Java for Mobile Devices: A Security Study

Mourad debbabi, Mohamed Saleh, Chamseddine Talhi and Sami Zhioua
Computer Security Laboratory
Concordia Institute for Information Systems Engineering
Concordia University, Montreal, Canada
{debbabi, m_saleh, talhi, zhioua}@ciise.concordia.ca

Abstract

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 the security architecture of J2ME CLDC. Second, we provide a vulnerability analysis of this Java platform. The analyzed 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 right 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 [14, 20]. 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.

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 [15, 21]. The APIs are CLDC [14] and MIDP[18, 20]. The tools are the pre-verifier and the Java Code Compacter (JCC). KVM (Kilobyte Virtual Machine) is an implementation of the Java Virtual Machine (JVM) [9]. 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 represents a careful study of J2ME CLDC security aspects with the purpose of providing a security evaluation for this Java platform. In this regard, we followed two main paths. One is related to the specifications

and the other to implementations. In the case of the specifications, we intend to provide a comprehensive study of the J2ME CLDC security architecture, pointing out possible shortcomings and aspects open for improvement. As for implementations, our aim is to look into several implementations of the platform like Sun's Reference Implementation (RI), phone emulators, and actual phones. This is carried out with the purpose of analyzing code vulnerabilities leading to security holes. The usefulness of such an investigation is to find out areas of common vulnerabilities and relate them either to the specifications or to common programming mistakes.

By identifying weaknesses that may represents sources of security breaches, our security evaluation gives a start point to a process that aims to improve the security of J2ME CLDC. This paper is organized into four sections beginning with the introduction. In section 2, we present the main security architecture of J2ME CLDC. In section 3, we list the results of the vulnerability analysis by starting with the previously reported flaws. Finally, section 4 concludes the paper.

2 J2ME CLDC Security Architecture

The high-level J2ME CLDC architecture defines 3 layers on top of the device's operating system (Figure 1): The *virtual machine*(KVM) [12], 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). J2ME CLDC platform can be further extended by combining various optional packages with the configurations and the associated profiles.

Applications developed for the J2ME CLDC platform are called `MIDlets`. They 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 following files: The *JAR manifest*, class files, and supporting files. The JAR manifest is a text file that contains various attributes like the MIDlet name and the vendor name. Class files are the preverified classes, and supporting files could be graphic files for instance.

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 text file that contains several attributes like the MIDlet name and MIDP version needed to run the MIDlet. Some of these attributes are mandatory while others are optional. The software entity on the device that is responsible for MIDlet management such as downloading, installing, running, etc. is called the `Application Management System` (AMS), or the `Java Application Manager` (JAM).

When presenting the security architecture of J2ME CLDC, we make distinction between CLDC and MIDP. The reason behind this distinction is that security concerns are distributed between the two. The security of J2ME CLDC platform can be categorized into *low-level* security, *application* security, and *end-to-end* security:

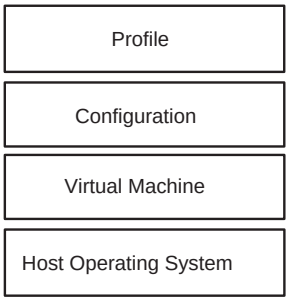

Figure 1. High-Level J2ME CLDC Architecture

- *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 [12].
- By *application-level* security, we mean that "Java application can access only those libraries, system resources and other components that the device and the Java application environment allows it to access" [14].
- *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.

In J2ME CLDC platform, low-level and application security are addressed in CLDC, while MIDP addresses *application* and *end-to-end* security.

2.1 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. Those dropped Java features are the following:

- *No Java Native Interface (JNI)*: Mainly for security and performance reasons, JNI [11] is not implemented in CLDC. Although, a Kilo Native Interface (KNI) [25] is provided for J2ME CLDC, KNI has not the ability to dynamically load and call arbitrary native functions from Java programs (which could pose 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.

2.1.1 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) [14].

