

Security Analysis of Wireless Java

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

Abstract

This paper represents a careful study of the security aspects of Java 2 Micro-Edition (J2ME) Connected Limited Device Configuration (CLDC). This Java platform is intended for resource-constrained devices, with the purpose of expanding the range of applications available for these devices, by enabling them to run Java programs. Consequently, malicious code would pose a threat to the security and safety of the devices. The purpose of this study is to identify vulnerabilities in the platform that could be exploited by malicious code. All of this is done with the final goal of contributing to the security of the Java platform, which would make way for more growth in application development for mobile devices.

Keywords: J2ME CLDC, Wireless Java Security, Mobile Code Security, Security Analysis, Vulnerability Analysis, Security Testing.

1. Introduction

There is an ever growing number of mobile devices that support Java applications. The latest (June 2004) list of mobile phones supporting Java 2 Micro-Edition -Connected Limited Device Configuration- (J2ME CLDC) with the Mobile Information Device Profile (MIDP) version 2.0, shows 60+ phone models from various manufacturers. According to IDC, a prestigious market research firm, there will be more than 1.2 billion deployed Java-based mobile devices by 2006. Java applications bring advanced functionalities to the mobile world. Moreover, they have the added advantage of being device-independent, so the same application could be run on various models of phones. Also, there is a large number of available Java programmers, their experience can be invested in the mobile market. All of these factors contribute to the current growing popularity of Java-enabled phones. Device manufacturers are motivated by the added functionalities Java can bring to their devices. Fur-

thermore, applications for the devices can now be easily developed by third parties with Java programming experience.

With the large number of applications that could be available for Java-enabled devices, security is definitely an issue. Applications can handle user-sensitive data such as phonebook data or bank account information. Moreover, J2ME CLDC supports networking, which means that applications can also create network connections and send or receive data. This paper represents an attempt to carefully study the security aspects of J2ME CLDC (and MIDP) with the purpose of providing a security evaluation for this Java platform.

In this regard, two different paths are followed. 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 model, 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, 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. We present here the results of our security analysis in the form of security vulnerabilities. This is a necessary step for any research effort aiming at hardening the security of J2ME CLDC platforms.

This paper is organized into five sections beginning with the introduction. In Section 2, we present the related work concerning J2ME CLDC security model and its evaluation. In Section 3 the security architecture as presented in the specifications is studied, followed by a vulnerability analysis in Section 4, which describes our approach and presents the main steps taken in the course of the study. The framework and results of the study are given in this section. Finally Section 5 concludes the paper.

2. Related Work

This section gives an overview of what has been published in the literature regarding J2ME CLDC security. The papers and articles can be classified in two categories. First, papers that provide tutorials on J2ME CLDC security model and try to evaluate that model. Second, papers that focus on building security into the application layer (within the MIDlets themselves).

The first category of papers, which is the most encountered in the literature, usually contains 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 the following we give an overview of the most relevant efforts in this category.

In [19], Kolsi and Virtanen, try to evaluate the new security features of MIDP 2.0. They start 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 conclude that several security problems are addressed by MIDP 2.0, but some problems are still present in MIDP 2.0 security, mainly related to the Public Key Infrastructure (PKI). These problems are illustrated in [6].

In [3], Sanjay Chadha discusses mainly two security concerns about J2ME applications. 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. This is not sufficient in the wireless world since it is already too late if a malign application is delivered to a mobile device. Therefore, 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 Right Management) support in J2ME. For example, most users prefer to try an application before making a purchasing decision, which would be possible by a DRM implementation.

Also [33] 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. 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. 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, the feasibility of developing advanced secure applications is discussed in the context of the smallest wireless devices using J2ME technologies. This article covers all J2ME security model including CLDC and CDC whereas in the current paper we are interested only in J2ME CLDC security. It should be noticed also that this article is not very recent (2002) and some issues discussed by the authors were addressed by new JSRs. For example, JSR 172 provides basic XML procession capabilities to J2ME.

