J Comput Virol
DOI 10.1007/s11416-006-0014-0

ORIGINAL PAPER

The Java mobile risk

Daniel Reynaud-Plantey

Received: 3 January 2006 / Accepted: 5 May 2006

Abstract Mobile security is a relatively new field of research. The specific risks and counter-measures have not been studied thoroughly yet. Java can be useful in this field because it has been designed to run untrusted code safely on mobile, heterogeneous networks. This paper, based on an experimental study of the low-level security of Java 2 Micro Edition (JSR 118 Expert Group, 2006), identifies some potential problems and their solutions in case Java malicious applications appear in the wild.

1 Introduction

Java technology is particularly interesting in mobile environments because of its built-in security features and portability. These features include downloading over the air and running untrusted code in a safe environment, as well as end-to-end security with networking and cryptography APIs.

This paper focuses on the practical aspects of the risks associated with Java 2 Micro Edition. Though the architecture is roughly the same as the standard edition, there are differences that induce specific problems. An experimental study of J2ME low-level security has been conducted on a Series 60 phone in order to identify and evaluate some of these risks.

After a quick introduction to J2ME, configurations, profiles and the new verification scheme, the methodology of the experimental study will be explained. Then, Sect. 5 underlines some problems that are seldom mentioned for the user. Finally, Sect. 6 underlines a few problems that analysts have to consider in order to disassemble a midlet correctly.

D. Reynaud-Plantey (✉)
Ecole Supérieure et d'Application des Transmissions,
Laboratoire de virologie et de cryptologie,
B.P. 18, 35998 Rennes Armées, France
e-mail: daniel.reynaud-plantey@esat.terre.defense.gouv.fr

2 Quick Introduction to J2ME

This section briefly explains what is J2ME compared to J2SE and J2EE and how it is organised.

2.1 Different platforms, different flavours

Java was supposed to be a kind of universal language, able to run on smart cards as well as on mainframes. As obviously you do not expect the same features from such different devices, the Java technology has to adapt to what people expect from it and to the capabilities of the device it is running on. For example, a Java program must be able to run for days without rebooting on an enterprise server. But on a workstation, you have different expectations: you probably want to use short term applications with an efficient graphical user interface. That's why Java comes in different flavours:

- Java 2 Enterprise Edition (J2EE), for running scalable enterprise applications;
- Java 2 Standard Edition (J2SE), aimed at workstations;
- Java 2 Micro Edition (J2ME), for mobile devices;
- Java Card, for smart cards.

For the Java developer, it means two things: with a single language, you can write applications for every platform.

But it also means that you will have to write code for a given platform, and it usually means that you may have to use different language constructs (for example you might not perform floating point operations on a smart card), and you will definitely have to use different APIs. This might or might not be a big deal, depending on what you want to do exactly. From a security perspective, the broad range of devices might sound appealing to malicious programmers but they will have to face another drawback of the Java technology: the lack of low-level control (such as pointer arithmetic or OS-specific features).

3 A closer look at J2ME

3.1 Configurations

So far, we have seen that J2ME was intended to bring Java to mobile devices. These devices include mobile phones and PDAs as well as washing machines, TV set-top boxes or car electronic equipment. What do they all have in common? They are mobile, which means they have a limited amount of memory and computing power. All their other characteristics may vary, some of them might need batteries, some might have a good bandwidth or not, and some might not even have a screen. As a consequence, Java needs again to be adapted to different kinds of mobile devices, depending on their capabilities. This is the role of a configuration [4]: provide the basic set of functionalities for a specific group of devices with similar characteristics, such as the total amount of memory and network connectivity. Therefore, **a configuration provides a virtual machine and a set of core classes**.

Two configurations have been defined:

- Connected, Limited Device Configuration (CLDC) [10]: it is the configuration currently in use on mobile phones and some PDAs. It is defined in the Java Specification Request (JSR) 30.
- Connected Device Configuration (CDC): for devices with more memory and processing power. From the developer point of view it is closer to J2SE.

3.2 Profiles

We have seen that J2ME had to be split into configurations in order to fit on different devices. But even similar devices like mobile phones might offer different ranges of memory and processing power. They also might have more to offer in terms of underlying user interface, or advanced features like Bluetooth. This is why configurations are not enough and as a result profiles have been defined. **A profile is a set of APIs which sit on top of a given configuration** in order to take better advantage of the features of a device.

