Challenges in Securing Networked J2ME Applications

An increasing number of smart phones support Java 2, Micro Edition. Mobile application developers must deal with J2ME's inherent security weaknesses as well as bugs in implementations on real devices. The new Security and Trust Services API for J2ME addresses some of these challenges, although it too has shortcomings.

André N. Klingsheim, Vebjørn Moen, and Kjell J. Hole
University of Bergen

A smart phone combines a full-featured mobile phone with advanced data and multimedia functionality including PDA capabilities, Internet and e-mail access, and MP3 and video playback. It typically includes a large color touch screen, a keyboard, Bluetooth technology to communicate with other devices, and substantially more memory and processing power than a regular mobile phone. Another key feature is the ability to install additional applications.

The smart phone market is growing fast,[1] spurring development of new mobile software for everything from gaming to online banking to GPS navigation. For example, total global revenue in the mobile gaming market is expected to soar from $2.6 billion in 2005 to $11.2 billion by 2010, with online multiplayer games generating 20.5 percent of market share.[2]

Many different development platforms exist for smart phones, categorized by phone manufacturers, mobile operating systems, and device capabilities. The most widespread is Java 2, Micro Edition (http:// java.sun.com/j2me), available on nearly 80 percent of currently marketed smart phones.

Experience gained during a commercial development project demonstrates how J2ME technologies, particularly security-related functionality, are implemented on real devices and provides insights into the problems researchers encounter during the development process.

J2ME TECHNOLOGIES

The Java 2 platform has several editions, including Enterprise Edition (J2EE) for the server side and Standard Edition (J2SE) for desktop systems. J2ME is a highly optimized Java runtime environment aimed at mobile phones, PDAs, and other small devices.

Configurations and profiles

J2ME *configurations* are intended for devices with similar characteristics in terms of processors and memory. *Profiles* target devices that are similar in terms of screen type, input devices, and network connectivity; they complement the low-level functionality of configurations by adding support for user interaction and network connectivity.

Published by the IEEE Computer Society
0018-9162/07/$25.00 © 2007 IEEE

A configuration specifies the supported Java virtual machine features, the included Java programming language features, and the supported Java libraries and application programming interfaces (APIs). Two configurations are widely available on J2ME devices: Connected Limited Device Configuration 1.0, which has been around for years, and CLDC 1.1, which is deployed in newly released devices (http://java.sun.com/products/cldc). The most notable difference between the two configurations is that CLDC 1.1 adds support for floating-point operations. Because CLDC 1.1 is a superset of CLDC 1.0, Java applications built for CLDC 1.0 will run without problems on CLDC 1.1. Thus, unless any of the additions in version 1.1 is specifically needed, developers should build their applications for CLDC 1.0 to be compatible with as many devices as possible.

Two profiles extend CLDC functionality: Mobile Information Device Profile 1.0 and 2.0 (http://java.sun.com/products/midp). The MIDP profiles add APIs for user interaction, network connectivity, and persistent storage. HTTP connections provide network connectivity, while a record management system provides persistent storage. MIDP 2.0, a superset of MIDP 1.0, includes support for secure HTTP (HTTPS) connections and more powerful graphics APIs for gaming.

Many developers fail to recognize that J2ME devices can behave inconsistently.

Specifications

New J2EE/J2SE/J2ME specifications emerge through the Java Community Process (JCP) program. Developers first submit a Java Specification Request for acceptance by an executive committee. Once the committee accepts a JSR, an expert group assumes responsibility for specification development. The specification draft must first pass a community review, then a public review. The expert group presents a proposed final draft, and an approval ballot decides if the specification is suitable for release.

An expert group typically consists of many participants. For example, the MIDP 2.0 expert group included, among others, Ericsson, Motorola, Nokia, Siemens, Sun Microsystems, Samsung, and Symbian. Expert groups enable large industrial parties to collaborate in specifying new functionality for the Java platform. This minimizes competition among different vendor specifications and increases the likelihood of a specification's wide adoption. All CLDC versions and MIDP versions resulted from JSRs, as did several other J2ME components such as the Security and Trust Services API (JSR-177), the Wireless Messaging API (JSR-120), and the Java APIs for Bluetooth Wireless Technology (JSR-82). All specifications are freely available at the JCP Web site (www.jcp.org).