The second category of papers discusses security solutions that do not rely on protocols at lower layers. In other words, the security related functions are implemented in the application (MIDlet) itself.

In [13], Itani and Kyassi propose an end-to-end application layer security solution for wireless enterprise applications using J2ME and J2EE. The solution they propose handles all the security-related functions at the application-layer without relying on lower-layer protocols. It uses Advanced Encryption Standard (AES) Rijndael symmetric block cipher algorithm to provide authentication and confidentiality between a mobile user and a server. A mobile banking application is implemented to illustrate the proposed solution. Although the paper is relatively recent (2003), the authors based their work on MIDP 1.0 and did not consider MIDP 2.0 that comes with security enhancements, in particular, an implementation of SSL protocol. Sun's implementation of SSL [11] (KSSL) is criticized as it assumes the presence of TCP which is not always true for J2ME devices. Their proposed implementation differs from KSSL by supporting client authentication.

It is well known that SSL is too heavyweight and memory-intensive to be used in mobile commerce. Thus Sun has developed a light-weight SSL implementation called KSSL [11]. MIDP 2.0 implementation of SSL is a modified version of KSSL. Another interesting cryptography package for J2ME is proposed in the Bouncy Castle project [29]. In his book [18], Knudsen dedicated the last chapter (chapter 12: Protecting Network Data) to be a tutorial on Bouncy Castle APIs.

The same author, Knudsen, published a series of four articles about building security into wireless Java applications (MIDlets). The first article [14] provides a general overview of secure system design and it acts as an introduction to the subsequent articles. The second one [15] is a tutorial on using SSL and TLS from MIDlet clients. The third article [16] shows how to implement authentication in MIDlets. Finally, the last article [17] describes how encryption can be implemented in MIDlets. It is worth mentioning that the examples given in the third and fourth articles make use of

the Bouncy Castle Cryptography API.

3. J2ME CLDC Security Architecture

In this section, we present the security architecture of J2ME CLDC. We distinguish between the security of the Connected Limited Device Configuration (CLDC) and the security of the Mobile Information Device Profile (MIDP). The reason behind this distinction is that security concerns are divided between the two. This will become apparent in the following discussion.

Applications developed for the J2ME CLDC Java 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 that contains the following files:

- JAR manifest: This is a text file that contains various attributes like the MIDlet name and vendor.
- Class files: These are the *preverified* class files of the MIDlet (preverification will be discussed later).
- Supporting files: Any other files needed by the application like graphic files for instance.

A JAR file can contain more than one application (MIDlet). This group of MIDlets is called a MIDlet suite. The JAD on the other hand, is a text file with several attributes like MIDlet name and the MIDP version needed to run the MIDlet.

3.1. CLDC Security

The security of CLDC is affected by the absence of some general Java features that have been dropped because of performance and security issues. This is shown in the following list:

- *No Java Native Interface (JNI)*
- *No User-defined class loaders*
- *No support for Reflection*
- *No Thread groups or daemon threads*

Low level security in CLDC is based, in general, on type safety mechanisms. The verifier is the module in charge of type checking. Because full bytecode verification is too heavy for resource-constrained devices, in CLDC, verification has been changed so that classes of one application are first pre-verified [9] on the development platform. Then, the virtual machine will only do limited verification of the application classes before executing the application on the target device.

3.2. MIDP Security

In this section, 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 more capabilities of the device to MIDlets and provides the needed mechanisms to control the use of these capabilities.

3.2.1 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* 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. This offers a good protection for the MIDlet's persistent storage.

Concerning connectivity protocols, the only network protocol provided in MIDP 1.0 is the HTTP protocol.

3.2.2 MIDP 2.0 Security

The difference between MIDP 1.0 security model and MIDP 2.0 security model is that, in MIDP 2.0, accessing the 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 in the device to one protection domain. A MIDlet is bound to a protection domain according to a well defined procedure that allows the Application Management System (AMS) to authenticate the origin of a MIDlet and identify the protection domain the MIDlet will be bound to. If one MIDlet can be authenticated, then it is qualified as *trusted*, otherwise, it will be qualified as *untrusted*.