Six profiles have been defined, but only one is a CLDC profile: Mobile Information Device Profile (MIDP). It is made to take advantage of the capabilities of the latest mobile phones, and its APIs cover user interfaces, connectivity, data storage, messaging and gaming. Profiles only address security issues at the application and end-to-end levels. For instance, they can provide cryptography APIs or features such as access to the file system.

3.3 J2ME low-level security

The main difference between J2SE and J2ME from the security approach is that **J2ME introduces a new verification algorithm**. A Java program is subject to some restrictions: it cannot explicitly deallocate memory, access a specific memory location, or interact directly with the OS. To ensure that a Java program can't perform these operations, it is first loaded in memory and then verified. The verification process is a very complex topic and if it is not made correctly the whole security architecture collapses. The new verification algorithm in J2ME comes along with an important modification in the class file format specification: a new attribute called StackMap has been defined. Each method must contain a StackMap attribute which makes the verification process faster and less memory-consuming. Basically, there must be a stack map at the beginning of each basic block in a method indicating the number and the types of the objects on the stack for this basic block. The former algorithm consisted in inferring and then checking type safety, with StackMap attributes the inference step can be skipped.

From the developer's point-of-view, it means that additional steps are required before a program can be run. After the compilation, the classes must be preverified (by the developer) and packed in a jar file. The so-called preverification step adds the StackMap attributes along with some other modifications. The name is quite deceptive however, because **it is actually an additional compilation more than a verification**. The real verification can only take place on the virtual machine of the user.

There are some other important differences between J2ME and J2SE from the security perspective, for example in J2ME there is no support for custom classloaders and for the Java Native Interface (JNI). These two features are very useful for virus writers and are almost needed for high-level, sophisticated viruses.

4 Experimental study

This section is a quick overview of the experimental study which led to this paper [9]. The goal of this study was to test the low-level security of a J2ME implementation on a telephone.

4.1 Testing strategy

There are two major strategies for testing software: white-box (or structural) and black-box (or behavioural) testing. The main difference between these strategies is that white-box testing assumes that the tester has access to the source code of the application. Though there is a reference implementation of the KVM for which the source code is available, there is no way to tell exactly which VM is installed for a given phone (and even if the reference KVM is used, it might have been compiled with different options set). Another interest of white-box testing is its ability to test each component of the software individually. This is not extremely useful for vulnerability assessment because bugs might be located in "unreachable" parts of the code, and therefore might not be exploitable. Moreover, due to the embedded nature of the KVM it is impossible to instrument its code on the telephone, and as a consequence white-box testing the KVM would have meant working on a PC rather than a telephone.

For all these reasons, **black-box testing** was chosen. Practically speaking, it means that test midlets had to be created, installed and run on the telephone, and then the behaviour of the telephone had to be observed and analysed.

4.2 Coverage

When people refer to Java low-level security they usually refer to the bytecode verifier. However, on a J2ME implementation some lower levels are worth testing. Here are the potential targets for a test:

- The installer;
- The jar file handler (of the installer or the KVM);
- The class file format integrity checker;
- The bytecode verifier;
- The runtime environment;
- The APIs.

In order to produce accurate results, the object of the test had to be narrowed. Therefore, the focus has mainly been put on the **jar file handler and the class file format integrity checker**. It turned out that the installer was tested too, though it was not a primary objective. Note that the bytecode verifier has not been tested directly because of the specific amount of work needed, it would, however, be worth testing too.