CURRENT CHALLENGES

J2ME developers face several challenges. J2ME-enabled smart phones generally have some software quality issues, and developers are likely to spend time solving problems resulting from the varying quality of J2ME implementations. Specific phone models can have their own bugs, forcing developers to maintain several parallel versions of their source code to support as many devices as possible.

This situation is inconsistent with the Java philosophy of "write once, run anywhere," and it severely increases the complexity of mobile software development and maintenance. In our experience, MIDP 2.0 implementations have fewer bugs than MIDP 1.0 implementations. However, to cover as much of the market as possible, developers must consider all existing MIDP 1.0 devices. MIDP 2.0 devices still constitute a minority of all J2ME devices sold in recent years.

Insufficient testing

Many developers fail to recognize that J2ME devices can behave inconsistently. Testing J2ME applications on many different devices is critical to ensuring that security-related functionality meets expectations. Businesses that base their testing on only four or five devices are often surprised when their application does not run correctly on other devices. In our opinion, too many businesses neglect the testing phase and let their customers do the beta testing. For such a strategy to succeed, the application must be unique and customers must be both enthusiastic and understanding.

To truly appreciate how J2ME phones operate, developers should test several devices from each major vendor, with each version of a vendor's development platform represented in the set of test devices. Of course, testing 40 to 50 or more devices is expensive. Businesses thus must carefully balance the cost of testing with the risk of releasing an application that might not function properly on some devices.

Permanent bugs

Desktop users commonly download patches from Microsoft Windows Update or similar Web sites to solve security issues and fix bugs in software. Mobile phone vendors also release new software versions, but these generally do not reach consumers who have already bought the device. In most cases, users must take their mobile phone to a repair shop to have a software upgrade performed. Since few people actually do this, the bugs on a new mobile phone usually stay there for the device's lifetime. However, a user who buys the same phone a year later will probably get a newer software version.

With mobile phone viruses starting to appear, there is a growing need for a patching solution that lets consumers upgrade the phone software themselves, similar to what they hopefully do with their desktop systems.

Resource management

Although smart phones have more memory, processing power, network bandwidth, and disk space than standard mobile phones, they are still a resource-constrained platform compared to the desktop. In addition to traditional functionality, developers must thus consider effective resource utilization to make user-friendly mobile applications.

Defensive programming is the key to creating a well-functioning application. For example, a program can query available runtime memory and storage memory while executing to ensure that the device has enough memory to carry out the program's operations. Otherwise, if the device runs out of memory, it will show an error message and terminate the application, giving the user the impression that an error occurred when in fact the application did not adapt to the available resources.

Functionality-rich security libraries can be too complex to bundle with an application.

Scarce resources limit developers' options, as functionality-rich security libraries can be too complex to bundle with an application. After all, if a J2ME implementation had the capacity to store and run such a library, it could have included one in the first place. The use of these libraries is not impossible, but older devices might not have the capacity to install and run the application.

Responsive applications

Applications must be responsive to provide a positive user experience. To achieve this goal, intimate knowledge of J2ME devices' inner workings is helpful, since mobile phones behave differently. One example is actively triggering garbage collection to reclaim memory from unused objects. In most cases, running a garbage collector in the background will minimally impact the application, but some devices will freeze for a few seconds while collecting memory, which is probably not what the developer wants or expects.

Multithreading is also important when developing applications for mobile phones. A program's execution flow is event-driven, so the main thread must be idle and ready to handle events. Time-consuming operations such as lengthy calculations or network communication should thus occur in a separate thread to retain a responsive user interface. This should be familiar to those developing graphical user interface applications on the desktop, as the same UI versus non-UI thread considerations apply.