Moreover, MIDP 2.0 introduces the ability to share record stores between MIDlet suites. Also, an important difference between the security of MIDP 1.0 and MIDP 2.0 is that the latter provides end-to-end security by allowing secure networking using HTTPS protocol.

• Sensitive APIs

Security-sensitive APIs are the ones that act as an interface between MIDlets and security-sensitive resources of the device. These APIs are protected by permissions such that no MIDlet can use them unless it belongs to a protection domain that is granted the necessary permissions. The protected APIs are all the APIs related to networking in addition to the `PushRegistry` class.

- **Trusted and Untrusted MIDlet Suites**

When downloading and installing a MIDlet, the device has to decide whether the MIDlet will be trusted or not. If the MIDlet is trusted then, it will be bound to a protection domain and will be granted the permissions specified in that protection domain. If the MIDlet is not trusted, it will be bound to the untrusted domain which is provided a limited set of permissions.

Figure 1 illustrates the use of a certificate chain to give the device the ability to authenticate the origin of a MIDlet suite and to bind the MIDlet suite to the appropriate protection domain. In this scenario, the device first, checks the JAR file signature to be sure that the JAR file was not tampered with, then validates the certificate chain to authenticate the origin of the MIDlet. If the MIDlet origin is authenticated and the certificate is valid then the device will install the MIDlet and bind it to the protection domain associated with the authenticated certificate authority (Cer.F in the example of the figure).

- **Secure Networking**

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 the certificate chain provided by a server.

- **Persistent Storage Security**

As mentioned earlier, in MIDP 2.0 a MIDlet suite can save data in the device persistent storage. The storage unit is called a record store, and each record store contains a numbers of records. Records are mainly arrays of bytes. With MIDP 2.0, a MIDlet can choose to share one or more of its record stores with other MIDlets. The sharing mode can be read-only or read-write. MIDP 2.0 provides measures to enforce the sharing rules so that no MIDlet can do an unallowed operation on a record store belonging to another MIDlet.

3.3. Security and Trust Services API

“Security and Trust Services API” (SATSA) [1], is a new API that provides additional security capabilities to the J2ME CLDC platform. It specifies a collection of APIs that provide security and trust services for J2ME CLDC by integrating a Security Element (SE). The SE is a hardware or a software component in a J2ME device. It provides the following features [1]:

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

With these features, J2ME applications would be able to have secure key stores as well as encryption and decryption capabilities. These features could be used to provide

security services for applications such as e-payments, mobile commerce, etc. A SE can be: (1) deployed as a smart card in wireless phones or, (2) can be implemented by a handset itself (e.g., embedded chips or special security features of the hardware) or, (3) may be entirely implemented in software. Although, some SATSA APIs packages are optimized for smart card implementations, the specification of SATSA API (JSR 177) does not exclude any of the possible implementations of a SE.

4 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 certain system with the purpose of detecting security related errors. These errors may come from faults in the system specifications or the system implementation. The presence of an error weakens the security structure of the system making it vulnerable to attacks. In this section, we describe first our approach to detecting security vulnerabilities, then we present some of the previously reported security flaws in the literature. Finally, we present the results of our own security investigation.

4.1 Approach

There are not many references that describe how to conduct a security analysis of a software system. In our opinion, this is due to the fact that security concerns are addressed differently at various levels. These include operating system security, network security, middleware security and application security. Even in each of these areas, security is almost always addressed with respect to a specific operating system (e.g., Unix, Windows), network design (e.g., TCP/IP) or programming language (e.g., C/C++, Java). It is noted however that 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 [20] and Bishop [2]. 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