Since conventional class file verification is too demanding for resource-constrained devices, class files are first *preverified* on the development platform before being saved on the device. The verifier performs only a linear scan of the bytecode, without the need of a costly iterative dataflow algorithm like the one used by the conventional verifier. The details of the J2ME CLDC verification process can be found in [5].

2.1.2 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*: In the CLDC Sandbox model, an application must run in a closed environment in which the application can access only those libraries that have been defined by the configuration, profiles, and other classes supported by the device. More specifically, the CLDC sandbox model requires that:
  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. This restriction ensures that:
  1. Java applications on a device cannot interfere with each other or steal data from each other.
  2. Third-party applications cannot gain access to the private or protected components of the Java classes that the device manufacturer or a service provider may have provided as part of the system applications.

2.2 MIDP Security

In the following, we present the security architecture of MIDP 1.0 and MIDP 2.0. Although, security models in both MIDP 1.0 and MIDP 2.0 are limited security models (compared to J2SE/EE), MIDP 2.0 provides more security mechanisms than those provided by MIDP 1.0. MIDP 2.0 exposes to MIDlets more capabilities of the device, and provides the security mechanisms needed to control the use of these capabilities.

2.2.1 MIDP 1.0 Security

Application security in MIDP 1.0 is based on the Java sandbox model. The sandbox security model provided by MIDP 1.0 (and CLDC) is different from the conventional Java sandbox model. In fact, no *Security Manager* nor *Security Policies* (as for J2SE/EE) are used for access control.

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 record stores between MIDlet suites is not allowed, which means that a MIDlet has no way to access a record store belonging to another MIDlet. This offers a good protection of MIDlet persistent storage.

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. MIDlet suites are usually downloaded from the Internet to the device without almost any protection using HTTP or WAP. The HTTP Basic Authentication Scheme is the only mandatory security mechanism, which is not a strong security mechanism. Since MIDlets in MIDP 1.0 cannot be signed, the integrity or authenticity of downloaded applications cannot be verified.

2.2.2 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 one 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 and identify the protection domain to be bound to this MIDlet. If one MIDlet can be authenticated, then it is qualified as *trusted*, otherwise, it will be qualified as *untrusted*. MIDP 2.0 introduces the ability to share record stores between MIDlet suites. The protection of record stores is discussed later in this section. 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 and are protected by permissions. Thus a set of APIs are defined to be used as interface between MIDlets and the exposed capabilities of the device. The set of these APIs is identified as *sensitive*. The sensitive APIs in MIDP 2.0 are the ones related to networking in addition to the *PushRegistry* class that allows for autimatic launching of MIDlets.

• **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*. The "Allowed" permission means that the MIDlet can access the sensitive API directly, whereas the "User" permission means that the user has to approve this access. This can be with one of the following interaction mode [18]:

- *Blanket* The permission is valid for every invocation of the protected API until the MIDlet suite is uninstalled or the permission is changed by the user.
- *Session:* The permission is valid during one execution of the MIDlet (any MIDlet in the MIDlet suite). For each execution of the MIDlet, the user must be prompted on or before the first invocation of the protected API.
- *Oneshot:* The user must be prompted for each invocation of the protected API.

In [26] 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.

• **Granting permissions to MIDlets**

A MIDlet suite (each MIDlet in the MIDlet suite) is granted permissions by applying the following principles:

- Each MIDlet suite is bound to a protection domain. This association depends on the degree of trust the device has for the MIDlet suite. A MIDlet suite can be either *trusted* or *untrusted*. An *untrusted* MIDlet suite is one for which the origin and the integrity of its JAR file cannot be reliably determined by the device. MIDlet suites compliant with MIDP 1.0 are considered as *untrusted* in MIDP 2.0. A *trusted* MIDlet suite is one for which the device can authenticate the origin and verify that the JAR file has not been tampered with.
- A MIDlet suite can require a set of permissions by listing them in two attributes of the JAD file:
  1. The `MIDlet-Permissions` attribute lists permissions that are vital (critical) to the execution of the MIDlet.
  2. The `MIDlet-Permissions-Opt` attribute lists permissions that may be needed during the execution but the MIDlet can still run if those permissions are not granted to it (non-critical).

