JOURNAL OF OBJECT TECHNOLOGY

Online at http://www.jot.fm. Published by ETH Zurich, Chair of Software Engineering. ©JOT, 2005

Vol. 5, No. 2, March–April 2006

Security Evaluation of J2ME CLDC Embedded Java Platform *

Mourad Debbabi, Mohamed Saleh, Chamseddine Talhi and Sami Zhioua

Computer Security Laboratory
Concordia Institute for Information Systems Engineering
Concordia University

Java 2 Micro-Edition Connected Limited Device Configuration (J2ME CLDC) is the platform of choice when it comes to running mobile applications on resource-constrained devices (cell phones, set-top boxes, etc.). The large deployment of this platform makes it a target for security attacks. The intent of this paper is twofold: First, we study and evaluate the security model of J2ME CLDC. Second, we provide a vulnerability analysis of this Java platform. The evaluated components are: Virtual machine, CLDC API and MIDP (Mobile Information Device Profile) API. The analysis covers the specifications, the reference implementation (RI) as well as several other widely-deployed implementations of this platform. The aspects targeted by this security analysis encompass: Networking, record management system, virtual machine, multi-threading and digital rights management. This work identifies security weaknesses in J2ME CLDC that may represent sources of security exploits. Moreover, the results reported in this paper are valuable for any attempt to test or harden the security of this platform.

1 INTRODUCTION

With the proliferation of mobile, wireless and internet-enabled devices (e.g. PDAs, cell phones, pagers, etc.), Java is emerging as a standard execution environment due to its security, portability, mobility and network support features. The platform of choice in this setting is J2ME CLDC [19, 25]. It is an enabling technology for a plethora of services and applications: games, messaging, presence and availability, web-services, mobile commerce, etc.

This platform has been deployed now by more than 20 telecommunication operators. The total number of deployed Java mobile devices in the market exceeds 250 million units worldwide. According to IDC, a prestigious market research firm, there will be more than 1.2 billion deployed Java-based mobile devices by 2006.

J2ME CLDC gained a big momentum and is now standardized by the Java Community Process (JCP) and adopted by many standardization bodies such as

Cite this document as follows: M. Debbabi, M. Saleh, C. Talhi and S. Zhioua: *Security Evaluation of J2ME CLDC Embedded Java Platform* *, in Journal of Object Technology, vol. 5, no. 2, March–April 2006, pages 125–154,
http://www.jot.fm/issues/issues_2006_3/article2

3GPP, OMA, etc. Another factor that has amplified the wide industrial adoption of J2ME is the broad range of Java-based solutions that are available in the market. All these factors made Java in general and J2ME in particular an ideal solution for software development in the arena of embedded and wireless systems.

The typical most widely deployed J2ME CLDC platform consists of several components that can be classified into virtual machine, APIs and tools. The virtual machine is the KVM [20, 26]. The APIs are CLDC [34] and MIDP[23, 25]. The tools are the pre-verifier and the Java code compacter. KVM (Kilobyte Virtual Machine) is an implementation of the Java Virtual Machine (JVM) [10]. It lies on top of the host operating system and its main goal is to execute compiled program units (class files). CLDC provides the most basic set of libraries and virtual-machine features for resource-constrained, network-connected devices. MIDP is a layer on top of CLDC configuration. It extends the latter with more specific capabilities, namely, networking, graphics, security, application management, and persistent storage. The preverifier checks all the Java classes to enforce object, stack and control-flow safety. This is done off-line and the result is stored as attributes in the compiled program units. The Java code compactor (JCC) is in charge of the romizing process. The latter is a feature of KVM that allows to load and link Java classes at startup. The idea is to link these classes off-line, then create an image of these classes in a file and finally to link the image with KVM.

With the large number of applications that could be available for Java-enabled devices, security is of paramount importance. Applications can handle user-sensitive data such as phonebook data or bank account information. Moreover, Java-enabled devices support networking, which means that applications can also create network connections and send or receive data. Security in all of these cases should be a major concern. Malicious code has caused a lot of harm in the computer world, and with phones having the ability to download and run applications there is an actual risk of facing this same threat. Currently, viruses for phones start to emerge (e.g. Cabir), a number of model specific attacks has been reported (e.g Nokia 6210 DoS, Siemens S55 SMS, etc.), and mobile attacks and exploits are starting to get attention in the hacker community (e.g. www.defcon.org).

This paper aims to provide a careful study of the J2ME CLDC security. The purpose of this paper is twofold; (first) study and evaluate the security model of J2ME CLDC, (second) provide a vulnerability analysis of this Java platform. In this regard, we investigated both, the published specifications and the available implementations. By analyzing specifications, we provide a comprehensive study of the J2ME CLDC security model, pointing out possible shortcomings and aspects open for improvement. By investigating available implementations, we discovered existent weaknesses through security testing. To make the results of our analysis as credible as possible, we investigated several implementations of the platform like Sun's RI, phone emulators, and actual phones.

By identifying weaknesses that may represent sources of security breaches, our security evaluation provides a starting point to a process that aims to improve the security of J2ME CLDC, which is the final goal of this research work.

This paper is organized into six sections beginning with the introduction. In section 2, we present related work concerning the comprehension of J2ME CLDC security and its evaluation. In sections 3 and 4, we present the security architecture of J2ME CLDC, followed by our evaluation of the underlying security model. They are followed by our vulnerability analysis, which starts by presenting the previously reported flaws, and then giving the actual results of our analysis. Finally section 6 concludes the paper.

2 RELATED WORK

This section gives an overview of what has been published in the literature regarding J2ME CLDC security evaluation. The majority of these efforts provided tutorials on J2ME CLDC security and tried to evaluate the underlying security model. Typically, these papers contain a description of the security model for J2ME CLDC (MIDP 1.0 and/or MIDP 2.0) followed by an evaluation of that model. Some papers point out weaknesses while others provide ideas to improve the security mechanisms.

In [15], Kolsi and Virtanen, tried to evaluate the new security features of MIDP 2.0. They started by giving an exhaustive classification of all threats and security needs in mobile environment, in particular for J2ME. Based on that analysis, the authors emphasized the security weaknesses in MIDP 1.0. Then, the new security mechanisms and features of MIDP 2.0 are analyzed showing how the security problems of MIDP 1.0 are addressed. Finally, the authors concluded that several security problems were addressed by MIDP 2.0, but some problems are still present mainly related to the PKI. These problems are illustrated in [3].

In [1], Sanjay Chadha discusses various issues about J2ME applications. In particular, he discusses two security concerns. First, he points out that the J2ME built-in security is not enough because the application has to be delivered to the device before it is processed through the security verification process. He claims that this is not sufficient in the wireless world since it is already too late if a malign application is delivered to a mobile device. He thinks that applications need to be tested for viruses and other potential malign behavior on the server side before they are deployed to the mobile device. Second, he highlights the lack of DRM (Digital Rights Management) support in J2ME. For example, most users prefer to try an application before making a purchasing decision. Without such capability, users are reluctant to buy games, which leads to lost revenue for operators.

Also [37] presents security challenges and solutions for J2ME-based mobile commerce applications. Yuan and Long closely examine the potential security advantages of J2ME-based applications over other wireless alternatives such as WAP and native applications. Then, they explain the security model available on the J2ME platform (CLDC and CDC). As part of their discussion, Yuan and Long suggest some potential ways to enhance network and data security for J2ME applications.

For example, they claim that point-to-point protocols such as SSL are not suitable for web services and to address this issue there is a need to an end-to-end security model with flexible encryption schemes. They believe that the focus should be on securing content rather than connection. Also, they propose to secure content through securing XML which is their format of choice for data communication between J2ME wireless applications and back-end services. Finally, they summarize the feasibility of developing advanced secure applications for the smallest wireless devices using J2ME technologies. It should be noticed also that their article is published in 2002 and some issues discussed by the authors were addressed by new JSRs. For example, JSR 172 provides basic XML procession capabilities to J2ME.

It is important to note that there is a large number of efforts in the J2ME CLDC security literature that focus on supporting security at the application layer [9, 35, 11, 12, 13, 14]. These papers discuss security solutions that do not rely on protocols at lower layers. That is, the security related functions are implemented in the application (MIDlet) itself.