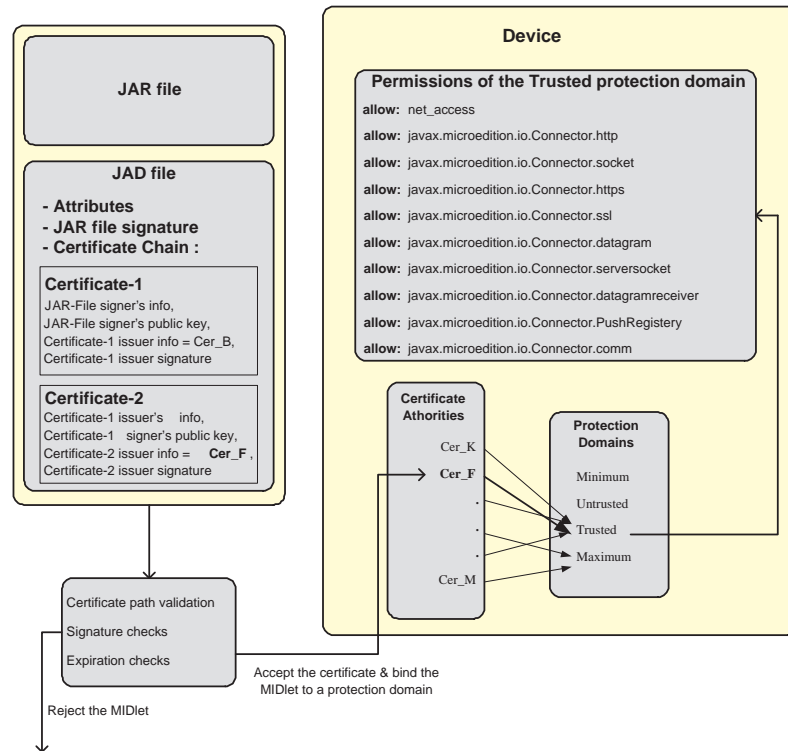


Figure 1. Trusting a Signed MIDlet and Binding it to a Protection Domain

common security-related errors or known unsafe function calls (e.g., function `strcpy()` in C/C++ is vulnerable to buffer overflow). In [32], static analysis for Java is presented together with a tool for the same purpose, whereas [31] presents a tool for C/C++ code.

In security testing, techniques of property-based testing [7] 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 [30], 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 [4]

