Java-Based Wireless Identity Module

Thomas Weigold

University of Westminster, London, UK; IBM Research Laboratory, Zürich, Switzerland

(twe@zurich.ibm.com)

Abstract: Many of today's wireless devices apply the Wireless Application Protocol (WAP). To fully implement the security features that WAP defines, a Wireless Identity Module (WIM) is required within the client device. This paper describes a prototype WIM implementation running on a Java-based smart card and presents an evaluation of this WIM application focusing on its resource consumption and the execution speed of the services it provides. The results show that the Java-based WIM performs well for both criteria and can have a positive impact on the transport-layer and application-layer security provided through a WAP device.

1 Introduction

In recent years, wireless networks, in particular digital mobile-phone networks, have spread so rapidly, that for people today, carrying a mobile phone has become common place. The fact that these networks not only offered voice but also data communications, together with the growing ubiquity of the Internet lead to the birth of wireless Internet technology. When Internet technology was first applied to wireless devices, it became apparent that the very limited resources on client devices were going to be a problem. This gave rise to the development of the Wireless Application Protocol (WAP) [1], which takes into account mobile-device constraints such as limited CPU power, memory, bandwidth, battery life and simple user interfaces. WAP defines its own protocol stack, which is similar to the Internet protocol stack. However, each layer is optimised and where necessary modified for wireless devices. Figure 1 shows the WAP protocol stack in comparison to its counterpart in the "wired Internet". Most of today's mobile phones implement the WAP specifications or at least the mandatory parts thereof.

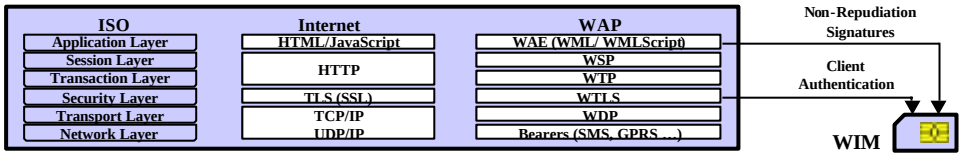

Figure 1: Internet vs. WAP protocol stack

1.1 WAP Security

The security features included in the WAP architecture are also derived from those used in the Internet and thus again are very similar. WAP defines a Wireless Public Key Infrastructure (WPKI) [2], which offers transport-layer security as well as application-layer security. The former is implemented by the Wireless Transport Layer Security (WTLS) [3] protocol as shown in Figure 1. Application-layer security is accessible via the WMLScript language implemented in WAP browsers. It includes the function "signText" that allows the creation of a digital signature of the text necessary to confirm an application-related transaction to the server. Typically, the text is presented to the user, the user then has to enter a PIN code to allow signature generation and finally the signature is sent to the server as proof for non-repudiation.

According to the level of security provided, WTLS instances are classified as being of class 1, 2 or 3 as follows:

- **Class 1** - Provides confidentiality and data integrity based on public-key cryptography between client and server. The two parties remain anonymous.
- **Class 2** - Additionally introduces server certificates to allow the client to authenticate the server.
- **Class 3** - Additionally introduces client certificates so that the WTLS session can be mutually authenticated and application-layer signatures can be generated as proof for non-repudiation.

Most of today's WAP services use WTLS class 1 or class 2, if security is applied at all. However, applications requiring the highest level of security need a corresponding class 3 infrastructure. In a WTLS class 3 scenario some parts of the security functionality on the client need to be performed inside a tamper-resistant device so that an attacker cannot retrieve sensitive data, in particular, the client's private key and thus its wireless identity. The same is true if application-layer signatures are to be supported by the WMLScript interpreter. To cater for this the WAP specifications define the Wireless Identity Module (WIM) [4]. The WIM provides secure storage as well as processing capabilities and thus allows the security sensitive parts of WTLS or application-layer protocols to be executed by the WIM rather than by the client device. Ideally, the WIM functionality is

implemented as an application on a microprocessor-based smart card since it provides a high level of tamper-resistance. Figure 1 illustrates the relation between the WAP protocol stack and the WIM.

This paper describes a prototype WIM implementation running on a Java-based smart card. Furthermore, it presents an evaluation of this WIM focusing on its resource consumption and on the execution speed of the services provided.

2 WIM Implementation

2.1 The Base Platform

The base platform for the prototype implementation of the WIM is the IBM JCOP SIM operating system running on the Philips P8WE5033 smart card microcontroller. JCOP is an open-standards-based Java operating system for smart cards implementing the Java Card 2.1.1 [5] and the Open Platform 2.0.1' [6] standards. Furthermore, it includes the functionality required by a Subscriber Identity Module (SIM) [7] used in Global System for Mobile Communications (GSM) [8] mobile-phone networks. JCOP represents the Java execution platform that the WIM instance is built upon.

The Philips chip can be considered a standard smart card microcontoller with the following basic characteristics: 8-bit/8051 CPU core operated at 8 MHz (internal clock speed); 96 Kbytes ROM; 32 Kbytes EEPROM; 2 Kbytes RAM; cryptographic coprocessor for Triple-DES; cryptographic coprocessor for large integer modular arithmetic; true random number generator; serial I/O interface (9.6-115 Kbits/sec).