3 J2ME CLDC SECURITY ARCHITECTURE

The high-level J2ME CLDC architecture defines 3 layers on top of the device's operating system: The *virtual machine*(KVM) [17], the *Configuration* (CLDC) which is a minimal set of class libraries that provide the basic functionalities for a particular range of devices, and the *Profile* (MIDP) which is an extension of the *Configuration* that addresses the specific demands of a device family. At the implementation level, MIDP also consists of a set of Application Program Interfaces (APIs).

Applications developed for the J2ME CLDC platform (`MIDlets`) are downloaded to the device in the form of two files: the Java Archive (`JAR`), and the Java Application Descriptor (`JAD`). The JAR is an archive file that contains the JAR manifest, which is a text file that contains various attributes like the MIDlet name. It also contains *preverified* class files of the MIDlet, plus any other *Supporting files* needed by the application (e.g. graphic files). One JAR file can contain more than one MIDlet and the set of MIDlets in a JAR file is called `MIDlet suite`. The JAD on the other hand, is a plain text file that contains several attributes like the MIDlet name and MIDP version needed to run the MIDlet. The JAD is also used to give some information about the MIDlet, such as the vendor's name, a small description, etc. The software entity on the device that is responsible for MIDlet management is called the `Application Management System` (AMS), or the `Java Application Manager` (JAM).

On J2ME CLDC devices security issues are classified as:

- *Low-level* security deals with safety issues related to the virtual machine. In general, the role of the low-level security mechanisms is to ensure that class files loaded into the virtual machine do not execute in any way that is not allowed by the Java virtual machine specification [17].

- By *application-level* security, we mean that "Java applications can access only those libraries, system resources and other components that the device and the Java application environment allow it to access" [34].

- *End-to-end* security has a larger scope involving secure networking. The main objective of *end-to-end* security is to ensure safe delivery of data and code between server machines and client devices.

Low-level and application security are addressed in CLDC, while MIDP addresses application and end-to-end security.

CLDC Security

To understand the security model of CLDC, it is important to notice that the security of CLDC is affected by the absence of some general Java features - that are usually present in Java platforms - and that have been dropped because of performance and security issues. Consequently security in CLDC is characterized by:

- *No Java Native Interface (JNI)*: Mainly for security and performance reasons, JNI [16] is not implemented in CLDC. Although, a Kilo Native Interface (KNI) [32] is provided for J2ME CLDC, KNI does not have the ability to dynamically load and call arbitrary native functions from Java programs (which could cause significant security problems in the absence of the full Java 2 security model).

- *No user-defined class loaders*: Mainly for security reasons, the class loader in CLDC is a built-in "bootstrap" class loader that cannot be overridden, replaced, or reconfigured. The elimination of user-defined class loaders is part of the "Sandbox" security model restrictions.

- *No thread groups or daemon threads*: While supporting multithreading, CLDC has no support for thread groups or daemon threads.

- *No support for reflection*: No reflection features are supported, and therefore there is no support for remote method invocation (RMI) or object serialization.

Low-Level Security

Low level security in CLDC is mainly based on type safety mechanisms. The class file verifier is the module in charge of type safety checking. The class file verifier ensures that the bytecodes and other items stored in class files cannot contain illegal instructions, cannot be executed in an illegal order, and cannot contain references to invalid memory locations or memory areas that are outside the Java object memory (the object heap) [34].

In J2ME CLDC and due to the constraints on device resources, this is done in two steps:

- *Off-device pre-verification*: During this phase, which is performed on the development platform, certain bytecodes will be removed, and class files will be augmented with additional *StackMap* attributes in order to speed up the runtime verification.
- *In-device verification*: The runtime verifier component uses the additional StackMap attributes generated by the preverifier to perform the actual class file verification efficiently [34].

Application-level Security

The CLDC application security is mainly ensured by adopting a *sandbox model*, by protecting system classes, and by restricting dynamic class loading:

- *Sandbox Model*:
  1. Java class files are properly verified and are valid Java classes.
  2. Only a closed predefined set of Java APIs is available to the application programmer, as defined by CLDC, profiles and manufacturer-specific classes.
  3. Downloading, installing, and managing MIDlets on the devices takes place at the native level inside the virtual machine. Therefore, the application programmer cannot modify or bypass the standard class loading mechanisms of the virtual machine.
  4. The set of functions accessible to the virtual machine is closed. Thus, developers cannot download any new libraries containing native functionality or access any native functions that are not part of the Java libraries provided by CLDC, MIDP, or the manufacturer.
- *Protecting System Classes*: In CLDC, the application programmer cannot override, modify, or add any classes to the protected system packages, i.e. packages belonging to configuration, profile, or manufacturer. Thus, the system classes are protected from the downloaded applications. Also, the application programmer is not able to manipulate the class file lookup order in any way.
- *Restrictions on dynamic class loading*: One important restriction is made on dynamically loading class files: A Java application can load application classes only from its own Java Archive (JAR) file.

MIDP Security

MIDP 1.0 Security

Application security in MIDP 1.0 is based on a Java sandbox model which was explained earlier. It is also important to note that in MIDP 1.0, MIDlet suites are allowed to save data in persistent storage files (called record stores). However, sharing of record stores between MIDlet suites is not allowed. With respect to end-to-end security, MIDP 1.0 specification does not include any cryptographic functionality. The only network protocol provided in MIDP 1.0 is the HTTP protocol.

MIDP 2.0 Security

The difference between MIDP 1.0 security and MIDP 2.0 security is that, in MIDP 2.0, accessing sensitive resources (APIs and functions) is not totally prohibited. Instead, MIDP 2.0 controls access to protected APIs by granting permissions to protection domains and binding each MIDlet on the device to one protection domain. Thus, a MIDlet will be granted all permissions provided to the protection domain that has been bound to it. A MIDlet is bound to one protection domain according to a well defined procedure that allows the AMS to authenticate the origin of a MIDlet: If one MIDlet can be authenticated, then it is qualified as *trusted*, otherwise, it will be qualified as *untrusted*. In addition, MIDP 2.0 introduces the ability to share record stores between MIDlet suites. The protection of record stores is discussed later in this section. Also, an important difference between the security of MIDP 1.0 and MIDP 2.0 is that MIDP 2.0 provides end-to-end security by allowing secure networking using HTTPS protocol.

Sensitive APIs

In MIDP 2.0, some capabilities of the device are exposed to MIDlets through a set of APIs that are identified as *sensitive* and therefore should be protected. The sensitive APIs in MIDP 2.0 are the ones related to connectivity and the `PushRegistry` class.

Permissions and Protection Domains

Access to sensitive APIs is protected by permissions. A protection domain defines a set of permissions, and for each permission, the protection domain defines the level of access to the API protected by the permission. The level of access can be either *Allowed* or *User*. For the *Allowed* level, the permission is granted without involving the user. As for the *User* level, access to the protected API requires explicit authorization from the user. This authorization can be in one the following modes [23]:

1. *Blanket* The permission is valid for every invocation of the protected API.

2. *Session:* The permission is valid during one execution of the MIDlet.
3. *Oneshot:* The user must be prompted for each invocation of the protected API.

By default, four protection domains are provided by MIDP 2.0:

- *Minimum*: This domain contains no permissions. Access is denied for all sensitive APIs.
- *Untrusted*: Requires that sensitive APIs can only be accessed through user permissions.
- *Trusted*: All permissions are granted.
- *Maximum*: Same as trusted.

In [33] which is an addendum to the MIDP 2.0 specification, protection domains are categorized into four classes, namely, *Manufacturer*, *Operator*, *Trusted third party*, and *Untrusted* domain. Protection domains are defined in a policy file. An example of the policy file is given in figure 1 which is the policy file provided with the RI. The procedure for determining whether a MIDlet suite is trusted is device-specific. Some devices might trust only MIDlet suites obtained from certain servers. Other devices might support only untrusted MIDlet suites. Others authenticate MIDlet suites using the Public Key Infrastructure (PKI), which is the case shown in figure 1. This authentication includes certificate path validation, signature checks and expiration checks for the certificates.