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

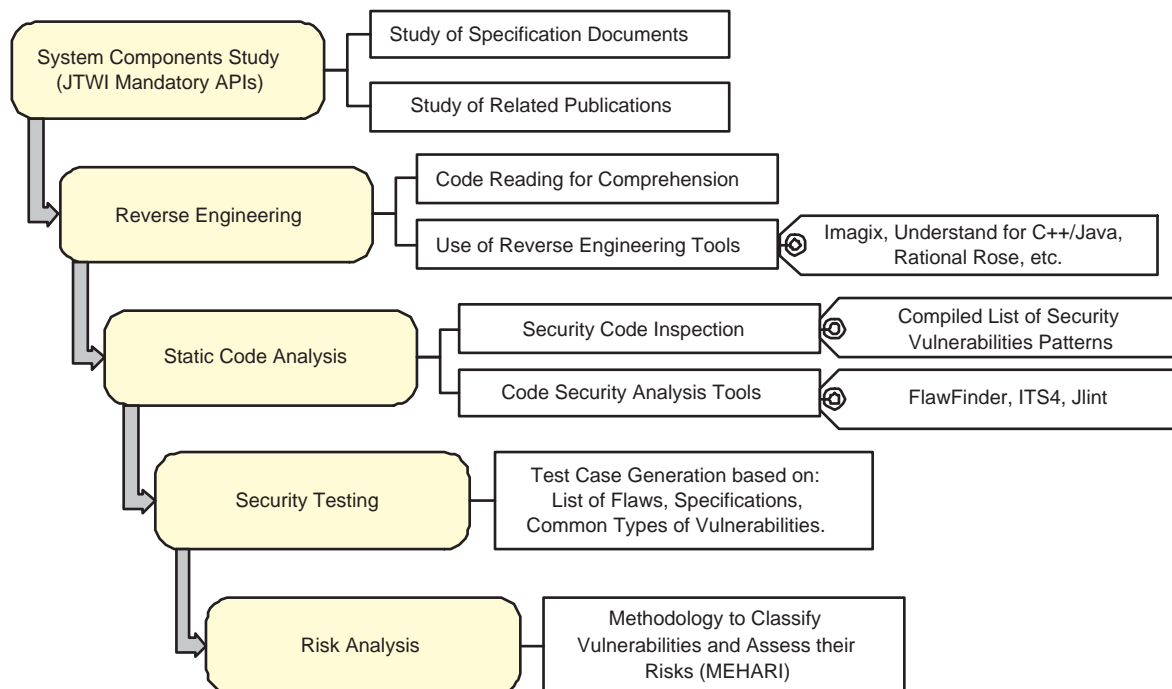


Figure 2. Methodology to Discover Vulnerabilities

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 [5]. 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 [31] for C, and Jlint [32] 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 [20] and (3) the security properties that are extracted from the specification documents according to property-based testing principles [7]. 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 [21] 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 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.

4.2 Previously Reported Flaws

Very few security flaws have been reported in the J2ME CLDC platform. The most serious one is the Siemens S55 SMS flaw. Besides, several problems about Sun's MIDP reference implementation have been reported.

4.2.1 Vulnerabilities in Siemens S55

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 is due to a race condition during which the Java code can overlay the normal permission request with an arbitrary screen display.

4.2.2 Vulnerabilities in Sun's MIDP Reference Implementation

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:

1. When establishing a socket connection (`socket://hostname:portnumber`), permissions are needed. But if one runs the RI on a platform, where `portnumber` is already occupied, the RI does not check for permission [28]. Instead, it throws an `IOException`. This is not correct because there is no need to access native sockets if the MIDlet does not have 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 was bypassed.
2. A problem has been reported in the implementation of the RSA algorithm. The big number division function checks the numerator instead of the divisor for zero [26].
3. Basic authentication is not fully supported in Sun's RI [27]. According to MIDP 2.0 specification, the device "must" be capable of responding to a 401 (Unauthorized) or 407 (Proxy Authentication Required) by asking the user for a username and password and responding the HTTP request with the credentials supplied. The device must be able to support at least the

RFC 2617, which corresponds to the basic authentication scheme [8]. However, the RI utilizes basic authentication only in the case of MIDlet suite installation (to retrieve JAD and JAR files).

4. The return value of `midpInitializeMemory()` method called in `main()` is never checked [25]. When memory allocation fails, the system will crash without any way to figure out the reason of this crash.

4.3 Vulnerability Analysis Results

In this section, we present in a structured way the results of the vulnerability analysis that we performed on the J2ME CLDC platform. For the sake of clarity, we organize the vulnerabilities according to the platform component in which they were discovered.

4.3.1 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 [11] from Sun Labs. During the SSL handshake, the protocol has to generate random values to be used to compute the master secret. The latter is then used to generate the set of symmetric encryption keys. Hence, generating random values that are unpredictable is an important security aspect of SSL. The method `PRand.generateData` is used in MIDP to generate pseudo-random data. This method computes the pseudo-random values progressively 16 bytes by 16 bytes. The first part is computed by applying a hash algorithm (MD5) on the seed. Afterwards, the first part is copied into the pseudo-random value array. The seed is then updated and the hash algorithm is applied a second time to generate the second part. This operation continues until the pseudo-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 pseudo-random value. Consequently, the challenge is: How to update the seed in an unpredictable fashion?.