MIDP 2.0 SECURITY FRAMEWORK

MIDP 2.0 includes several mechanisms to secure applications and communication channels. Applications can be signed to obtain authenticity and integrity, while HTTPS connections realize secure communication channels, which in most cases rely on the Secure Sockets Layer (SSL) or Transport Layer Security (TLS) protocol. In addition, MIDP 2.0 tries to ensure that an application cannot read other J2ME applications' persistent data without explicit permission.

However, studying MIDP 2.0's details reveals that some critical security functionality is actually optional on J2ME devices or can be based on insecure mechanisms. Developers will also be disappointed to realize that, when real testing begins, some smart phones do not correctly implement mandatory security functionality.

Application signing

X.509 public-key infrastructure (PKI)-based signature verification,[3] an optional feature in the MIDP 2.0 specification, lets devices verify a signed J2ME application's origin and integrity. Signed applications are generally desirable for m-commerce and mobile government services, but when users install a signed application on devices that do not verify signatures, parts of the security architecture crumble.

In fact, two newly released Samsung smart phones we tested during our project refused to install J2ME applications signed with the private key belonging to a code-signing certificate obtained from Verisign. However, the Nokia devices we tested validated the certificate and applications without problems.

To support all devices, developers must provide both signed and unsigned versions of applications to customers, effectively making the security architecture's signed application feature optional.

Certificate management and verification

All certificate verification procedures on a mobile phone rely on a set of preinstalled root certificates (belonging to certificate authorities), equivalent to what is present in Web browsers. Of course, different mobile phones have different root certificates installed.

Additional root certificates can be installed on smart phones, which is very useful during a test phase. We successfully created and installed a self-signed X.509v3 certificate on the Nokia 6600 by publishing it to a Web server and then downloading it to the phone via the Wireless Application Protocol (WAP). A certain level of user interaction was needed after installing the certificate, as all certificates in this device have properties describing their area of use. A user-installed certificate

must thus be enabled for verification of signed applications or server authentication before it can be used.

An important requirement when working with certificates is the ability to validate a certificate. Time-limited validity is one mechanism, but support for certificate revocation is more critical. "Certificate revocation can be performed if the appropriate mechanism is implemented on the device," MIDP 2.0 states. "Such mechanisms are not part of MIDP implementation and thus do not form a part of the MIDP 2.0 security framework." Consequently, a certificate's validity must be based on the assumption that it has not been revoked.

During our project, we observed that the Samsung SGH-E720 had preinstalled Verisign class 1, 2, 3, and 4 root certificates reported valid from 1 October 1999 to 1 January 1970. (Other preinstalled certificates showed sensible validity intervals.) The same Verisign root certificates on the Nokia 6600 all had a 1 October 2049 expiration date. Giving Samsung the benefit of the doubt, we assume that the invalid date is not the value actually stored in the certificate. However, because the date is not presented correctly, the expiration date might not be interpreted correctly during the certificate validation process. Unfortunately, we could not verify this, as finding a Web site that uses a certificate signed by a specific root certificate is not a trivial task.

The lack of sources of randomness in J2ME implementations is an important issue.

Secure communication

Secure connections in MIDP 2.0 must be implemented by one or more of the following specifications:

- HTTP over TLS (RFC 2818) with TLS v1.0 (RFC 2246),
- SSL v3.0,
- Wireless Transport Layer Security (WTLS), or
- WAP TLS Profile and Tunneling Specification.

A developer cannot know which specifications are implemented on a specific device without actually testing it. In the case of WTLS, end-to-end encryption is not provided. Secure connections will then exist between the phone and the WAP gateway and from the gateway to the final destination. Thus, the gateway has access to unencrypted data and must be fully trusted. This is unacceptable for a high-security application, as the mobile network provider usually operates the gateway.

In addition, secure connections in MIDP 2.0 require the server to have a valid certificate for authentication. However, there is no support for certificate-based authentication of the client, so the client must be authenticated at the application level by other means.

Secure storage

MIDP 2.0 does not support encrypted storage, making it vulnerable to hardware attacks. On some devices it is possible to install a file system explorer, locate the files that J2ME uses, and send those files to another device via Bluetooth technology. Thus, J2ME's storage system cannot be trusted with sensitive data unless the application itself secures the data. Cryptographic libraries for Java might be a partial solution—for example, Bouncy Castle's lightweight cryptography API supports several symmetric ciphers. However, the availability of cryptographic APIs does not automatically solve the plaintext storage problem.