Signing a MIDlet suite

In order to sign a MIDlet suite, the signer needs to have a private and public key pair, and a certificate for his public key. If this certificate is not a certificate authority (a certificate that is stored in the device), there should be another certificate that vouches that the first one is valid. If this second certificate is still not a certificate authority, it requires a third certificate vouching for it, and so on until a root certificate is reached.

The procedure of signing the MIDlet consists of the execution of the following steps:

- The signer computes a digital fingerprint of the JAR file by applying a hash function (SHA-1).
- They then sign the digital fingerprint by encrypting it with the private Key.
- The signed fingerprint is placed in the JAD file.

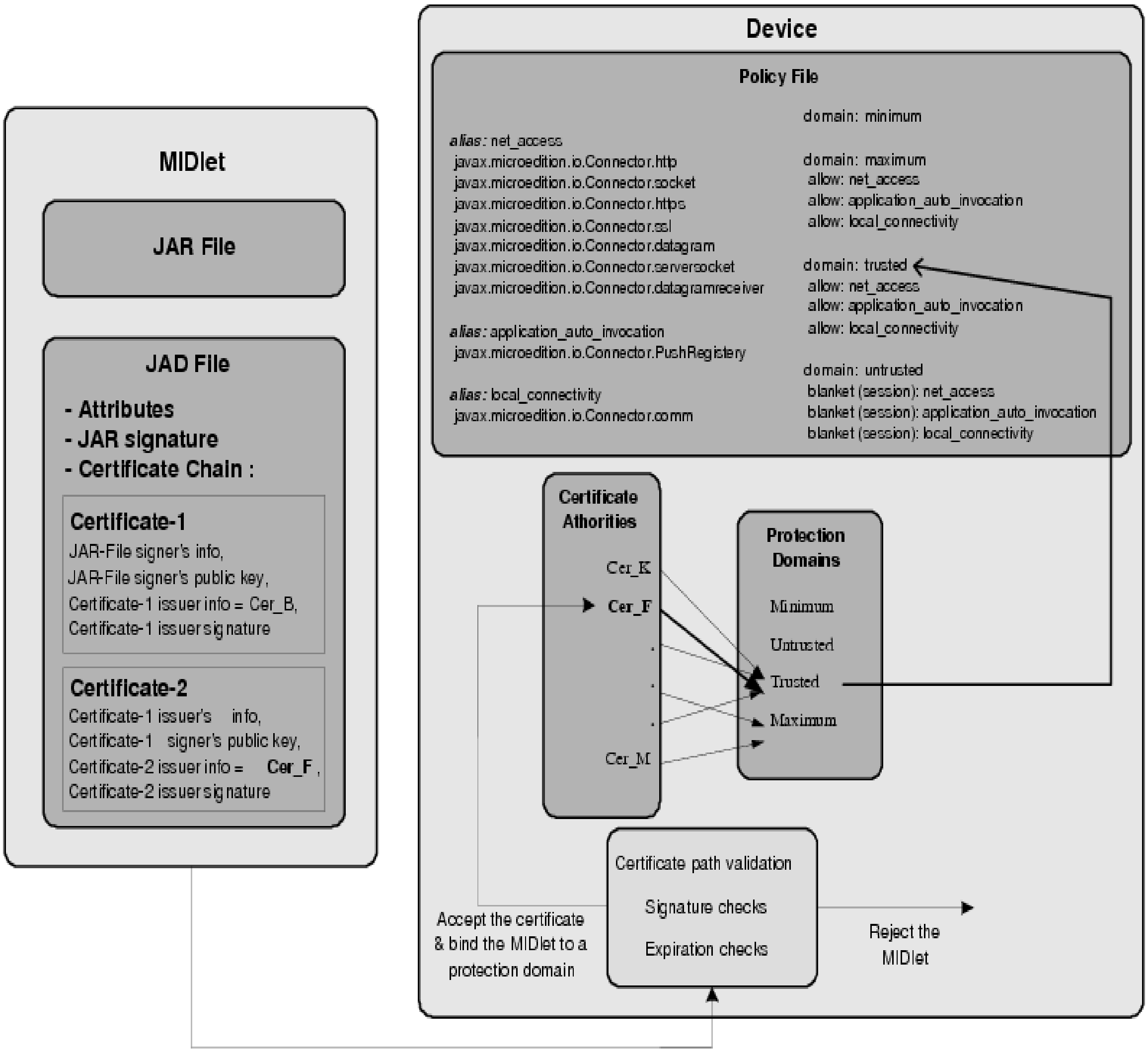

MIDlet
JAR File
JAD File
- Attributes
- JAR signature
- Certificate Chain :
Certificate-1
JAR-File signer's info,
JAR-File signer's public key,
Certificate-1 issuer info = Cer_B,
Certificate-1 issuer signature
Certificate-2
Certificate-1 issuer's info,
Certificate-1 signer's public key,
Certificate-2 issuer info = Cer_F ,
Certificate-2 issuer signature
Device
Policy File
alias: net_access
javax.microedition.io.Connector.http
javax.microedition.io.Connector.socket
javax.microedition.io.Connector.https
javax.microedition.io.Connector.ssl
javax.microedition.io.Connector.datagram
javax.microedition.io.Connector.serversocket
javax.microedition.io.Connector.datagramreceiver
alias: application_auto_invocation
javax.microedition.io.Connector.PushRegistery
alias: local_connectivity
javax.microedition.io.Connector.comm
domain: minimum
domain: maximum
allow: net_access
allow: application_auto_invocation
allow: local_connectivity
domain: trusted
allow: net_access
allow: application_auto_invocation
allow: local_connectivity
domain: untrusted
blanket (session): net_access
blanket (session): application_auto_invocation
blanket (session): local_connectivity
Certificate Athorities
Cer_K
Cer_F
.
.
.
Cer_M
Protection Domains
Minimum
Untrusted
Trusted
Maximum
Certificate path validation
Signature checks
Expiration checks
Accept the certificate & bind the MIDlet to a protection domain
Reject the MIDlet

Figure 1: Trusting a MIDlet suite and binding it to a protection domain

- The certificate of the public key is placed in the JAD (except if the certificate is the root certificate, which resides on the device), as well as the other certificates, if any.

Persistent Storage Security

In MIDP 2.0 a MIDlet suite can save data in a persistent storage area. The storage unit in J2ME CLDC is the *record store*. Each MIDlet suite can have one or more record stores, these are stored on the persistent storage of the device. Record stores are identified by a unique full name, which is a concatenation of the vendor name, the MIDlet suite name, and the record store name. Within the same MIDlet, two record stores cannot have the same name. However, if they belong to two different MIDlet suites, they can have the same name since their full names will be

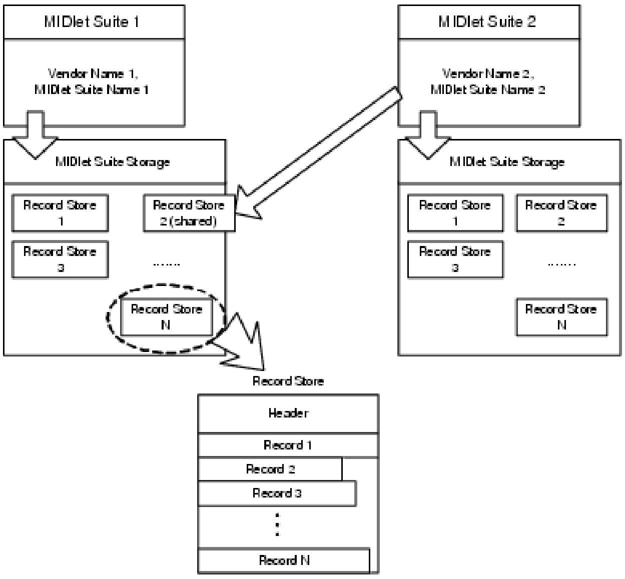


Figure 2: Record Stores in MIDP.

unique. The actual structure of the record store on the device storage consists of a header and a body. The header contains information about the record store while the body consists of a number of byte arrays called records, these contain the actual data to be stored. Figure 2 shows the structure of the storage system. The part of the Java platform responsible for manipulating the storage is called the Record Management System (RMS).