The method used in the reference implementation 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);
}
```

Here, 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 pseudo-random value computation. To this end, sniffing tools can be used. This allows the attacker to guess a narrow interval of the correct system time. Afterwards, it remains only 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 [10] used sniffing tools to determine the seconds part of the system time. Then, to find the microseconds part, every possible value of the 1 million possibilities is tried. To sum up, the current implementation of the pseudo-random number generator makes the SSL protocol vulnerable.

• 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 is 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 under the assumption 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 below. 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 to 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. Sun reference implementation of MIDP is not vulnerable to this attack. Indeed, when the device asks the user for permission, MIDP RI prevents any modification to 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.

4.3.2 Storage System Vulnerabilities

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 unique. The actual structure of the record store on the device 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 3 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, the sharing of record stores is allowed. The MIDlet suite that created the record store can choose to share it or not. Moreover, the sharing mode can be set either 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). Detailed analysis of the RMS using MIDP specifications and Sun's reference implementation 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 backup device. Moreover, the whole storage system in

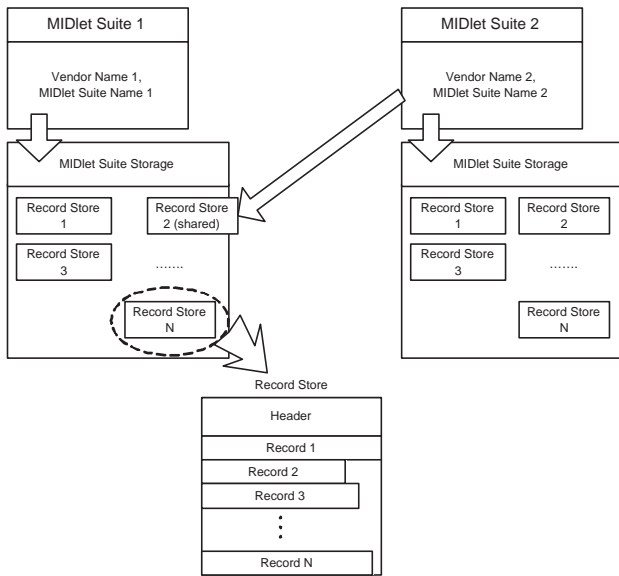


Figure 3. Record Stores in MIDP.

MIDP can be accessed from any file browsing application on the device. An example of such access can be performed using the FExplorer software, which is an application for Series 60 Nokia phones. It is worth noting that the SATSA API provides tools that can be used to protect data by means of cryptographic operations. However, such a protection is not part of the standards and is left to the applications.

• **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 MIDlet. This means that one cannot prevent a MIDlet from getting all the available space on the persistent storage of the device. 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 reference implementation as well as in the wireless toolkits.

• **Unprotected Internal APIs Vulnerability**

MIDP APIs provide the capabilities (methods) that are needed by MIDlet programmers in order to develop mobile applications. However, these are high-level APIs, which are designed to ease programming tasks. Therefore, they use helper low-level APIs that call native methods to deal with the device hardware. These low-level APIs have more privileges and less restrictions when dealing with the de-

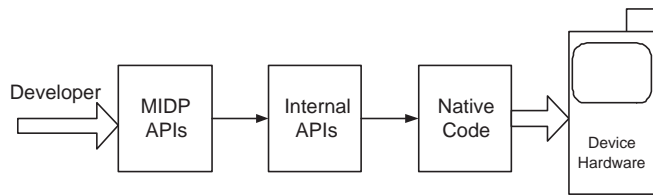


Figure 4. Various Levels of Abstraction in APIs for Mobile Devices.

vice hardware. Accordingly, it is crucial from the security standpoint to manage that only high-level MIDP APIs could access these low-level privileged methods. In other words, developers should not have access to these low-level APIs. We discovered that the reference implementation of J2ME CLDC does not enforce this security measure. Figure 4 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 enforce data security and integrity. For instance, no MIDlet is allowed to delete a record store of another MIDlet. There is another low-level class, RecordStoreFile, which is closer to the device hardware. It calls native methods and provides services to the RecordStore class. The RecordStoreFile class should not be available for direct use by developers, because it has more access rights and bypasses the security checks. In Sun’s reference implementation, 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 deletes a record store belonging to another MIDlet.

First, we developed a MIDlet `rmstest` 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 to create the record store, which

was then deleted by the attacking MIDlet.

• Retrieving and Transferring JAR Files from a Device

A typical scenario for downloading a MIDlet is to connect to a mobile application provider. Once a MIDlet is installed on the device, the user should be able to perform two kinds of operations, namely, execution and un-installation of the MIDlet. If, in addition, the user has the capability to transfer the MIDlet to another device, this might result into a breach of the intellectual property rights of the MIDlet provider. In our experiments, we succeeded to transfer MIDlets from one device to another, using an application available for Series 60 phones [22]. For instance, the FExplorer software [12] makes it possible to navigate through files and MIDlets installed on the device. It also provides options to send these files to other phones. It is important to note however that MIDlets protected by Digital Right Management (DRM) cannot be transferred from one device to the other (protection should be at least in the forward lock mode [23]).

• Retrieving and Transferring MIDlet Persistent Data

Using FExplorer software, it is possible to transfer MIDlet persistent data from one device to another. Indeed, on Nokia 3600 phone, the `rms.db` file that holds all MIDlet persistent data is in the same location as JAD and JAR files and can be transferred following the same steps. Moreover, DRM protection does not cover `rms.db` files. Even if the MIDlet is DRM protected, the `rms.db` file can be transferred because DRM protection holds only for JAR files [23].

4.3.3 KVM Vulnerabilities

• Buffer Overflow Vulnerability

Buffer overflow is a well-known problem and may result in many security breaches. It occurs when the application does not perform bounds checking on data before copying it into a buffer. This can happen for instance when an application tries to overwrite a certain buffer with data from a larger buffer using the `strcpy` function (for C code). In this 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 this address, the attacker will be able to execute the code that he wants. By inspecting the source code of KVM, we identified a memory overflow vulnerability. The vulnerable code in `native.c` is the following :