  The presence of these two attributes in the JAD file allows the AMS to verify that the associated MIDlet suite is suitable for the device before loading the full JAR file.
- If the `MIDlet-Permissions` attribute is defined and the corresponding permissions set is CP, then the set of permissions granted to the MIDlet suite is equal to CP if CP is included in the permissions set granted to the protection domain.
- If the `MIDlet-Permissions` attribute is not defined, then the set of permissions granted to the MIDlet suite is equal to the set of permissions granted to the protection domain.

• **Trusting MIDlet Suites**

The procedure for determining whether a MIDlet suite is trusted or untrusted 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).

• **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 can not 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 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. 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.

3 Vulnerability Analysis

In this section, we present our vulnerability analysis of the J2ME CLDC security. We start by listing the most important previously reported flaws. After that, we present the vulnerabilities discovered by our team using mainly two principle tools. The first relies on inspecting the reference implementation source code, looking for possible security related flaws. The second consisted of executing black box tests on the code (using phone emulators and actual phones), with the purpose of finding possible attacks on the platform.

3.1 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 the Sun's MIDP RI have been reported.

3.1.1 Siemens S55 SMS

In late 2003, the Phenoelit hackers group [19] has discovered that the Siemens S55 phone has a vulnerability that makes the device send SMS messages without the authorization of the user. This attack can be carried out by a malicious MIDlet that when loaded by the target user, will send an SMS message from the target user's system without asking for permission. This is due to a race condition during which the Java code can overlay the normal permission request with an arbitrary screen display.

3.1.2 Problems on Sun's 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 relevant from the security standpoint.

Permissions are necessary needed to establish a socket connection (`socket://hostname:portnumber`). But if one runs the RI on PC where `portnumber` is already occupied, the application does not check for permission [24]. Instead, it throws an `IOException`. This is not correct because there is no need to access native sockets if application has not enough permissions. We investigated this problem on MIDP 2.0 RI and it generated a ConnectionNotFoundException which means that permission checking were bypassed.

A problem has been reported on RSA algorithm implementation claiming that the big number division function checks the numerator instead of the divisor for zero [23]. On the available MIDP 2.0 RI, we could not check this problem because the RSA algorithm implementation is provided only in object files.

The return value of `midpInitializeMemory()` method called in `main()` is never checked [22]. When memory allocation fails, system will crash without any way to figure out the reason of that crash.

In the sequel, we present the approach we followed to discover vulnerabilities in J2ME CLDC, then we list the vulnerabilities we were able to find. They are organized according to the components in which they were discovered (e.g. storage system, KVM, etc.).

3.2 Our Approach

Efforts in software security analysis, (i.e., developing techniques to assess security of software and to avoid security flaws) fall into: Vulnerability analysis, static code analysis, security testing, formal verification, and security evaluation standard methodologies. Vulnerability analysis mainly refers to efforts directed towards classification of security bugs. A good example for this is the work done by Krsul [10] and Bishop [1]. The ultimate goal is to develop tools that would detect vulnerabilities in software based on

the characteristics of the various "types" of vulnerabilities. The term "vulnerability analysis" is also sometimes used meaning the analysis of a software system (using various techniques) to detect security flaws.

Static code analysis can be used to find security-related errors. Several methods exist that could be manual as in code inspection or automated using tools. The main idea is to look for coding errors based on a compiled list of common security-related errors or known unsafe function calls (e.g., function `strcpy()` in C/C++ is vulnerable to buffer overflow). In [29], static analysis for Java is presented together with a tool for the same purpose, whereas [28] presents a tool for C/C++ code.

In security testing, techniques of property-based testing [4] are mostly used. Attention is focussed on proving that the software under test satisfies a certain property extracted from the specifications. This property could, for instance, be that users should be authenticated before they are allowed to do any action. In this case, the software entity responsible for authentication is tested. However, research in security testing also investigates other techniques such as in [27], where fault injection and stress testing are considered. It is important to note here that also formal verification methods can be used for the verification of security properties (e.g., using model checking), many examples exist for security protocols.