For MIDP 1.0, record stores were not allowed to be shared among MIDlet suites. In MIDP 2.0, sharing of record stores is allowed; the MIDlet suite that created the record store can choose to make it shared or not. Moreover, the sharing mode can be set to `read-only` or `read/write`. Sharing information is stored in the header of each record store, and the default mode of sharing is private (no sharing).

End-to-end Security

MIDP 2.0 specification mandates that HTTPS be implemented to allow secure connection with remote sites. HTTPS implementations must provide server authentication. The Certificate authorities present in the device are used to authenticate sites by verifying certificate chain provided by a server.

4 J2ME CLDC SECURITY MODEL EVALUATION

We provide in this section our evaluation of the J2ME CLDC security model. This section is organized into to parts:

- A comparison of the J2ME CLDC model with the security model of J2SE.

The objective of such comparison is to extract the resemblances between the two models and identify mechanisms dropped for the J2ME CLDC model or changed to fit in embedded platforms.

- Our evaluation is presented by giving the advantages and the drawbacks of the security model. This evaluation covers permissions, protection domains, security policy, and the protection of persistent storage.

To provide a credible evaluation of the J2ME CLDC security model, we investigated both, the published specifications and the available implementations. By analyzing specifications, we provide a comprehensive study of the J2ME CLDC security model (see the previous section), pointing out possible shortcomings and aspects open for improvement. The available implementations are investigated in order to enhance our understanding of the J2ME CLDC security model and to discover existent weaknesses through security testing. It is important to mention that we investigated several implementations of the platform like Sun's RI, phone emulators, and actual phones. Our evaluation is reinforced by a compilation of the reported flaws related to the security of J2ME CLDC security. The results of investigating the implementations are reported in the Section 5, which presents a vulnerability analysis of the J2ME CLDC platform.

J2ME CLDC Security versus J2SE Security

The security concepts of Java 2 (permissions, protection domains, security policy, etc.) are not typical to the J2ME CLDC platform. They were first introduced in the Java 2 Platform Security Architecture [31]. This architecture defines the security features of J2SE platform. However, the scope, the design, and the implementation of these concepts vary greatly in J2ME CLDC and J2SE platforms. In the following we highlight the main differences between them.

J2SE platform comes with a complete and full security model (Security Manager, Access Controller, etc.). This model cannot be fully deployed in J2ME CLDC platform because of the severely constrained resources (memory budget, processing capabilities, and power consumption). Indeed, the security model of J2ME CLDC can be considered as a subset of the J2SE security model. Some features were disabled and others were replaced by more lightweight ones.

In the following, we consider each concept at a time and we try to exhibit the main differences between both security models according to that concept. The main concepts that we focus on are: *permissions, protection domains, security policy, and security manager.*

Permissions

A permission protects access to a system resource. The system resources can be connectivity protocols, files, etc. Although the purpose is the same, two main differences exist in how this concept is implemented in J2ME CLDC and J2SE. (First), the number of resources protected by permissions in J2ME CLDC is less than the one in J2SE platform. In J2ME CLDC RI, only *connectivity protocols* and *push functionality* are guarded by permissions. We note that in more recent JSRs, other resources are defined to be protected by permissions (SMS and MMS [2], Bluetooth [36], etc). In J2SE, however, various resources are guarded by permissions (File, Property, Runtime, etc.). Moreover, it is possible to define new permissions that are more tailored to the specific needs of a given program. This is not possible in J2ME CLDC. For example, in the RI, all permissions are defined in `com.sun-.midp.security.Permissions` class which cannot be modified by the programmer. This difference between the two platforms may explain the rigidity of the J2ME CLDC security model compared to J2SE. Indeed, since only few valued resources are available in J2ME CLDC, the implementation of a fine-grained security policy will not entail important gains. However, the resulting overhead will be important. (Second), Permissions in J2ME CLDC are more atomic and are limited to either enabling or denying access to protected resources. In J2SE, permissions specify more information such as the CodeBase and the actions to be allowed. Due to the requirements of the mobile environment, J2ME CLDC introduces the concept of interaction mode which is not present in J2SE. This concept results in more involving the user when granting permissions to a given MIDlet. We can think of the interaction mode as a concept in J2ME CLDC to compensate the fact that users cannot change the security policy.

Protection Domains

In J2SE, a protection domain *ProtDom* is defined by a set of permissions *P* and a source *S*. All classes coming from *S* are bound to *ProtDom* and are granted permissions *P*. Whereas in J2ME CLDC, the protection domain *ProtDom* is defined by a set of permissions *P* and a certificate authority *CA*. All MIDlets signed by *CA* are bound to *ProtDom*. Hence, the first difference between protection domains in the two security models lays in the level of granularity. In J2SE, each class is bound to a protection domain, whereas in J2ME CLDC each MIDlet (Java application) is associated with a protection domain.

The way the protection domain concept is implemented differs in both platforms. In J2SE, a protection domain is created by instantiating the `java.security.ProtectionDomain` class. The set of permissions associated to that protection domain is determined based on the security policy. Hence, the protection domain is an object that specifies a set of permissions. In J2ME CLDC a protection domain is not represented by an object. It is only defined in the security policy. Since the security policy is not open to users for manipulation, there is no way by which the user can add new protection domains.

A major difference between J2ME CLDC and J2SE lays in the mechanism of assigning protection domains to classes and MIDlets. Once a MIDlet or class is loaded, it should be associated to a protection domain. In J2SE, this is done by the class loader and is based on the origin of the code. The source of the code can be defined by its URL and/or the entity or entities that signed it.

In J2ME CLDC, the association of MIDlets to protection domains is achieved according to the certification authority that is used to check the signature of the MIDlet. Indeed, the device maintains a list of root certificates. Each certificate is associated with a protection domain. Once a MIDlet is loaded, its certification chain is checked and the root certificate of this chain is used to bind the MIDlet to a protection domain. So Associating protection domains is based on the root certificates present in the device.

Security Policy

The security policy specifies which permission can be granted to which code. In the RI of J2ME CLDC, security policy is represented in one policy configuration file (`policy.txt` file). This file specifies for each protection domain the associated permissions with their interaction modes. In J2SE, the security policy is represented by a policy object (`java.security.policy`). Nevertheless, the policy information itself is stored typically in one or more policy configuration files. These files are composed of grant entries that specify the permission to be granted with the corresponding CodeBase and allowed actions.

In J2SE, the security policy can be changed statically or dynamically. By statically, we mean changing the security policy before launching the Java Runtime Environment (JRE). This can be achieved in two ways. Fist by using the *policy-Tool*. Second by changing directly the security policy file. By dynamically, we mean changing the security policy while the JRE is running. This can be achieved in two steps. The first step is to change the security policy file in the same way as the static alternative. The second step is to call the `refresh` method. Accordingly, the modifications to the security policy will be considered at runtime.

In J2ME CLDC, the security policy can only be changed statically. There is only one alternative which is modifying directly the policy configuration file. However, it should be noted that on real devices, the policy configuration file is not open to users for manipulation.

Security Manager

The security manager is the entity responsible for checking accesses to protected resources. Typically, it defines several methods whose names begin with the word "`check`". In J2SE, the `java.lang.SecurityManager` class represents the implementation of the security manager. The latter defines all "`check`" methods. The security manager of J2SE uses the Access Controller to decide whether to grant a permission requested by a given program. Actually, since JDK 1.2 all access control checks are done by the *AccessController*. The *securityManager* classes are kept to support compatibility with previous versions.

In J2ME CLDC, no security manager class is defined. It is up to the MIDP implementor to provide an implementation for this concept. For instance, in the RI the alternative is the Security Token (ST). Each entity that wants to access a protected resource needs a valid ST, that is, a ST with the required permissions. The `SecurityToken` class defines all "`check`" methods. However, these methods do not use an *AccessController*. As mentioned earlier, the reason for the absence of the *AccessController* is that at a given moment, only one protection domain is in effect. So, access control consists in checking only that protection domain. However in J2SE, several protection domains may be in effect at a given moment. Therefore, an *AccessController* is needed to analyze the current context and decide whether or not to grant a requested permission. A typical *AccessController* algorithm consists in inspecting the stack [4], that is, checking the protection domain of each frame in the stack.