```
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 an

array of 512 characters, it is clear that the code might result in a strange behavior if the size of the string to be stored in it exceeds 512 characters. On the Motorola V600 phone, the virtual machine crashes when the buffer in question is overflowed. One part of this string is the name of the invoked native method. Since no restrictions are imposed by the virtual machine on the size of method or field names, we wrote a simple Java program that declares a native method name counting 2000 characters. 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 overflows `str_buffer` causing the overwriting of 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...(); }
```

• Native Methods vulnerability

In J2ME CLDC, there is no support for a native interface as is specified in the J2SE/EE platforms. Dealing with native methods is delicate and any Java platform must provide strong security mechanisms before allowing user applications to define and run native methods. Since the implementation of these mechanisms (like the Java Native Interface of J2SE) is too heavy to fit in the J2ME CLDC platform, MIDlets are not allowed to define and implement native methods. Although, the specification of J2ME CLDC is clear about this point, the implementations investigated do not respect the specification and we succeeded to install MIDlets declaring native methods in all J2ME CLDC implementations that we tested during this work. Thus, we used this vulnerability to carry out a buffer overflow attack as shown earlier. This vulnerability is common for all J2ME CLDC implementations, including the reference implementation, the emulators, and the actual phones. Fortunately, only the signature of a native method can be declared in a MIDlet and there is no way to include a native implementation in a MIDlet suite and have the possibility of running it on the device.

5. Conclusion and Future Work

In this paper, we presented a security evaluation of J2ME CLDC. We started by presenting the security architecture followed by our vulnerability analysis of the platform. We investigated the reference implementation, phone emulators

and actual phones. The discovered vulnerabilities were presented and classified according to the system component that they affect. Also, an important result of this work is the design of a test suite that can be used to evaluate any implementation of J2ME CLDC.

With this study in hand, modifications can be done to improve J2ME CLDC security, by suggesting modifications to the security model, and by providing a clear set of security functions to be included in any implementation of the platform in order to achieve the security goals.

References

- [1] S. Ahmad. JSR 177 Security and Trust Services API for J2ME, September 2004.
- [2] M. Bishop. Vulnerability Analysis. In *Proceedings of the Second International Symposium on Recent Advances in Intrusion Detection*, pages 125–136, September 1999.
- [3] S. Chadha. J2ME Issues in the Real Wireless World. http://www.microjava.com/articles/perspective/issues?content_id=4323, January 2003.
- [4] C. Criteria. Common Criteria for Information Technology Security Evaluation (Parts 1, 2 and 3). Technical report, The Common Criteria Project, August 1999.
- [5] 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.