Several standard methodologies exist that aim to provide guidelines for IT systems security evaluators. The idea is to propose a number of security mechanisms the system can implement, and a number of checks the system has to go through to provide a certain level of assurance that the security mechanisms were correctly implemented. The most prominent is the Common Criteria (CC) methodology [2] that was selected as an ISO standard (ISO 15408). It is meant to be a replacement for some other methods that preceded it; namely, the Trusted Computer System Evaluation Criteria (TCSEC), and the Information Technology Security Evaluation Criteria (ITSEC).

• **Our Methodology** The methodology used to do the vulnerability analysis is depicted in Figure 2 and consists of the following five phases:

- *Phase 1*: Study of platform components.
- *Phase 2*: Reverse engineering.
- *Phase 3*: Static code analysis.
- *Phase 4*: Security testing.
- *Phase 5*: Risk analysis.

We explain hereafter the details of each phase of this methodology.

Phase 1 aims to identify the major system software components. We consider those component APIs that are recommended as mandatory in the latest revision of the Java Technology for the Wireless Industry (JTWI) i.e., JSR 185. Besides KVM, the mandatory components are CLDC, MIDP and Wireless Messaging API (WMA). Available specification documents from the Java Community Process (JCP) and related publications are studied.

Phase 2 aims to reverse engineer the platform. The analyzed source code is that of Sun's reference implementation (RI) for KVM, CLDC, MIDP, and WMA. The languages used in the RI are C (for KVM and CLDC), and Java (for CLDC, MIDP and WMA). In order to achieve a better understanding of the code, we resort to reverse engineering tools (e.g., Understand for C++, Understand for Java, and Rational Rose). Using these tools, we are able to compute abstractions and recover the underlying architecture and design of the platform.

Phase 3 aims to carry out a security analysis of the code for the purpose of discovering vulnerabilities. To this end, we use two techniques: Security code inspection and automatic security analysis. Security code inspection is carried out according to the "checklist approach" listed in [3]. For this purpose, we compile two lists of common security errors; one for Java, and the other for C, to be used as a guide in the inspection process. The automatic code security analysis is carried out by tools such as FlawFinder and ITS4 [28] for C, and Jlint [29] for Java. Tools are applied to *all* the source files. The result of this phase is a list of *probable* security flaws. This list is used to feed the next phase.

Phase 4 aims to discover more vulnerabilities by means of security testing. To this end, we design test cases in the form of security attacks. The design of these attack scenarios is based on: (1) The list of probable weaknesses that we compiled during code inspection, (2) the known types of vulnerabilities that are presented in several papers such as [10] and (3) the security properties that are extracted from the specification documents according to property-based testing principles [4]. These test cases are run on:

- Sun's reference implementation.
- Phone emulators: Sun's Wireless Tool Kit (WTK), Siemens, Motorola and Nokia.
- Actual phones: Motorola V600 and Nokia 3600.

To be more focused, each test case is designed to attack a certain functional component of the system. These components are: The virtual machine, the networking components, the threading system, the storage system (for user data, and JAR files), and the display.

Phase 5 aims to structure the discovered vulnerabilities and assess the underlying risks according to a well-established and standard framework. The MEHARI method [13] is used to achieve this objective. The criteria of MEHARI are used to structure the discovered vulnerabilities into an appropriate classification. Afterwards, the seriousness of each vulnerability is assessed based on the