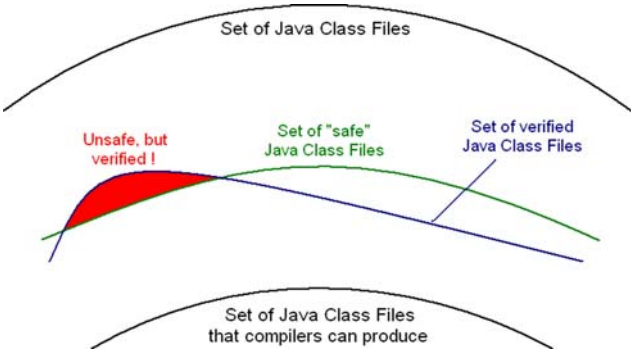

Fig. 1 Class files organisation

4.3 Target

The next step was to define exactly the target of the study, that is to say the kind of bugs to look for. Ultimately, the goal was to find class files as shown in Fig. 1. The expression *class files* is used instead of *classes* because the study involved creating invalid class files, which may not represent any class (according to the specification). This distinction is very important: jar files are created, containing class files, which might eventually but not necessarily be parsed and translated into working classes. Figure 1 refers to the set of "safe" Java class files. Giving a formal definition of "safe" is very hard, and this is not very important for this study (we can assume that "safe" and "valid" mean the same). The set of verified Java class files is also mentioned, it is the set of class files that will be declared valid by a verifier. The mission of the verifier is to validate only safe class files, but Fig. 1 shows that some unsafe class files are verified. For this study, we assumed that the subset of **unsafe but verified class files** existed, and this is what we are looking for.

4.4 Tools Used

At the time of the study, no low-level manipulation tools for jar (i.e. zip) files could be found. And one of the only standard tools for manipulating classes was Jasmin, a Java assembler. Its purpose, however, was more to learn the virtual machine's instruction set rather than to create invalid files. Moreover, the existing Java disassemblers were not satisfying, most of them crashing or giving no output when they encounter an invalid class file. Therefore there was a need for new tools, fulfilling the following requirements:

- They must give sufficient low-level control over the creation of binary files (zip and class);
- The disassembling tools must work on broken, invalid, illegal or truncated files. This is a very strong constraint, and this is one of the top reasons to create new tools;
- The tools must automate a maximum of the process of creating invalid midlet suites.

The toolkit has been written in Java, it has a command-line interface and has been released under the GNU General Public License [8]. It contains the following tools:

- **zip2xml**: a zip "disassembler". It converts a zip file to an equivalent xml file, each element in the xml file representing a data structure defined in the ZIP File Specification [6].
- **xml2zip**: the opposite tool, taking the xml representation of a zip file and producing the binary file. The output file can be totally invalid because each value in the specification can be modified manually.
- **jasmin**: the de-facto standard Java assembler. For the occasion, it has been updated in order to give more control over the output files. Some of the latest Java attributes are now also supported.
- **dejasmin**: a Java disassembler, able to produce output in the new Jasmin format.

A future improvement idea would be to provide tools such as class2xml and xml2class in order to give absolute control over the output class file.

4.5 Experimental protocol

Once the strategy, the target and the tools have been found, an experimental protocol had to be defined. The idea of producing test files, running them on the telephone and then analysing its behaviour is fine but it would have given nothing. The problem is that on the tested telephone, when an error or an exception is thrown, the VM shuts down silently. There is no error message or stack trace that could help understanding the problem. Therefore, it is impossible to make behavioural testing with just a telephone, precisely due to the lack of explicit behaviour. A small workaround has been used: first produce invalid jar and class files, and then run them on a telephone, an emulator and a J2SE runtime environment. This way, it is possible to understand the behaviour of the KVM on the telephone, compared to the other VMs. The test platform was the following:

- A Nokia 6680. It is a Series 60 v2 phone, with CLDC 1.1 and MIDP 2.0 on Symbian OS v8.0a;
- Sun Wireless Toolkit 2.3 Beta as the emulator;
- Java 1.5 SDK on Windows XP SP2 as the J2SE runtime environment.

Finally, what is the nature of the errors in the jar and the class files? The ZIP file format specification and the class file format specification define data structures with certain structural or semantic constraints. The idea was to **create jar and class files with invalid values for each data structure**, one at a time. For example, for an unsigned 4-byte value in a class file, it can be tested for the values 0, 0xFFFF, 0x7FFFFFFF, 0xFFFFFFFF and some other values that might seem interesting or are declared forbidden for this particular data structure. This technique is inspired from a more general idea called fuzzing [1], which can be applied to any input parser and which consists in generating slightly invalid input for a given parser, or valid input but with unusual sizes or values in order to find implementation flaws.