Security Model Evaluation

Now, we present our evaluation of the J2ME CLDC security model. This evaluation is based on our understanding of the J2ME CLDC and the specification of security mechanisms deployed in this platform. Before presenting our evaluation of the J2ME CLDC security model, it is important to explain the positive impact of dropping some general Java features in the J2ME CLDC platform. The eliminated features related to security are:

- *Java Native Interface (JNI)*: JNI is absent from J2ME CLDC because it is very expensive to be deployed in embedded devices (memory constraints). This absence protects the platform from downloaded native code, which is difficult to protect from and can seriously damage the device.
- *User-defined class loaders*: The class loader in CLDC is a built-in class that cannot be overridden or replaced by the user. Having only one class loader will protect the platform from attacks that are based on user defined class loaders, which is one of the main attacking techniques on Java platforms [18].
- *Support for reflection*: As a result of dropping reflection features, no support for RMI (Remote Method Invocation) nor object serialization are provided in J2ME CLDC. This results in more protection of the internal architecture from being exposed to MIDlets.

J2ME CLDC security model is based on restricting access to sensitive resources on the device. Giving to a MIDlet the ability to access particular resources is based on the level of *trust* the device has for the MIDlet. The implementation of this model consists of the concepts of *permissions*, *protection domains* and *security policies*. Moreover certain security measures are taken by the Record Management System to protect data in the persistent storage of the device. In the following we present our comments on the previously mentioned security features of J2ME CLDC.

Permissions

- *The user is the weakest link in security*: One of the key features of MIDP 2.0 security model is that the device asks the user (by means of dialog screen) whether to allow a given MIDlet to call security-sensitive APIs or not. The advantage of this concept is that the user is aware of the attempts to use security-sensitive APIs. Also, this allows MIDlets in the *untrusted* protection domain to call protected APIs provided that the user accepts to grant the permissions. However, with this concept of interaction mode goes the risk of having the user answer automatically without checking the displayed messages. This may happen if the MIDlet asks for too many permissions. Moreover, one can maliciously develop a MIDlet that pushes the user to ignore the displayed messages and answer automatically just to get rid of the annoying messages. Hence, the strict procedure of signing MIDlets with trusted certificates, upon which the whole MIDP security model is based, can be circumvented by a precipitated permission granted by the user.

- *Programmers cannot define new permissions*: Programmers are not provided with the ability to define new permissions. In J2ME CLDC RI, a single class implements all permissions and it is not possible to define new permissions by subclassing this class. It is interesting to have the power of personalizing the defined set of permissions.

- *Permissions are atomic*: One MIDlet is granted access to the resource or not. The actions that one MIDlet can perform on the granted resource is not specified. It is preferred to have more detailed specification of what a MIDlet can perform on granted resources.

Protection Domains

Protection domain is a central concept in MIDP 2.0 security. A protection domain is defined as a set of permissions that will be granted to each MIDlet that is associated with this protection domain. According to this point of view, defining protection domains in J2ME CLDC is a way of organizing permissions. Our main critiques to the definition of protection domains in J2ME CLDC are the following:

- *Protection domains can be misused*: The task of binding the appropriate protection domain to a given MIDlet is critical for the security of MIDP. When granting a set of permissions to a MIDlet, one must be sure that the MIDlet will not misuse the granted permissions (accidentally or maliciously) and cause harm to the device. Developers could guarantee that their MIDlets are correct and safe by signing them. But in practice, it is often the case that a program written by a trusted developer can make a wrong behavior. The task of binding MIDlets to protection domains will be more complicated if the signer is a third party. Third parties are often manufacturers where the corresponding protection domains have more granted permissions.

- *Protection domains are not well standardized*: In MIDP specification, manufacturers and operators are free to define their proper protection domains. This will result in a standardization problem of protection domains. The absence of such a standard will have a negative impact on the facility of developing and deploying MIDlets. A developer who wants to deploy his MIDlets would have to require a different protection domain for each different manufacturer or operator. Moreover, a developer can be obliged to change his MIDlet if the set of permissions required by the MIDlet do not correspond to any permissions set of any protection domain on the device. This would prevent developers from one of the main advantages of the Java programming language, which is "Write Once, Run Everywhere".

- *No way to deny permissions of a protection domain*: We found interesting to have the ability to deny a subset of permissions among those already granted by a protection domain associated with a MIDlet suite. This is motivated by the two following arguments:

  1. According to the specification of protection domains, even if a permission is not granted by a protection domain (either allowed or user), a MIDlet still has the ability to get the permission by asking the user directly.
  2. A MIDlet that declares some permissions in the `MIDlet-Permissions` attribute, can during execution ask the user for other permissions of the protection domain.

The model of protection domains presented in J2ME CLDC has room for some refinements in order to harden the overall security model. We believe that denying permissions granted by the protection domain and not required by the MIDlet suite, will enhance considerably the security of J2ME CLDC. We can give the following main advantages:

1. The user will be protected from being asked for permissions that are not granted by the protection domain. In this case, the signing authority will have the complete responsibility of granting permissions to a MIDlet suite.
2. A MIDlet will no longer be able to ask for permissions that it did not ask for in its `MIDlet-Permissions` attribute.

3. No technique bypassing the user - like Siemens attacks - can be deployed to get permissions that are denied for one MIDlet suite.

Security Policy

The security policy in J2ME CLDC specifies protection domains with the associated permissions and interaction modes. Since the security policy file defines permissions to access sensitive resources, MIDlets do not have the power to change the security policy file. Access to protected resources is critical and will cost money to the user of the device. Thus preventing MIDlets from modifying the security policy is motivated by commercial and safety reasons.

At the other side, it is viewed as too rigid that the users do not have the ability to modify the security policy for any subset of sensitive APIs or for the data produced by MIDlets on the device. One possible extension to the current security policy model is to provide MIDlets with the power of defining permissions to access data in the persistent storage (which can be records in data bases, game scores, .etc)

Persistent Storage Protection

J2ME CLDC provides a model to store application data and protect them. Data can be stored in a persistent storage shared by all MIDlets. Data in record stores can be accessed or modified only from MIDlets creating them. The only case where a record store can be accessed (read or read/write) by a MIDlet other then the one creating it, is when the creating MIDlet declares the record store as shared record. Our critics to this model are the following:

- *Record stores are either shared or not*: This is a rigid model because often, MIDlets need to share their data with a specific set of MIDlets and rarely want to share records with all MIDlets on the device. If the shared data is critical, then it can be red by all MIDlets on the device. The risk will be more serious if the sensitive data is shared by read/write mode. In this case, the data will be exposed to malicious MIDlets on the device and can be modified in a non safe manner.

- *Data can be accessed within the device*: Critical data can be vulnerable to any attack from outside the RMS; record stores can be accessed from the device's utilities (without using a MIDlet). This is a serious problem because record stores will be manipulated as files (copied, renamed, deleted, etc.).

- *No encryption used for persistent data*: No encryption method is specified in order to protect sensitive data in the persistent storage. Thus, no critical data can be stored securely in the persistent storage.

5 VULNERABILITY ANALYSIS

The purpose of this section is to thoroughly investigate the presence of vulnerabilities in J2ME CLDC platform and to assess each found vulnerability. Vulnerability analysis is the process of carefully studying a system with the purpose of detecting security related errors. These errors may come from faults in the system specification or the system implementation. The presence of an error weakens the security structure of the system, making it vulnerable to malicious attacks.

Approach

There are several possible strategies for detecting vulnerabilities in software. In this paper, two main paths were taken for this purpose. The first path relied on inspecting the source code (RI of J2ME CLDC), looking for possible security related flaws. The second path consisted of executing black box testing (using phone emulators and actual phones), with the purpose of finding possible attacks on the platform. Our vulnerability analysis is reinforced by looking for already reported bugs, and investigating them to assess their seriousness from the security point of view.