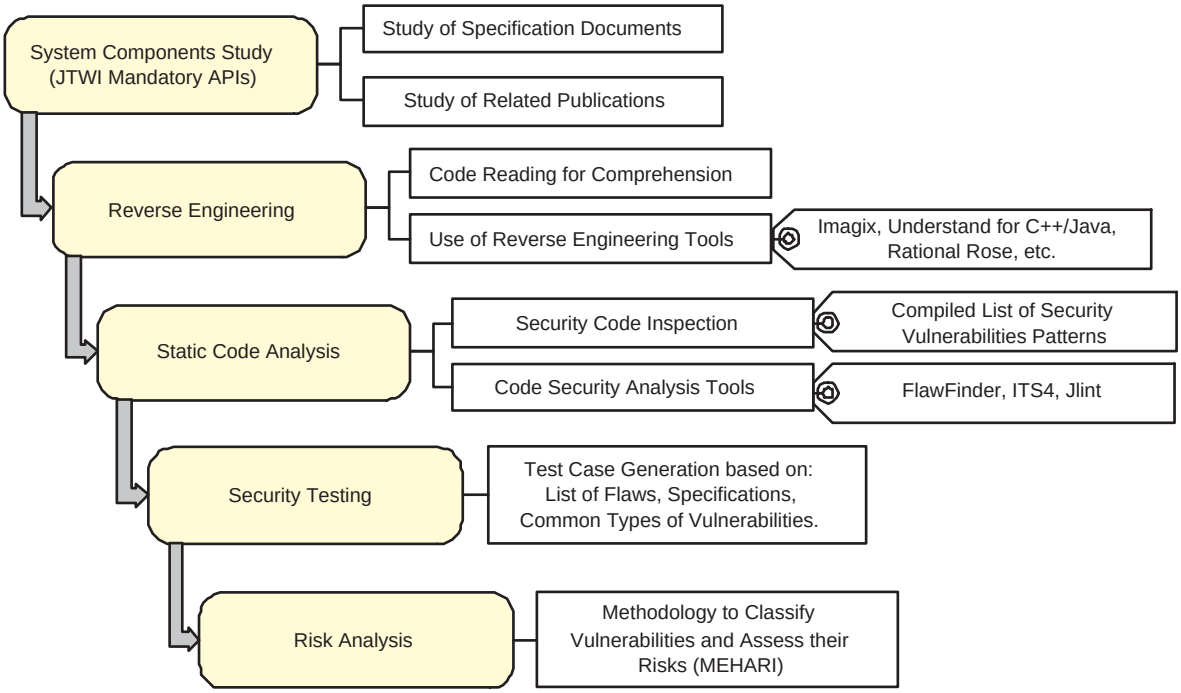

Figure 2. Methodology to Discover Vulnerabilities

guidelines of the MEHARI risk analysis methodology. As a downstream result of this phase, a reasonable and efficient set of security requirements is elaborated in order to harden the security of J2ME CLDC platform implementations.

The results presented in the following sections are the ones we obtained from phases 1 to 4.

3.3 Networking Vulnerabilities

3.3.1 MIDP SSL Vulnerability

In order to establish a secure connection with remote sites (HTTPS), the reference implementation of MIDP uses SSL v3.0 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 the set of symmetric keys for encryption. Hence, generating random values that are unpredictable is an important security aspect of SSL. It is well known that the challenge in producing good random values is how to update the seed. The seed is an initial value on which you apply a certain algorithm in order to generate random values. Generating a set of random values occurs in the following way: the current seed value is used to generate a random value, then, the seed is updated and a second random value can be generated and so on. By inspecting the Reference Implementation of SSL, we noticed that the seed update depends only on the system time (`System.currentTimeMillis`). In order to obtain a concrete proof of this insight, we fixed the system time value in the `updateSeed` method. Then, by executing the SSL handshake, we noticed that the first random value generated is always the same. Similarly, the next random values are perfectly predictable. 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.

3.3.2 Unauthorized SMS Sending Vulnerability

As every security-sensitive API, Wireless Message API (WMA), allowing the exchange of SMS messages, requires appropriate permissions to be used. This is due to charges that may result in the connection. Usually, user permission is obtained through an on-screen dialog. That is, when a program needs to send an SMS message, the device displays a dialog asking the user whether he accepts to send the SMS message and hence to assume charges. Consequently, sending an SMS message without the authorization of the user is

considered as a security flaw. As mentioned earlier, the Phenoelit hackers group [19] has discovered that the Siemens S55 phone has a vulnerability that makes the device 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 because 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 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 run the MIDlet on Siemens S55 emulator using Sun One Studio 4. The result was as we expected: the SMS authorization dialog was obscured by a different item. This makes the user think that he is answering an invitation to play a game! Since its publication and in the few documents where it is published, 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. The reason is that it blocks any modifications to the screen before an answer is received from the user. Similarly, Motorola V600 phone is not vulnerable to this attack.