2.2 WIM Features

The prototype WIM implementation provides all mandatory functionality defined in [4], supporting 1024-bit RSA operations (encrypt, decrypt, sign, verify) as well as the WTLS pseudo-random function based on SHA-1. As an additional feature, going beyond the current standard, it provides on-card RSA key generation. Table 1 lists all of the services offered by the WIM instance. The functionality is implemented using the Java API provided by the operating system. No additional native code has been introduced so that the WIM prototype can be considered a pure Java application.

3 Evaluation

3.1 Test Environment and Evaluation Criteria

The WIM instance was evaluated focusing on two main criteria. Firstly, the memory consumption of the WIM application, which includes its code size as well as the RAM and EEPROM space required for data. Secondly, the execution speed of the service primitives provided by the WIM and the implications to the WAP security services (WTLS/application-layer signatures) provided by the client device.

To determine the memory consumption of the application the source code and the executable code were examined and the JCOP development tools where used to trace runtime memory usage. To measure the execution speed the test environment shown in Figure 2 was set up. WAP simulation software was developed that communicates with the WIM application on the smart card just as a real WAP device would. The simulation software was running on a PC with a smart card reader attached. Additionally, a smart-card line-analyser was utilized to trace the communication protocol between the reader and the card. This allowed the precise measurement of application response times.

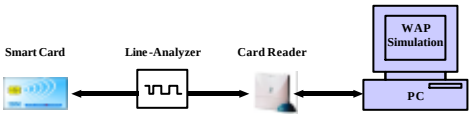

Figure 2: Test environment

3.2 Results

The memory consumption of the WIM application is as follows:

Application code size: 2779 bytes
This is the number of Java byte codes stored on the card. The WIM application was stored in ROM as part of the operating system.

RAM usage: 29 bytes
This does not include the usage of the communication buffer to store temporary data.

EEPROM usage: 419 bytes
This figure represents the EEPROM space used for application internal data structures such as PINs and the WIM file system structure. In particular, public keys used for signature verification and key transport are temporarily buffered in EEPROM due to the RAM limitations of the hardware.

EEPROM space required to store user data (e.g. files for storing certificates, private keys, WTLS session related information) depends on personalization settings and is therefore not considered here.

The execution speed of the WIM service primitives, broken down to WIM Application Protocol Data Unit (APDU) commands as defined in [4], is shown in the table below.

Service Primitive WIM COMMAND	Comments	Execution Time[1] [milliseconds]
WIM-OpenService		
MANAGE CHANNEL Open		5.46
SELECT Application		6.62
WIM-CloseService		
MANAGE CHANNEL Close		4.78
WIM-PerformVerification		
VERIFY	PIN-G or PIN-NR	27.55
WIM-DisableVerificationRequirement		
DISABLE VERIFICATION REQURIEMENT	PIN-G	38.37
WIM-EnableVerificationRequirement		
ENABLE VERIFICATION REQURIEMENT	PIN-G	38.35
WIM-ChangeReferenceData		
CHANGE REFERENCE DATA	PIN-G or PIN-NR	50.57
WIM-UnblockReferenceData		
RESET RETRY COUNTER	PIN-G or PIN-NR	60.86
WIM-OpenFile		
SELECT FILE	Non-varying for all WIM files	3.17
WIM-ReadBinary		
READ BINARY	Reading 1 byte / reading 256 bytes	7.20 / 9.88
WIM-UpdateBinary		
UPDATE BINARY	Updating 1 byte / updating 255 bytes	15.21 / 31.71
WIM-ComputeDigitalSignature		
MSE Set	7 bytes input data (file path/key reference)	4.05
PSO Compute Digital Signature	RSA[2] PKCS#1 signature; input: 20 bytes	159.20
WIM-VerifySignature		
MSE Set	set RSA[2] public key + 20 bytes SHA-1 hash	15.67
PSO Verify Digital Signature	Input: RSA[2] PKCS#1 signature	81.23
WIM-GetRandom		
ASK RANDOM	Generate 12 bytes of true random	20.25
WIM-KeyTransport		
MSE Set	Set RSA[2] public key + generate random	42.20
PSO Encipher (Key Transport)	RSA[2] PKCS#1 encryption	173.07
WIM-DeriveMasterSecret		
MSE Derive Key	Input: 45 bytes seed	178.24
WIM-PHash		
MSE Set	3 bytes input data (result length value)	2.99
PSO Compute Cryptographic Checksum	Input: 35 bytes seed; output 12 bytes result	131.68
	Input: 50 bytes seed; output: 44 bytes result	231.10
WIM-Decipher		
MSE Set	7 bytes input data (file path/key reference)	4.05
PSO Decipher	RSA[2] PKCS#1 decryption, output: 16 bytes plaintext	203.27
MSE Restore	WTLS_RSA_SE / WIM_GENERIC_RSA_SE	6.23
(Proprietary Extension)		
GENERATE PUBLIC KEY PAIR	RSA[2] key pair (averaged over 50 iterations)	4528.00

Table 1: Performance of WIM services

[1] Execution times were measured using the line-analyser and encompass the time between the transmission of the last bit of the command APDU sent to the card and the transmission of the first bit of the response APDU returned by the card. Therefore, the figures represent the execution speed excluding communication overhead. The communication overhead is dependent on the data transfer rate used. Modern GSM handsets, for instance, can support a data transfer rate of up to 115 Kbits/sec when communicating with the SIM/WIM.

