations and the network operations.

The **Service Provider** consists of the service classes that are accessible through a web server and a SOAP engine. There also is the Web Services Descriptor Language (WSDL) description, an XML document that describes the web service itself, the method signatures, and the binding information. The web server, for example the Apache HTTP Server, provides the web transport, and the SOAP engine, for example Apache's Axis, provides the JAX-RPC SOAP runtime for service invocation. The service ties are proxies for the actual service classes, and are responsible for handling details such as data marshalling. The service classes provide the actual service implementation [6].

The **Network** represents the Web, the transport (typically HTTP), and the SOAP-encoded messages.

II. Implementation Aspects

With the ever present concern over security, software applications must consider how to secure confidential data. A mobile application is not immune from privacy concerns. In fact, mobile devices and their software application have special considerations given that most people carry these devices wherever they go**.**

Bouncy Castle APIs - In order to encrypt sensitive data we used Bouncy Castle Cryptography APIs. Bouncy Castle is an open source Java API for encrypting and decrypting data. There is a lightweight package that is suitable for MIDP applications where only a fraction of the API will be used at any one time.

Obfuscation process - One problem inherent to most mobile devices is the limited amount of memory. As with most any library you use, only a small portion of the code is typically needed by your application. One common way to eliminate unused code, and at the same time make it more challenging to reverse engineer an application, is to use a Java obfuscator.

Reverse engineering of Java programs is not too difficult. As a matter of fact, there are free decompilers that will do the work for you. Ta make it a little more challenging to reverse engineer applications, many Java developers use an obfuscator to rename classes, methods, and fields. The intention of this renaming process is to make the source more unreadable.

A side effect of the obfuscation process is the reduction of class file size. This is accomplished in two steps. First, a lot of bytes can be saved by replacing names of classes, methods, and field names that are one or two characters in length. In addition, obfuscators will remove unused classes, methods, and fields. The combination of these two steps can significantly reduce the size of the final application. We used open source obfuscator ProGuard.

Security and Trust Services API (SATSA) - Security and Trust Services API is a new API that provides additional security capabilities to the J2ME CLDC platform. It specifies a collection of APIs that provide security and trust services for J2ME CLDC by integrating a Security Element (SE).

The SE is a hardware or a software component in a J2ME device. It provides the following features:

- Secure storage to protect sensitive data.
- Cryptographic operations.

With these features, J2ME applications would be able to have secure key stores as well as encryption and decryption capabilities. These features could be used to provide security services for applications such as e-payments, mobile commerce, etc. A SE can be: (1) deployed as a smart card in wireless phones (e.g. SIM PKI cards) or, (2) can be implemented by a handset itself (e.g., embedded chips or special security features of the hardware) or, (3) may be entirely implemented in software [7].

The support for cryptographic smart cards is of particular interest to developers writing J2ME applications for smart phones. Keys and certificates can be stored on the smart card and data can be signed without the private key ever leaving the card. High-end smart cards are temper resistant and provide authentication schemes, such as requiring a PIN or a password before access to the smart card is granted. This way security is dependent on the smart card not being compromised. Private keys do not have to be stored on diverse insecure clients, enabling vendors to focus on keeping the smart card secure from physical tempering and, just as important, smart card API exploitation.

SATSA APIs - The SATSA specification defines for APIs, SATSA-APDU, SATSA-JCRMI, SATSA-PKI, and SATSA-CRYPTO. The first two APIs add functionality for smart card interaction. SATSA-APDU enables communication with smart cards using the Application Protocol Data Unit (APDU) protocol defined by the

ISO7816-4 specification. SATSA-JCRMI enables high level communication with smart cards through the Java Card Remote Method Invocation Protocol (JCRMI).

SATSA-PKI enables applications to request digital signatures from an SE, hence providing authentication and possibly non-repudiation by using keys stored on a smart card. Client certificate management is also provided by SATSA-PKI, giving an application the opportunity to add or remove certificates from an SE. The most interesting part of the certificate management is the possibility to request generation of a new key-pair and then produce a Certificate Signing Request. The fact the client generates its own keys is one of the key factors needed to support non-repudiation in a system. Note that key generation is dependent on the SE, the SE might not support key generation at all. Hence, the SE must be chosen with care, considering the application requirements.

SATSA-CRYPTO offers cryptographic tools like message digests, digital signature verification, and ciphers. The API enables applications to store data encrypted and signed on a mobile device, ensuring both confidentially and integrity. Applications that require secure storage of highly sensitive information can therefore be realized. Note that it is up to the developer to decide which ciphers and digest algorithms to include. The SATSA specification recommends 3DES and AES as symmetric ciphers, RSA an asymmetric cipher, and SHA-1 as the digest algorithm. SHA1withRSA is the recommended algorithm for digital signatures [7].

III. A POSSIBLE WEB SERVICE SCENARIO

WSA uses the idea of stub classes, so other technology components such as cryptography, XML signature and Java Card technology have to fit into WSA stub classes [5]. The Java Card technology provides a means of securely storing confidential information, private and secret keys. You can use the Java Card technology for two basic purposes:

- to securely store a cryptographic key;
- to implement signature calculation algorithm.

Suppose User_A is a Java Card user, who wants to produce a cryptographic signature using the Java Card. He will need to provide a username and password while accessing his Java Card. The username-password pair will work only User_A's Java Card. Therefore, if a hacker steals his Java Card, the hacker will also need to know the username-password pair for that specific Java Card. This type of security is sometimes called dual factory security.

In order to communicate with a Java Card, a J2ME device will need an API called Security and Trust Services API (SATSA). Another important technology for developing secure web service is XML signature. You can use XML signature to wrap a cryptographic signature within an XML message.