3.4 Storage System Vulnerabilities

3.4.1 Managing the Available Free Persistent Storage Vulnerability

When a MIDlet needs storing 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 a given MIDlet.

If no restrictions are made on the persistent storage granted to one MIDlet, we can not prevent any MIDlet from getting all the available free space on the persistent storage for its record stores. By allowing this, all other MIDlets will be prevented from getting additional persistent storage (that can be vital for their life cycle).

The previous vulnerability was discovered in the MIDP RI as well as the Wireless Toolkits. However, it was not present in the real phones we tested (Motorola V600 and Nokia 3360).

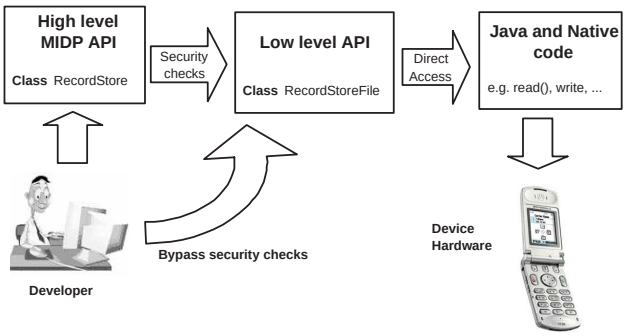

Figure 3. Storage System Vulnerabilities: *Bypassing Internal APIs protection***.**

3.4.2 Unprotected Internal APIs Vulnerability

MIDP APIs were designed in several levels of abstraction (Figure 3). The highest level contains all what a developer needs in order to develop MIDlets. 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. 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.

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.

One of the high level APIs in MIDP is the class `RecordStore`. It 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 Sun's 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.

3.4.3 Retrieving and Transferring JAR Files from a Device

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 becomes 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 [16]. 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. We installed FExplorer software on Nokia 3600 phone. 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. In Series 60 phones JAD and JAR files are typically stored in:

```
\midp\<vendor>\<domain>\<midlet_name>\
```

directory. For example, in our case JAR and JAD files of `SunSmsAttack` MIDlet which is installed on the device can be found in:

```
\midp\CSA\untrusted\SunSmsAttack\
```

directory. Then, all what remains to do is to choose "Options" and "send via Bluetooth", "SMS" or "Infrared." This operation is also possible in all Series 60 devices. These include Samsung, Siemens, Panasonic and mainly Nokia devices [16]. 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 [17]).

3.4.4 Retrieving and Transferring MIDlet Persistent Data

In addition to JAR and JAD files, using FExplorer software, it is 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 [17]. This may have a serious impact on the privacy of the MIDlet since it is possible to tamper with its persistent data.

3.5 KVM Vulnerabilities

3.5.1 Memory Overflow Vulnerability

Memory overflow is a well-known problem and may result in many security breaches. A program suffers from memory overflow vulnerability if, somewhere in the program code, it allows the copy of data to a memory location without checking the size of the saved data. Thus, memory overflow may happen if the size of the data to save is greater than the size of the memory location.

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

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

`sprintf` is a C function that does not check the size of the data to format in a memory location. Thus, the statement will not check the size of the message that will be formatted in `str_buffer`. Knowing that `str_buffer` is a global variable declared as:

```
char str_buffer[512];
```

In our analysis, we arranged to build a MIDlet that takes advantage of this vulnerability and make MIDP crash. This MIDlet is tested for some real phones making MIDP crash.

3.6 Threading System Vulnerabilities

J2ME CLDC supports multithreading, the threading system was analyzed and vulnerabilities were discovered.

3.6.1 Threading and Storage System Vulnerability

Although multi-threading is supported, no measures were taken to synchronize access to the storage system. When two or more threads attempt to read or write data to/from the storage system, data integrity can not be guaranteed. Synchronization is left as the programmer's responsibility. A malicious MIDlet could make use of this fact to corrupt the data belonging to another MIDlet (in case of shared data). Moreover, integrity of the data stored by a MIDlet in its own storage can be compromised in case of several threads trying to read and write data.