4.6 Test results

First of all, no security flaw was discovered during the study. However, some interesting results were found and their consequences are going to be detailed in the following sections. Here is a quick summary:

- A serious J2SE bug has been found in the class file parser and has been reported to Sun [3] (the attribute_length, a 4-byte unsigned value, was parsed as a signed value).
- There were many differences between the telephone and the emulator, contrary to one of the initial assertions in this study. For example, it was possible to create a jar file that can run correctly only on the telephone, not on the emulator.
- Some totally unexpected behaviours showed up, particularly in the Application Management Software on the telephone. For example, the result of a given installation can depend on the previous installations. It was even possible to craft a particular jar file which installation always fails but will also make the next installation of a midlet fail.
- Many bugs seemed to come from the fact the the installer unpacks and parses each class file in the jar file (detailed in Sect. 5.3).
- The jar file parsing is totally different for J2SE, for the emulator and for the VM on the telephone (detailed in Sect. 6.2). This might be exploited by malware authors to protect their creations.

A detailed analysis of the results as well as the results raw data can be found at [9].

5 The Java mobile risk for the user

This section underlines the importance of mobile devices security for the user and some problems that might not always be obvious: namely that a nice logo does not necessarily mean the product is secure and that installing a program without even running it can be dangerous.

5.1 A sensitive environment

Why would there be a specific risk for mobile Java? The short answer is the **potential cost for the user**. Java is widely used on the web for applets and though there have been historical security problems with them, hostile applets have not been that important. Given the fact that the average workstation can be considered compromised in some way, people don't seem to pay too much attention to "exotic" threats such as Java malware. However, if a malicious application gains control of a mobile phone, it may be able to send an arbitrary number of SMS or MMS, or even issue calls. Therefore, the serious risk here is to end up with a huge monetary cost for the user.

Another problem specific to mobile phones is that the damage done to the device can be hard to put back. It might for example be impossible to totally disinfect a virus or restore some data. The simplest solution in many cases is probably to just replace the telephone with a new one.

5.2 Signed midlets

To gain more privileges on the target device, a midlet can be signed. It can then access a different protection domain, depending on the MIDP version and the root certificate used to sign the application. MIDP 2.0 defines four protection domains: Untrusted, Trusted Third Party, Operator and Manufacturer. An unsigned midlet runs in the untrusted protection domain, with very few privileges. For example, an untrusted midlet can only access the file system with explicit user approval and the access to some files might not be permitted at all. The trusted third party domain does not allow full access to the system, as opposed to the situation with applets. And it is no longer interesting to self-sign midlets because they will run in the untrusted domain. The situation is better for malware development with applets: you can self-sign your malicious applet and the user just has to answer yes when prompted and you have full access to the system of the victim.

To have a midlet signed for the trusted third party domain, it is possible to use the Java Verified program [2]. It was initiated by the industry under the Unified Testing Initiative label. It is a commercial program giving you the opportunity to promote your midlet and to use the Java Powered logo. In order to be signed, the midlet has to be tested automatically and manually by a test provider (which is a company). If the midlet is compliant with some quality and behaviour rules it is signed and will run in the trusted third party domain on mobile phones for which the UTI Root Certificate is available.

Users have to be aware that running a Java program with the Java Powered logo, which has been thoroughly tested and digitally signed does not mean it is not a malicious application. This is possible because programs such as Java Verified are commercial and are therefore made for commercial applications. It means that when you submit your midlet for testing, you do not have to provide the source code. And due to some intellectual property legislation, the tester probably does not have the right to reverse engineer the application to check that there are no hidden features. So the tester can just ensure that the program is not a malware by just using it a few times, which is of course impossible. Consequently, **malicious applications can undergo the Java Verified testing process successfully**, the malware author just has to implement something like: `if (date < someDate) then showFakeGame() else launchRealMalware()`. This is not specific to the Java Verified program, it is true for every behaviour-based testing program.

5.3 Malicious installation

The study which led to this paper outlined many problems in the Application Management Software on the telephone. It is the piece of software responsible for the installation and removal of applications on the phone. Most of these problems showed up while testing the JAR file format. The purpose was to study the response of the virtual machine to specially crafted JAR files but it turned out that most of the time these JAR files never reached the virtual machine at all because they caused errors in the AMS. It was surprising because some behaviours that have been observed were totally unexpected, for example on the tested telephone the AMS seemed to unpack each class file in the archive (even unused class files) and to parse them up to `this_class` (it is a field in class files located after the constant pool). Though there must be a good reason for it to be implemented this way, it is quite puzzling.