The objective of security code inspection is to look for known types of vulnerabilities (e.g. buffer overflow vulnerability). In this regard, a list of security checks was compiled both for C and Java code to guide the inspection process. Also, we identified sensible areas in the code and investigated them more seriously in order to find vulnerabilities to certain known attack scenarios. As for security testing, test cases were designed, implemented, and executed in order to perform attacks compromising the security and safety of the platform. Each success of one test case execution will reveal the presence of some security vulnerability in the platform.

Previously Reported Flaws

Few security flaws about J2ME CLDC have been reported. The most serious one is the Siemens S55 SMS flaw. Besides, several problems about MIDP RI have been reported.

Siemens S55 SMS

In late 2003, the Phenoelit hackers group [24] has discovered that the Siemens S55 phone suffers from one vulnerability that allows malicious code on the device to send SMS messages without the authorization of the user. This attack can be carried out by a malicious MIDlet, which does not have the permission to send SMS messages. By executing the attack the MIDlet will be able to send an SMS message without the explicit authorization of the user. This is due to a race condition during which the Java code can overlay the normal permission request with an arbitrary screen display.

Problems on MIDP RI

The Bug Database of Sun Microsystems contains hundreds of problems about J2ME CLDC. However, few are related to security. In the following we describe the problems that we deem as relevant from the security standpoint:

1. When establishing a socket connection (`socket://hostname:portnumber`), permissions are necessarily needed. But if one runs the RI on PC where `portnumber` is already occupied, the application does not check for permission [30]. Instead, it throws an `IOException`. This is not correct because there is no need to access native sockets if application has not enough permissions. When investigating MIDP 2.0 RI for the same vulnerability, the execution results in throwing a `ConnectionNotFoundException` exception which means that the permission checking were bypassed.

2. A problem has been reported on the RSA algorithm implementation claiming that the big number division function checks the numerator instead of the divisor for zero [28]. On the available MIDP 2.0 RI, we could not be sure of this problem because the RSA algorithm implementation is provided in object files (without source files).

3. Basic Authentication Scheme is not fully supported in the RI [29]. According to the MIDP 2.0 specification, the device MUST be capable of "responding to a 401 (Unauthorized) or 407 (Proxy Authentication Required) response to an HTTP request by asking the user for a user name and password and re-sending the HTTP request with the credentials supplied. The device MUST be able to support at least the RFC2617 Basic Authentication Scheme" [5]. However, the RI utilizes Basic Authentication Scheme only in the case of MIDlet Suite installation (to retrieve JAD and JAR). Any other HTTP responses with 401, 407 won't be recognized to re-send the request with the user-supplied credentials.

4. The return value of `midpInitializeMemory()` method called in `main()` is never checked [27]. When memory allocation fails, system will crash without any way to figure out the reason of that crash.

Vulnerability Analysis Results

This section represents the findings of our vulnerability analysis performed on the J2ME CLDC platform. Results are in the form of flaws that would compromise the security and safety of the platform. They are organized according to the platform component in which they were discovered (e.g. storage system, KVM, etc.).

Networking Vulnerabilities

MIDP SSL Vulnerability

In order to establish a secure connection with remote sites (HTTPS), MIDP uses the SSLv3.0 protocol. The implementation is based on KSSL [7] from Sun Labs.

During the SSL handshake, the protocol has to generate random values to be used to compute the master secret. The master secret is then used to generate a set of symmetric keys for encryption. Hence, generating random values that are unpredictable is an important security aspect of SSL.

The method `PRand.generateData` is used to generate random data. This method computes the random values progressively 16 bytes by 16 bytes. The first chunk is computed by applying a hash algorithm (MD5) on the seed. Then, after the first chunk is copied into the random value array, the seed is updated and the hash algorithm is applied a second time to generate the second chunk. This operation continues until the random value array is completely populated.

Since MD5 is deterministic, in the case it is applied on the same seed, it will generate the same “random” value. Consequently, the challenge is “how to update the seed in an unpredictable fashion?”.

The RI method used to update the seed is called `updateSeed` and is given in the following:

```
public void updateSeed() {
  long l = System.currentTimeMillis();
  byte abyte0[] = new byte[8];

  for(int i = 0; i < 8; i++) {
    abyte0[i] = (byte)(int)(l & 255L);
    l >>>= 8;
  }
  md.update(seed, 0, seed.length);
  md.doFinal(abyte0,  0, abyte0.length, seed, 0);
}
```

It is easy to notice that the seed update depends only on the system time (`System.currentTimeMillis`). Hence, in order to obtain the random value generated by the client, all what the attacker has to do is to guess the precise system time (in milliseconds) at the moment of the random value computation. To this end, popular Ethernet sniffing tools can be used. These tools (e.g. tcpdump) record the precise time they see each packet. This allows the attacker to guess a very close interval of the correct system time. At that moment, it remains to try all possible values in that interval. For example, the attack that was carried out against the

Netscape browser implementation of SSL in 1996 [6] used sniffing tools to determine the seconds variable of the system time. Then, to find the microseconds variable, every possible value of the 1 million possibilities is tried.

Unauthorized SMS Sending Vulnerability

As mentioned earlier, the Phenoelit hackers group [24] has discovered that the Siemens S55 phone has a vulnerability that allows malicious code on the device to send SMS messages without the authorization of the user. The idea was to fill the screen with different items when the device is asking the user for SMS permission. In this way, the user unwittingly will approve sending SMS messages since he thinks that he is answering a different question.

In order to prove this vulnerability, we developed a MIDlet that tries to take advantage of this flaw. The MIDlet uses two threads. The first sends an SMS message and the second fills the screen with other items but without changing the buttons of the screen. The important code chunks of this MIDlet are illustrated in the following:

The first thread sends the SMS message:

```
public void startApp (){
  .....
  display = Display.getDisplay(this);
  .....
  ObscuringThread T  =  new  ObscuringThread();
  T.start();
  SMS.send("15142457980", SMSstr);
  .....
}
```

The second thread obscures the screen with other items:

```
public void run(){
 ObscureCanvas OCanvas  =  new ObscureCanvas();
 ....
 sleep(1000);
 smsAttack.display.setCurrent(OCanvas);
 ....
}
```

The key point in this attack is that only the screen is overwritten. The buttons (soft buttons) behavior is not changed and it is still about the SMS message permission. We ran the MIDlet on Siemens S55 emulator using Sun One Studio 4. The result was as we expected: The SMS authorization dialog (i.e. `send SMS ?`) was obscured by a different item (`play game`). This makes the user think that he is answering an invitation to play a game!

Since its publication, this flaw was always bound to Siemens S55 phones. However, nothing was said about its applicability on other phones. We run the previous MIDlet on other Siemens phones emulators, namely, 2128, CF62, and MC60. We found that all these phones are vulnerable to SMS authorization attack. By checking the APIs of all these phones, we found that the SMS APIs are almost the same, which explains our findings.

Unlike Siemens phones, Sun RI of MIDP is not vulnerable to this attack. Indeed, when the device asks the user for permission, MIDP RI prevents any modification of the screen until an answer is received. This is achieved by `preemptDisplay` method, which locks access to the `display` until the user provides an answer, then the `display` is unlocked by the `doneDisplay` method.

Storage System Vulnerabilities

Detailed analysis of the RMS using MIDP specifications and RI revealed the vulnerabilities listed below.

Unprotected Data Vulnerability

Data in record stores are not protected against malicious attacks. There is no mention in the specification of protecting sensitive user data for example with encryption and/or passwords. Data can be vulnerable to any attack from outside the RMS, such as when transferring data to or from a PC for backup and restoring. Moreover the whole storage system in MIDP can be accessed from any other file system on the device. This can happen in the case of PDAs, where the operating system can access the actual files of the record stores on the device storage and hence any sensitive data stored in them. An example of such access can be performed using the FExplorer software. This example is discussed with further issues in the next sections.