Encryption is a computation-intensive task, and when implemented in Java it can be time-consuming because low-level optimizations such as those for CPU architecture are impossible. Native libraries or cryptographic hardware are likely to perform encryption more efficiently.

Another important issue is the lack of sources of randomness in J2ME implementations. A strong cipher can be used, but the encryption key must be impossible to guess. MIDP 2.0 does not provide a cryptographically strong pseudorandom number generator similar to SecureRandom in J2SE. Thus, the PRNG in J2ME is not suitable for generating encryption keys. Nevertheless, developers use it for other purposes, and this can have a major impact on a system's security, especially since they tend to seed the PRNG with the current time.[4] One example is an attack on SSL in Sun's MIDP reference implementation.[5]

Because MIDP 2.0 does not provide encrypted storage and MIDP 1.0 does not provide HTTPS links, some J2ME development companies specify their own lightweight cryptographic schemes. In most cases, these well-meaning efforts to create new cryptographic algorithms result in solutions that keep data hidden from average users but are rarely cryptographically strong. Such initiatives usually rely on the secrecy of the encryption algorithm, which is considered very bad practice.[4]

Many countries have laws regulating the protection of private data during transportation over a medium that the two communicating parties do not control. These laws often require encrypting the data with an algorithm of strength equal to or better than the Triple Data Encryption Standard; its successor, the Advanced Encryption Standard, also fulfills this requirement. However, homemade cryptographic solutions almost certainly lack the strength of well-tested encryption algorithms such as 3DES or AES.

USING CLIENTS TO ATTACK THE SERVER

Beyond the well-known problem of Internet server

attacks from viruses and worms and distributed denial-of-service attacks, developers must consider the many ways in which client applications can attack the server application. Two examples are clients sending commands out of order or sending unexpectedly large data chunks as input to an application.

Client software can find its way into the hands of all kinds of people, including those who cannot resist trying to break it. Hackers can reverse engineer software and study the source code or, short of that, simply tamper with the binary code. The gaming industry is experiencing the problem of games being cracked to avoid license key checks on a daily basis. Another strategy hackers employ is to use a network sniffer to identify the application protocol and then write their own malicious client that behaves similarly to the original one.

Developers can take several measures to increase the level of security in a client-server application, but in our experience, most place too much trust in their client software. Their arguments are along the lines of: "Hey, we wrote it. Why shouldn't we trust it?" In our opinion, this is a dangerous attitude.

Support for cryptographic smart cards is particularly useful to developers writing J2ME applications for smart phones.

FUTURE J2ME SECURITY: SATSA

The new Security and Trust Services API (http://java.sun.com/products/satsa) for J2ME addresses several shortcomings in the MIDP 2.0 security architecture. SATSA relies on a *security element* implemented in either software or hardware. This implies that the SE can take different forms such as a software component, dedicated hardware in the device, or a removable smart card. Several SEs can be available in one device. The exact form of the SE is transparent to the application developer, as the SATSA implementation handles interaction with the SE.

Support for cryptographic *smart cards* is particularly useful to developers writing J2ME applications for smart phones. A smart card can store keys and certificates and sign data without the private key ever leaving the card. High-end smart cards are tamper-resistant and provide authentication schemes, such as requiring a PIN or a password before granting access. Private keys need not be stored on diverse insecure clients, enabling vendors to focus on protecting the smart card from physical intrusion and, just as important, API exploitation.[6]

Many banks already issue smart cards that include a magnetic strip to be compatible with old ATMs. By giving customers smart cards with cryptographic tools, a bank could also provide client software for mobile phones, PDAs, and desktop computers, thereby extending nearly the same level of security for key storage on all platforms. Of course, different operating systems have different levels of security, so the bank would have to carefully analyze each platform to control smart card access.