[2] Modulus length: 1024 bits; private keys in CRT format; public exponent: 65537.

Based on the figures in Table 1 the overall time necessary to execute the set of commands required for a complete WTLS RSA handshake as described in [4] (section 11.4.4) is 1411.68 milliseconds. Similarly,

generating an application-layer signature as described in [4] (section 11.4.6) takes 197.03 milliseconds and application-related deciphering (section 11.4.7) 213.55 milliseconds. Unfortunately, no publications where found that included similar measurements, which would allow for comparison.

4 Conclusions

Security for wireless devices is a topic of increasing importance. In particular, applications requiring the highest level of security such as mobile financial transactions depend on the security infrastructure available [9]. WAP security including the WIM is one approach to provide this infrastructure. However, what seems to be missing is a "reality-check" that gives an example of what a Java-based WIM implementation can achieve. This paper provides such an example. While the performance obviously depends on the underlying operating system and the hardware, the figures presented provide a good reference for people planning to build solutions incorporating a WIM.

The memory consumption evaluation of the WIM implementation has shown that it is feasible to implement the WIM in Java on today's smart card microcontrollers and thus to take advantage of the platform independence a Java approach offers. Furthermore, since the hardware used is not considered "high-end" but a "mid-range" smart card chip similar to many of the SIM cards deployed today, it shows that introducing WIM functionality on SIM cards does not imply that more expensive hardware must be used. The operating system used to build the WIM prototype has a size of 48 Kbytes, thus the introduction of WIM functionality as a built-in feature increased the code size by less than 6%. The only problem identified was the limited RAM resources, which forced the usage of some EEPROM space to store temporary data.

The minimal memory consumption and good execution speed has justified the decision to implement the WIM in Java. Considering that many mobile devices have limited processing power and no hardware support for cryptographic operations, the fact that a WIM only needs about 1.4 seconds to do the crypto operations required for a complete WTLS class 3 handshake becomes of great value. In contrast, it has been shown that some software implementations of the RSA algorithm running on PDAs can take up to one minute to calculate a 1024-bit RSA signature [10]. On a typical mobile-phone CPU it would be significantly slower. This shows that using the WIM not only makes sense for protecting people's wireless identity against attacks but also for accelerating cryptographic operations by off-loading them to the smart card. Depending on the WAP device, it may already be worth taking advantage of the WIM to execute the two public key operations performed during a WTLS class 2 handshake.

At the time of writing, only a few WAP implementations support WTLS class 3 and the WMLScript crypto library. However, as security requirements for wireless transactions increase, these WAP features will certainly become more common. The WIM evaluation presented in this paper shows that availability and performance of Java-based SIM/WIM cards should not be the bottleneck in these developments.

5 References

[1] Wireless Application Protocol Forum, Ltd. 2001. Wireless Application Protocol, Architecture Specification, Version 12-July-2001, WAP-210-WAPArch-20010712. Available: http://www.wapforum.org/.

[2] Wireless Application Protocol Forum, Ltd. 2001. Wireless Application Protocol, Public Key Infrastructure Definition, Version 24-April-2001, WAP-217-WPKI. Available: http://www.wapforum.org/.

[3] Wireless Application Protocol Forum, Ltd. 2001. Wireless Application Protocol, Wireless Transport Layer Security, Version 06-April-2001, WAP-261-WTLS-20010406-a. Available: http://www.wapforum.org/.

[4] Wireless Application Protocol Forum, Ltd. 2001. Wireless Identity Module, Part: Security, Version 12-July-2001, Wireless Application Protocol WAP-260-WIM-20010712-a. Available: http://www.wapforum.org/.

[5] Sun Microsytems, Inc. 2000. JavaCard 2.1.1 API Specification, Runtime Environment (JCRE) Specification and Virtual Machine Specification.

[6] Visa International Service Association 1999. Open Platform Card Specification Version 2.0.1', 7 April 2000.

[7] European Telecommunications Standards Institute (ETSI) 2000. Digital cellular telecommunications system (Phase 2+); Specification of the Subscriber Identity Module – Mobile Equipment (SIM-ME) interface (GSM 11.11 version 8.3.0 Release 1999). Available: http://www.etsi.org/.

[8] Lamb Elliot. GSM – An Introduction. International Card Manufacturers Association (ICMA) article, January 1998. Available: http://www.icma.com/.

[9] Secure Identity in Mobile Financial Transactions, Nokia White Paper, January 2001, Available: http://www.nokia.com/.

[10] Vipul Gupta and Sumit Gupta, KSSL: Experiments in Wireless Internet Security, November 2001, SML Technical Report Series, Sun Microsystems, Available: http://research.sun.com/techrep/.