Managing the Available Free Persistent Storage Vulnerability

When a MIDlet needs to store information in the persistent storage, it can create new records. Since the persistent storage is shared by all Midlets installed on the device, restrictions must be made on the amount of storage attributed to each MIDlet. This is motivated by the fact that embedded devices have limited memory resources. As we can see from the MIDP specification, there is no restriction on the size of storage granted to one MIDlet, which means that one cannot prevent any MIDlet from getting all the available free space on the persistent storage for its

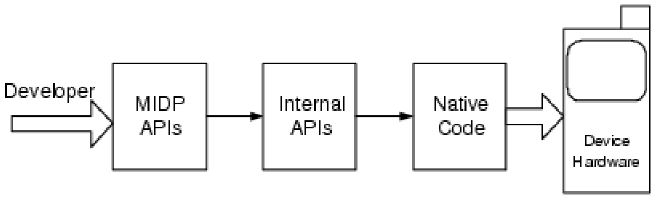


Figure 3: Various Levels of Abstraction in APIs for Mobile Devices.

record stores. By allowing this, all other MIDlets will be prevented from getting additional persistent storage (that can be vital for their life cycle). This vulnerability was discovered in the MIDP RI as well as in the Wireless Toolkits.

Unprotected Internal APIs Vulnerability

MIDP APIs were designed to provide all the functionalities needed by the developer, and they should be the only APIs available for direct use by developers. However, these are high level API's designed to make programming easier, so they need the help of other low level APIs and native methods to deal with the device hardware. The low level APIs are closer to the device hardware, and therefore are more difficult to program, but they have more privileges and less restrictions in order to be able to have more freedom when dealing with the device hardware. This should not affect the system security, provided that access to these APIs is restricted to the higher level APIs. In other words, developers should not have access to these low level APIs. This is not the case with the J2ME CLDC RI. Figure 3 illustrates how the different levels of APIs are accessed.

In order to exhibit the danger of having the programmer access to internal APIs, we give the example of deleting a record store belonging to another MIDlet. In the storage system of MIDP, `RecordStore` is one high level API that provides the functionalities needed by the developer to manipulate record stores such as opening, closing, deleting, etc. This class also checks for access rights before doing such actions, this is to protect data security and integrity, for instance no MIDlet is allowed to delete a record store of another MIDlet. There is another low level class, which is `RecordStoreFile`, this class is closer to the device hardware, it calls native methods and provides services to the `RecordStore` class. This class should not be available for direct use by developers, because it has more access rights and bypasses the security checks. In RI, this class can be used directly by programmers, which can compromise data security. We were able to use this vulnerability to have a MIDlet that deleted a record store belonging to another MIDlet.

First, a MIDlet `rmstest` was developed that creates a private (unshared) record store called `attack`. Then, another MIDlet was developed which uses the methods of the `RecordStoreFile` class to bypass security checks and delete the record store created by the previous MIDlet:

```
 private TextBox tb;
 ...
    Display.getDisplay(this).setCurrent(tb);
    tb.setCommandListener(this);
    String s = RecordStoreFile.getUniqueIdPath
                     ("Unknown", "rmsTest", "attack");
 boolean b = RecordStoreFile.deleteFile(s);
    if (b){
    tb.insert("Deleting successful", tb.size());
    }
    else{
    tb.insert("Cannot delete", tb.size());
    }
}
```

The first MIDlet was run first to create the record store, then the second one was run and succeeded to delete it.

Retrieving and Transferring JAR Files from a Device

A typical scenario of getting a MIDlet into a device is to connect to a provider web site and to download a chosen MIDlet usually after paying a certain amount of money. Once a MIDlet is installed on a device the user should be able to perform two kinds of operations, namely, executing and uninstalling the MIDlet. If, in addition, the user has the capability to transfer the MIDlet and make it run on another device, it results in a problem for the provider of the MIDlet. Indeed, this allows for illegal redistribution of MIDlets and consequently for financial losses.

In our experiments, we succeeded to transfer MIDlets from one device to another. This was possible thanks to a free software for Series 60 phones [21]. The FExplorer software [8] makes it possible to navigate through the files and MIDlets installed on the device just like navigating on a desktop file system. In order to transfer MIDlet JAD and JAR files into a second device, all we had to do is to go to the location where these files are stored and choose any sending option. This operation is also possible in all Series 60 devices. These include Samsung, Siemens, Panasonic and mainly Nokia devices [21].

Finally, it is important to note that transferring is not possible for DRM (Digital Right Management) protected MIDlets (protection should be at least by the forward lock mode [22]).

Retrieving and Transferring MIDlet Persistent Data

Using FExplorer software, (in addition to JAR and JAD files) it is also possible to transfer MIDlet persistent data from a device to another. Indeed, `rms.db` file that holds all MIDlet persistent data is located in the same location as JAD and JAR files and can be transferred following the same steps. Moreover, the DRM issue is no more valid for `rms.db`. That is, even if the MIDlet is DRM protected the `rms.db` can be transferred because the DRM protection holds only for JAR files [22]. This may have a serious impact on the privacy of the MIDlet since it is possible to tamper with its persistent data.

KVM Vulnerabilities

Buffer Overflow Vulnerability

Buffer overflow is a well-known problem and may result in many security breaches. It occurs when the program does not perform bounds checking of buffers before copying. A typical scenario consists in the following: a buffer (e.g. an array of integer) is located in the execution stack. The program tries to overwrite that buffer with another one using for example `strcpy` function (for C code). The danger occurs when the second buffer (source) is larger than the first (destination). In that case some values in the execution stack may be overwritten. Among these values it is possible to overwrite the return address of the current function. By overwriting that address, the attacker will be able to jump to whatever code he wants. This allows the attacker to execute the code that he wants with the same privilege as the program.

By inspecting the source code of KVM, we identified a memory overflow vulnerability. The vulnerable code is the following code in `native.c` file:

```
sprintf(str_buffer," Method %s :: %s not found",
        className, methodName(thisMethod));
```

This code throws an exception if a native method is declared in a class file without implementing it elsewhere. The code does not check the size of the message that will be stored in `str_buffer`. Knowing that `str_buffer` is a global variable declared as an array of characters (`char str_buffer[512];`), it is clear that the code might result in a strange behavior if the size of the string to be stored in `str_buffer` exceeds 512. One part of the string to be stored in `str_buffer` is the name of the invoked native method. Knowing that no restrictions are imposed by the VM on the size of method or field names, we wrote a simple Java program that declares a native method name counting 2000 characters (greater than the `str_buffer` size). This native method is declared without giving any implementation (in order to force the throw of the exception). When the exception is thrown, the native method name is concatenated in `str_buffer` overwriting more than 1500 characters in the memory segment. The Java program exploiting this vulnerability is the following:

```
public HelloWorld() {

 System.out.println("Hello World");
 /*  the native method name is 2000 characters */
 HelloWorldHelloWorld...();
}

public static void main(String arg[]) {
 HelloWorld hw = new HelloWorld();
}

//  the native method name is 2000 characters //
public native void HelloWorldHelloWorld...(); }
```

6 CONCLUSION AND FUTURE WORK

In this paper, we provide a careful study of J2ME CLDC security. We started by presenting the security architecture of this Java platform, followed by our evaluation of the underlying security model. In our study, we investigated the existing implementations of the platform in order to look for vulnerabilities. Actual phone models were tested and an important set of attack scenarios were designed and executed. We showed that the J2ME CLDC security model needs some refinements (e.g. permissions and protection domains). Moreover, we demonstrated the presence of some vulnerabilities exist in the RI of MIDP 2.0 (e.g. SSL implementation). Some phones were also shown to be vulnerable to security attacks like the Siemens SMS attack, while other phones followed a restrictive approach in implementing the J2ME CLDC platform. With this study in hand, improvements can be performed to harden the J2ME CLDC security. Therefore, the security model can be improved by fixing the discovered vulnerabilities and by proposing new security extensions.

REFERENCES

[1] S. Chadha. J2me issues in the real wireless world. http://www.microjava.com/articles/perspective/issues?content_id=4323, January 2003.

[2] J. Eichhloz. JSR 205 Wireless Messaging API 2.0, June 2004.

[3] Carl Ellison and Bruce Schneier. Ten risks of PKI: What you're not being told about Public Key Infrastructure. *Computer Security Journal*, 16(1):1–7, 2000.

[4] C&#233;dric Fournet and Andrew D. Gordon. Stack inspection: theory and variants. In *Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 307–318. ACM Press, 2002.

[5] J. Franks, P. Hallam-Baker, J. Hostetler, S. Lawrence, P. Leach, A. Luotonen, and L. Stewart. RFC 2617: HTTP Authentication: Basic and Digest Access Authentication, June 1999.