SATSA APIs

The SATSA specification defines four APIs: APDU, JCRMI, PKI, and CRYPTO. The first two add functionality for smart card interaction. SATSA-APDU enables communication with smart cards using the Application Protocol Data Unit protocol that the ISO7816-4 specification defines, while SATSA-JCRMI enables high-level communication with smart cards through the Java Card Remote Method Invocation Protocol.

SATSA-PKI lets applications request digital signatures from an SE, thereby providing authentication and possibly nonrepudiation[3] using keys stored on a smart card. Certificate management gives applications the opportunity to add or remove client certificates from an SE. It also includes the possibility to request generation of a new key-pair and then produce a Certificate Signing Request. Having the client generate its own keys is a critical element in supporting nonrepudiation in a system. Because the SE might not support key generation, the developer must choose it with care, considering the application requirements.

SATSA-CRYPTO offers cryptographic tools including message digests, digital signature verification, and ciphers that let applications store data encrypted and signed on a mobile device, ensuring both confidentiality and integrity for sensitive information. It is up to the SATSA implementer to decide which ciphers and digest algorithms to include. The specification recommends DES, 3DES, and AES as symmetric ciphers, RSA as an asymmetric cipher, and the Secure Hash Algorithm version 1 as the digest algorithm. SHA1 with RSA is the recommended algorithm for digital signatures.

Operating system issues

SATSA is distributed as a part of the Java Runtime Environment in some smart phones. The phone's operating system must thus be fully trusted, as the JRE depends on services from the OS. The SATSA specification states that both SATSA and the application using it must trust the OS. When SATSA assumes UI control—for example, when asking the user for the smart card PIN—the UI must be distinguishable from one generated by external sources to prevent a malicious application from mimicking the SATSA UI. Further, external sources cannot be able to retrieve or insert the PIN. These requirements put a lot of responsibility on the OS.

Because a user enters the PIN on the device through the keypad, a keylogger application could possibly

obtain the number. Thus, the OS must limit the distribution of key-pressed events to the SATSA implementation exclusively. Keyloggers exist for Symbian OS versions prior to version 9, so this could be a real issue. Another potential problem is that the OS stores the PIN in memory before handing it over to the smart card; other applications could possibly read this memory.

J2ME applications use a per-application dedicated logical channel to communicate with the smart card. This channel might be vulnerable to hijacking via the OS's low-level functionality. A hacker who acquires access to the smart card on a level below the SATSA implementation could send requests to the smart card, circumventing the implementation. If this functionality was included in, for example, a Trojan horse, the attacker could have full access to the smart card without the user's knowledge.

An application cannot be more secure than the OS it runs on.

These scenarios illustrate problems with putting complete trust in the OS. In our opinion, an application cannot be more secure than the OS it runs on. It remains to be seen how the mobile platform copes with viral threats, as mobile phone viruses are only beginning to emerge.

The Trusted Computing Group has proposed an initiative to make OSs on mobile devices more secure (www.trustedcomputinggroup.org/groups/mobile). In the future, mobile devices are likely to become more trustworthy, facilitating development of secure applications.

SATSA shortcomings

Client certificates that the SATSA implementation stores cannot be used for authentication while setting up HTTPS links using SSL or TLS. Developers thus must handle certificate-based client authentication themselves.

One alternative is to open an SSL connection to a server, which authenticates the server and provides a secure communication channel. Client authentication can then be carried out using a certificate provided by the client application and having the client sign a challenge from the server.

This scheme withstands dictionary or brute-force attacks and is thus more robust than widely used password authentication. Consequently, SATSA can improve security in current client-server applications by replacing old-fashioned authentication schemes. However, it would be convenient if SSL or TLS implementations found in most newer smart phones could use the client certificates the SATSA implementation stores.

Signature verification with SATSA is more difficult than signature creation. SATSA generates signed messages on the Cryptographic Message Syntax format but does not easily validate them. To verify a signature, the application must first split a message into its respective data and signature parts and then supply a public key. Libraries exist for handling CMS messages, so developers need not implement CMS message parsing. However, it would be easier if SATSA could verify its own signed messages.