3.6.2 Threading and Display Vulnerability

The method `setCurrent` of the class `Display` is responsible for setting the display of a certain MIDlet to a certain `Displayable` object such as a `TextBox`. For instance, the code:

```
Display.getDisplay(this).setCurrent(tb);
```

will display the `TextBox` object `tb` on the device screen. This method, however, is not synchronized, which makes it up to the programmer to synchronize the display for use between different threads. This can cause problems and unless *all* threads use synchronized access to the display, some threads may not get access to the display.

4 Conclusion and Future Work

In this paper, we presented the security architecture of J2ME CLDC and provided a vulnerability analysis of this Java platform. In our analysis, we investigated both, specifications and implementations (the reference implementation as well as several other widely-deployed implementations of this platform). The J2ME CLDC components covered by the analysis are mainly: Virtual machine, CLDC API and MIDP API. We performed the vulnerability analysis by using mainly code inspection and black box testing of the reference implementation. To have a reliable analysis, we

studied vulnerabilities that are already reported in the literature. In our study, we investigated the existing implementations of the platform in order to look for vulnerabilities. Also, we designed an important set of attack scenarios and tested the resulting test suite on reference implementation and actual phone models.

After the study that we carried out, The following two points were made clear through the analysis of J2ME CLDC security:

- Serious vulnerabilities exist in the reference implementation of MIDP 2.0 (e.g. SSL implementation).
- Some phones could be vulnerable to serious 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, modifications can be done to improve J2ME CLDC security. The security hardening can be performed by following two main paths: (1) Fixing the discovered vulnerabilities by suggesting modifications to the current security architecture, (2) Extending the security architecture by new security functions.

References

[1] M. Bishop. Vulnerabilty Analysis. In *Proceedings of of the Second International Symposium on Recent Advances in Intrusion Detection*, pages 125–136, Sptember 1999.

[2] C. Criteria. Common Criteria for Information Technology Security Evaluation (Parts 1, 2 and 3). Technical report, The Common Criteria Project, August 1999.

[3] A. Dunsmore, M. Roper, and M. Wood. The Development and Evaluation of Three Diverse Techniques for Object-Oriented Code Inspection. *IEEE transactions on software engineering*, 29(8), 2003.

[4] G. Fink and M. Bishop. Property Based Testing: A New Approach to Testing for Assurance. In *ACM SIGSOFT Software Engineering Notes*, pages 74–80, July 1997.

[5] G. Bracha, T. Lindholm, W. Tao and F. Yellin. CLDC Byte Code Typechecker Specification. `http://jcp.org/aboutJava/communityprocess/final/jsr139/index.html`, January 2003.

[6] I. Goldberg and D. Wagner. Randomness and the Netscape Browser. *Dr. Dobb's Journal of Software Tools*, 21(1):66, 68–70, Jan. 1996.

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

[10] I. Krsul. *Software Vulnerability Analysis*. PhD thesis, Purdue University, 1998.

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

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

[13] MEHARI. MEHARI. Technical report, Club de la Securite des Systemes d'information Francais, August 2000.

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

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

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

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

[18] J. V. Peursem. JSR 118 Mobile Information Device Profile 2.0, November 2002.

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

[20] R. Riggs, A. Taivalsaari, M. VandenBrink, and J. 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.

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

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

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

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

[25] 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.

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

[27] H. H. Thompson, J. A. Whittaker, and F. E. Mottay. Software Security Vulnerability Testing in Hostile Environments. In *SAC '02: Proceedings of the 2002 ACM symposium on Applied computing*, pages 260–264, New York, NY, USA, 2002. ACM Press.

[28] J. Viega, J. Bloch, Y. Kohno, and G. McGraw. ITS4: A Static Vulnerability Scanner for C and C++ Code. In *ACSAC 2000*, 2000.

[29] J. Viega, G. McGraw, T. Mutdosch, and E. Felten. Statically Scanning Java Code: Finding Security Vulnerabilties. *IEEE Software*, September/October 2000.