[6] Ian Goldberg and David Wagner. Randomness and the Netscape browser. *Dr. Dobb's Journal of Software Tools*, 21(1):66, 68–70, January 1996.

[7] Vipul Gupta and Sumit Gupta. KSSL: Experiments in Wireless Internet Security. Technical Report TR-2001-103, Sun Microsystems, Inc, Santa Clara, California, USA, November 2001.

[8] D. Hugo. FExplorer Web Site. http://users.skynet.be/domi/fexplorer.htm.

[9] Wassim Itani and Ayman Kayssi. J2me application-layer end-to-end security for m-commerce. *Journal of Network and Computer Applications*, 27(1):13–32, January 2004.

[10] G. Steele J. Gosling, B. Joy and G. Bracha. *The Java Language Specification Second Edition*. The Java Series. Addison-Wesley, Boston, Mass., 2000.

[11] J. Knudsen. MIDP Application Security 1: Design Concerns and Cryptography. http://developers.sun.com/techtopics/mobility/midp/articles/security1/, September 2002.

[12] J. Knudsen. MIDP Application Security 2: Understanding SSL and TLS. http://developers.sun.com/techtopics/mobility/midp/articles/security2/, October 2002.

[13] J. Knudsen. MIDP Application Security 3: Authentication in MIDP. http://developers.sun.com/techtopics/mobility/midp/articles/security3/, December 2002.

[14] J. Knudsen. MIDP Application Security 4: Encryption in MIDP. http://developers.sun.com/techtopics/mobility/midp/articles/security4/, June 2003.

[15] O. Kolsi and T. Virtanen. MIDP 2.0 Security Enhancements. In *Proceedings of the 37th Annual Hawaii International Conference on System Sciences (HICSS'04) - Track 9 - Volume 9*, Washington DC, USA, dec 2004.

[16] Sheng Liang. *Java Native Interface: Programmer's Guide and Specification*. Addison-Wesley, Reading, MA, USA, 1999.

[17] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification (Second Edition)*. Addison Wesley, April 1999.

[18] LSD Research Group. Java and Java Virtual Machine Security, Vulnerabilities and their Exploitation Techniques. http://www.blackhat.com/presentations/bh-asia-02/LSD/bh-asia-02-lsd-article.pdf, September 2002.

[19] Sun MicroSystems. Connected, Limited Device Configuration. Specification Version 1.0, Java 2 Platform Micro Edition. Technical report, Sun MicroSystems, California, USA, May 2000.

[20] Sun MicroSystems. KVM Porting Guide. Technical report, Sun MicroSystems, California, USA, September 2001.

[21] Nokia. Series 60 Platform. http://www.nokia.com/nokia/0,8764,46827,00.html.

[22] OMA. Implementation Best Practices for OMA DRM v1.0 Protected MIDlets, May 2004.

[23] J. Van Peursem. JSR 118 Mobile Information Device Profile 2.0, November 2002.

[24] Phenoelit Hackers Group. http://www.phenoelit.de/, 2003.

[25] Roger Riggs, Antero Taivalsaari, Mark VandenBrink, and Jim Holliday. *Programming wireless devices with the Java 2 platform, micro edition: J2ME Connected Limited Device Configuration (CLDC), Mobile Information Device Profile (MIDP)*. Addison-Wesley, Reading, MA, USA, 2001.

[26] T. Sayeed, A. Taivalsaari, and F. Yellin. Inside The K Virtual Machine. http://java.sun.com/javaone/javaone2001/pdfs/1113.pdf, Jan 2001.

[27] Bug 4824821: Return value of midpInitializeMemory is not checked. http://bugs.sun.com/bugdatabase/view_bug.do?bug_id=4824821, February 2003.

[28] Bug 4959337: RSA Division by Zero. http://bugs.sun.com/bugdatabase/view_bug.do?bug_id=4959337, November 2003.

[29] Bug 4963644: Basic Authentication Scheme is not fully supported . http://bugs.sun.com/bugdatabase/view_bug.do?bug_id=4963644, December 2003.

[30] Bug 4802893: RI checks sockets before checking permissions. http://bugs.sun.com/bugdatabase/view_bug.do?bug_id=4802893, January 2004.

[31] Sun Microsystems. Java 2 Platform Security Architecture. http://java.sun.com/j2se/1.4.2/docs/guide/security/index.html, 2002.

[32] Sun Microsystems. KNI Specification K Native Interface (KNI) 1.0. http://www.carfield.com.hk/java_store/j2me/j2me_cldc/doc/kni/html/index.html, October 2002.

[33] Sun Microsystems. The Recommended Security Policy for GSM/UMTS Compliant Devices, Addendum to the Mobile Information Device Profile version 2.0, 2002.

[34] A. Taivalsaari. JSR 139 J2ME Connected Limited Device Configuration 1.1, March 2003.

[35] The Legion of the Bouncy Castle. Bouncy Castle Cryptography API. http://www.bouncycastle.org, 2004.

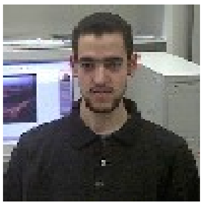
[36] R. Viswanathan. JSR 82 Java APIs for Bluetooth, Mars 2002.

[37] Michael Juntao Yuan and Ju Long. Securing wireless j2me. http://www-106.ibm.com/developerworks/java/library/wi-secj2me.html, June 2002.

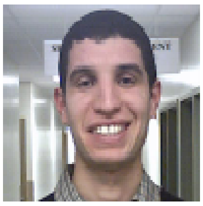
ABOUT THE AUTHORS

Mourad Debbabi holds Ph.D. and M.Sc. degrees in computer science from Paris-XI Orsay, University, France. He published more than 80 research papers in journals and conferences on computer security, formal semantics, Java security and acceleration, cryptographic protocols, malicious code detection, programming languages, type theory and specification and verification of safety-critical systems. He is a Full Professor and the Associate Director of the Concordia Institute for Information Systems Engineering, Concordia University, Montreal, Quebec, Canada. He is the Specification Lead of four Standard JAIN (Java Intelligent Networks) Java Specification Requests (JSRs) dedicated to the elaboration of standard specifications for presence and instant messaging. In the past, he served as Senior Scientist at the Panasonic Information and Network Technologies Laboratory, Princeton, New Jersey, USA; Associate Professor at the Computer Science Department of Laval University, Quebec, Canada; Senior Scientist at General Electric Research Center, New York, USA; Research Associate at the Computer Science Department of Stanford University, California, USA; and Permanent Researcher at the Bull Corporate Research Center, Paris, France. He can be reached at debbabi@ciise.concordia.ca. See also `http://www.ciise.concordia.ca/~debbabi`.

Mohamed Saleh is a Ph.D. student at Concordia Institute for Information Systems Engineering (CIISE), Concordia University, Montreal, Quebec, Canada. He is a member of the Computer Security Laboratory (CSL) at CIISE. He is pursuing a Ph.D. thesis on the specification and analysis of security protocols using game semantics. He can be reached at m_saleh@ece.concordia.ca.



Chamseddine Talhi is a researcher at CSL (Computer Security Laboratory) research group at Concordia Institute for Information Systems Engineering, Concordia University, Montreal, Quebec, Canada. He holds an M.Sc. degree from Constantine University, Algeria. He is pursuing a Ph.D. thesis on the security of embedded Java platforms. He is interested in characterizing enforceable security policies in resource-constrained platforms. In the past years, he worked on formal specification and verification of telecommunication services. He participated in the design and the implementation of a Java optimizing compiler for J2ME/CLDC. He can be reached at talhi@ciise.concordia.ca.



Sami Zhioua was a researcher at CSL (Computer Security Laboratory) research group at Concordia Institute for Information Systems Engineering, Concordia University, Montreal, Quebec, Canada. The main topic of his research activities is the acceleration of Java in the context of embedded systems. The goal of his research is to design and implement new techniques in order to improve the performance of embedded Java virtual machines. He participated in the design and implementation of a dynamic optimizing compiler for J2ME/CLDC. He can be reached at zhioua@ciise.concordia.ca.