In addition, SATSA handles signature generation and verification differently. The underlying SATSA implementation takes control of the UI and presents the user with the certificate for signing along with the data to sign. The user can then be confident in signing the intended data. Signature verification is just as important, but in this case the application must present details about the signature to the user. In other words, you trust SATSA when signing data and the application to show you correct information when verifying signatures. Ideally, SATSA should be trusted on both occasions.

Moreover, SATSA does not support certificate verification. The developer must implement the certificate verification process and public-key extraction from the certificate, along with presenting the certificate and signed data to the user. SATSA provides the most basic building blocks for PKI-enabled applications, but it does not include any certificate validation functionality present in J2SE.

Finally, private keys stored on the smart card cannot be used for decryption, as they are accessible only for signing. SATSA supports asymmetric cryptography, and the ability to select a certificate and its corresponding private key for decryption would be convenient. Data could then be decrypted on the smart card, without exposing the private key to the application or OS.

Developing secure J2ME applications is difficult due to limitations in the J2ME security model as well as bugs in real devices. A proper security analysis and testing of real devices must be carried out to determine the level of security that can be achieved.

The SATSA specification adds cryptographic capabilities to the J2ME platform. With SATSA, symmetric encryption is possible, and authentication schemes can rely on public-key cryptography instead of the usual username and password authentication. User certificate storage and support for digital signatures in smart cards open up possibilities for applications that require a high level of security.

Although SATSA is a good fit for scenarios requiring strong client authentication and digital signatures on behalf of the client, it provides only the basic cryptographic building blocks for a PKI. Important functionality found in J2SE such as certificate parsing,

validation, and storage is left to the developer. Implementing signature validation and public-key cryptography based on public keys in certificates thus remain complex tasks.

Acknowledgment

We thank World Medical Center Norway for its cooperation during this project and for giving us access to the mobile phones needed to conduct the security-related tests.

References

1. Canalys.com, "Worldwide Smart Phone Market Soars in Q3," 25 Oct. 2005; www.canalys.com/pr/2006/r2006071.htm.
2. Telecoms.com, "Mobile Games Industry Worth USD $11.2 Billion by 2010," 19 May 2005; www.telecoms.com/itmgcontent/tcoms/search/articles/20017303052.html.
3. C. Adams and S. Lloyd, *Understanding PKI: Concepts, Standards, and Deployment Considerations,* Addison-Wesley Professional, 2nd ed., 2002.
4. J. Viega and G. McGraw, *Building Secure Software: How to Avoid Security Problems the Right Way,* Addison-Wesley Professional, 2001.
5. K.I.F. Simonsen, V. Moen, and K.J. Hole, "Attack on Sun's MIDP Reference Implementation of SSL," *Proc. 10th Nordic Workshop Secure IT-Systems,* 2005, pp. 96-103; www.nowires.org/Papers-PDF/MIDP-SSL.pdf.
6. R. Anderson et al., "Cryptographic Processors—A Survey," *Proc. IEEE,* vol. 94, no. 2, 2006, pp. 357-369.

André N. Klingsheim *is a PhD student in the Department of Informatics, University of Bergen, Norway. His research interests include network and application security. Klingsheim received an MSc in computer science from the University of Bergen. He is a member of the IEEE Computer Society. Contact him at klings@ii.uib.no.*

Vebjørn Moen *is a researcher in the Department of Informatics, University of Bergen. His research interests include network and application security. Moen received a PhD in computer science from the University of Bergen. Contact him at moen@ii.uib.no.*

Kjell J. Hole *is a professor in the Department of Informatics, University of Bergen. His research interests include network and application security. Hole received a PhD in computer science from the University of Bergen. He is a member of the IEEE Computer Society. Contact him at kjell.hole@ii.uib.no.*

Sign Up Today

For the IEEE Computer Society Digital Library E-Mail Newsletter

- Monthly updates highlight the latest additions to the digital library from all 23 peer-reviewed Computer Society periodicals.
- New links access recent Computer Society conference publications.
- Sponsors offer readers special deals on products and events.

Available for FREE to members, students, and computing professionals.

Visit **http://www.computer.org/services/csdl_subscribe**