A possible secure mobile web service scenario include following steps to WSA security (see Fig. 4):

1. Suppose User_A, a municipal resident, wants to access the municipal document exchange service to request for issuance of residence certificate. User_A's J2ME cell phone has a secure Municipal MIDlet running. So he will invoke the MIDlet.
2. The MIDlet hands over the request to a set of enhanced stub classes that use four technologies: XML signature, Cryptography, SATSA/Java Card, and WSA to author a secure reserve SOAP request that wraps user authentication data.
3. Enhanced stub classes use cryptography and SATSA to fetch all cryptographic support required by the secure SOAP request.
4. SATSA, in turn uses a Java Card application to compute cryptographic signature value over User_A's SOAP request.
5. Next, enhanced stub classes use the XML signature support to author a complete XML signature and wrap the signature in the SOAP request.
6. Enhanced stub classes use WSA framework to send the request over to the remote Web service endpoint implementation.
7. The remote Web service implementation will need to transform the incoming SOAP request before processing. A transformation module hosted in the remote Web service will do the job.

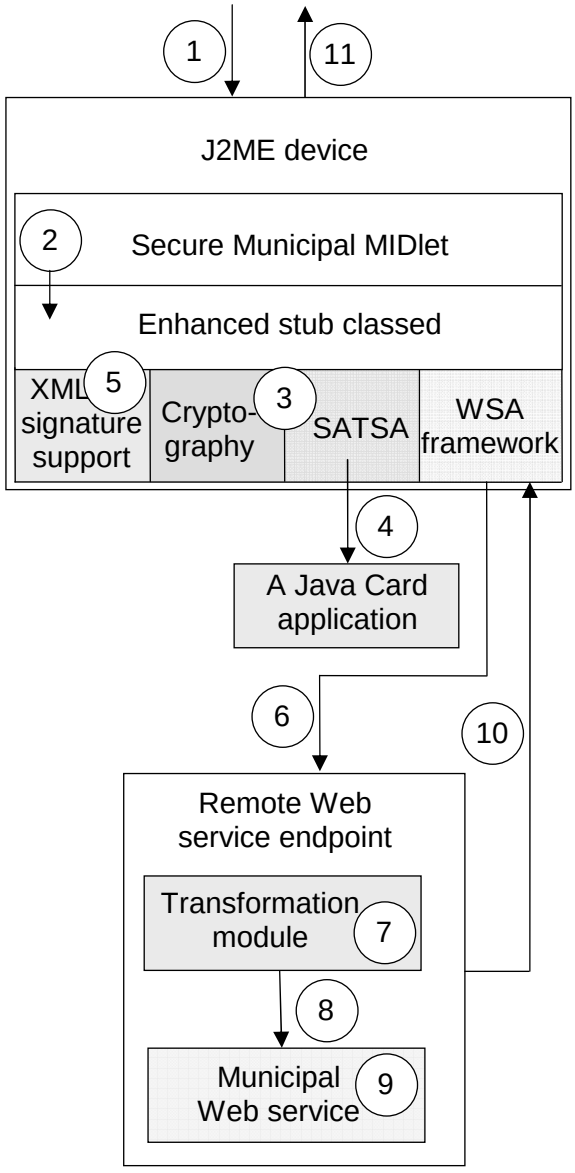

Figure 4. Security components for wireless access to Web services

8. The transformation module will hand over the request to the actual Web service implementation.
9. The Web service will perform request processing.
10. The web service will send certification document issuance status in the form of a SOAP response back to the WSA framework.
11. The WSA will process the SOAP response, extract certification status and display the same to the User_A.

IV. CONCLUSION

Web services are a good way to allow smaller devices and applications to use the processing power available on larger machine. Java 2 Micro Edition (J2ME) is a runtime environment for resource-constrained environments. The Java Specification Request 172 (JSR 172) specifies standardized client-side technology to enable J2ME applications to consume remote services. WSA uses the idea of stub classes, so other technology components such as cryptography, XML signature and Java Card technology have to fit into WSA stub classes

In this work, a possible secure mobile web service scenario is considered. In the scenario we use the idea of stub classes, so other technology components such as cryptography, XML signature and Java Card technology have to fit into WSA stub classes. We considered the scenario where private asymmetric keys and digital certificates stored on the smart card and data can be signed without the private key ever leaving the card.

ACKNOWLEDGEMENT

This work is being carried out in the context of the IST international cooperation project SWEB (044979). This paper is based on the work performed within the context of this project and the authors would like to acknowledge all SWEB partners.

DISCLAIMER

REFERENCES

[1] SWEB Project Homepage, http://www.sweb-project.org

[2] O. Kolsi, T. Virtanen, "MIDP 2.0 Security Enhancements", Proceedings of the 37th Hawaii International Conference on System Sciences, 2004.

[3] C. E. Ortiz, "Introduction to J2ME Web Services", April 2004, http://developers.sun.com/techtopics/mobility/apis/articles/wsa/.

[4] IBM Workplace Client Technology Micro Edition Version 5.7.1: Application Development and Case Study, Redbook, June 2005, sg246496.pdf, www.ibm.com/redbooks.

[5] B. Siddiqui, "Building a secure SOAP client for J2ME, Part 1: Exploring Web Services APIs (WSA) for J2ME", 16 Jun 2006, http://www-128.ibm.com/developerworks/edu/

[6] C. E. Ortiz, "Understanding the Web Services Subset API for Java ME", March 2006, http://developers.sun.com/techtopics/mobility/midp/articles/webservices/.

[7] MIDP 2.0: SATSA-APDU API Developer's Guide, version 1.0, February 2nd, 2007. Forum Nokia, *Handbook.* Mill Valley, CA: University Science, 2007.