- [6] C. Ellison and B. Schneier. Ten Risks of PKI: What You're Not Being Told About Public Key Infrastructure. *Computer Security Journal*, 16(1):1–7, 2000.
- [7] 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.
- [8] 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.
- [9] 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.
- [10] I. Goldberg and D. Wagner. Randomness and the Netscape Browser. *Dr. Dobbs's Journal of Software Tools*, 21(1):66, 68–70, Jan. 1996.
- [11] 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.
- [12] D. Hugo. FExplorer Web Site. <http://users.skynet.be/domi/fexplorer.htm>.
- [13] W. Itani and A. Kayssi. J2ME Application-Layer End-to-End Security for m-Commerce. *Journal of Network and Computer Applications*, 27(1):13–32, January 2004.
- [14] J. Knudsen. MIDP Application Security 1: Design Concerns and Cryptography. <http://developers.sun.com/techttopics/mobility/midp/articles/security1/>, September 2002.
- [15] J. Knudsen. MIDP Application Security 2: Understanding SSL and TLS. <http://developers.sun.com/techttopics/mobility/midp/articles/security2/>, October 2002.
- [16] J. Knudsen. MIDP Application Security 3: Authentication in MIDP. <http://developers.sun.com/techttopics/mobility/midp/articles/security3/>, December 2002.
- [17] J. Knudsen. MIDP Application Security 4: Encryption in MIDP. <http://developers.sun.com/techttopics/mobility/midp/articles/security4/>, June 2003.
- [18] J. Knudsen. *Wireless Java: Developing with Java 2, Micro Edition, Second Edition*. Books for professionals by professionals. Springer-Verlag, Berlin, Germany / Heidelberg, Germany / London, UK / etc., February 2003.
- [19] 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.
- [20] I. Krsul. *Software Vulnerability Analysis*. PhD thesis, Purdue University, 1998.
- [21] MEHARI. MEHARI. Technical report, Club de la Securite des Systemes d'information Francais, August 2000.
- [22] Nokia. Series 60 Platform. <http://www.nokia.com/nokia/0,8764,46827,00.html>.
- [23] OMA. Implementation Best Practices for OMA DRM v1.0 Protected MIDlets, May 2004.
- [24] Phenoelit Hackers Group. <http://www.phenoelit.de/>, 2003.
- [25] Bug 4824821: Return value of midpInitializeMemory is not checked. http://bugs.sun.com/bugdatabase/view_bug.do?bug_id=4824821, February 2003.
- [26] Bug 4959337: RSA Division by Zero. http://bugs.sun.com/bugdatabase/view_bug.do?bug_id=4959337, November 2003.
- [27] Bug 4963644: Basic Authentication Scheme is not fully supported. http://bugs.sun.com/bugdatabase/view_bug.do?bug_id=4963644, December 2003.
- [28] Bug 4802893: RI checks sockets before checking permissions. http://bugs.sun.com/bugdatabase/view_bug.do?bug_id=4802893, January 2004.
- [29] The Legion of the Bouncy Castle. Bouncy Castle Cryptography API. <http://www.bouncycastle.org>, 2004.
- [30] 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.
- [31] J. Viega, J. Bloch, Y. Kohno, and G. McGraw. ITS4: A Static Vulnerability Scanner for C and C++ Code. In *ACSAC 2000*, 2000.
- [32] J. Viega, G. McGraw, T. Mutdosch, and E. Felten. Statically Scanning Java Code: Finding Security Vulnerabilities. *IEEE Software*, September/October 2000.
- [33] M. J. Yuan and J. Long. Securing wireless J2ME. <http://www-106.ibm.com/developerworks/java/library/wi-secj2me.html>, June 2002.