The problem with these kind of behaviours is that they are unexpected because they are totally undocumented. For testers, it sounds more likely to find bugs in undocumented pieces of software that in publicly specified and reviewed processes such as the bytecode verification. It turned out to be true: it was very easy to find bugs in the AMS and they might eventually become security flaws. This is even more deceptive for the user because we naturally have the intuition that it might be dangerous to *run* something. In this case, **it might be dangerous to *install* something**, without even having to run it to trigger the payload.

5.4 Need for more usability

The general impression is that the whole J2ME security architecture is way too tight. Developers need more features, and users want more usability. For example, they want to be able to install a game without being prompted five times with "are you sure?", and then five more times in the game when it tries to use Bluetooth to connect to their friend and to access the file system to save the game. In the early times there was no support at all for Bluetooth or file I/O, now it is supported but with a lot of security prompts. These **security prompts are likely to disappear** or at least to be less omnipresent. This approach, setting up a J2ME environment with "too tight" security, must be intentional. The industry probably did not want a virus outbreak in the early days of mobile Java but the situation is probably going to evolve as J2ME environments mature.

On the tested telephone, it seemed to be impossible to make a Java virus without breaking the sandbox because the file connection attempts at other JAR files threw a SecurityException. This is good news. However, the bad news is that the file permission system is not transparent at all, and it is more than likely that it is telephone dependant and maybe even firmware dependant. Anyway, it is not transparent at all, therefore there is a risk as explained above.

6 The Java mobile risk for the analyst

In this part of the paper, we assume that mobile Java malicious applications exist in the wild (this is currently not the case) and that they have to be analyzed quickly in order to be stopped. So the word "analyst" refers to a reverse engineer trying to figure out what a given malware does. Some of the problems analysts might encounter with their reverse engineering tools are going to be underlined here.

6.1 What you see is not what you run

This is particularly true if midlets are to be reverse engineered. Analysts are very skilled people but they are humans: they are going to use tools to do their job. The problem is that most Java reverse engineering tools are outdated, if maintained at all. Most disassemblers and decompilers have been developed at a time when people found it revolutionary that you could translate accurately Java bytecode back to source code. In the mean time, the Java language has evolved and J2ME appeared. J2ME introduces a new verification process with StackMap attributes. If a vulnerability is found in the verifier, it may use malformed StackMap attributes. Therefore it might be impossible to understand the vulnerability without looking at the StackMap attributes. And here is a big problem for the analyst: **StackMap attributes are silently ignored by most Java reverse engineering tools**, along with other new attributes. So it is possible to spend hours trying to figure out what a given program does, but it is just impossible without some "hidden" attributes.

The last problem of analysis tools is the way they resolve ambiguities. Sometimes, file format specifications might contain ambiguities that might complicate the file parsing. The class file format is very well specified and there are only very few ambiguities. Here is an example of ambiguity: as specified in the Java Virtual Machine Specification, the `attribute_length` field of a `ConstantValue` attribute must be 2. But what happens if a class file parser encounters an attribute with name "ConstantValue", a physical length of 2 but an `attribute_length` value different from 2? The behaviour is implementation-dependant and might vary between a real virtual machine which might skip this attribute and a reverse engineering tool which might not know what to do with this strange attribute. These kinds of ambiguities are very common in the zip file format.

The solution to these problems is to use up-to-date, reliable and low-level reverse engineering tools. The tinapoc toolkit described in Sect. 4 has been created for this purpose.

6.2 Multiple behaviors exploitation

Different systems have different reactions, or behaviours, when they have to deal with slightly invalid input. Here are some of the possible behaviours:

- Clean exit;
- Error message and exit;
- Warning but the program tries to cope with the error;
- Bug or incorrect behavior;

- Crash;
- Nothing: the error has not been noticed.

By running slightly invalid jar and class files on different platforms, very different behaviours have been encountered:

- On the telephone: "clean exits" for the VM (some crashes might have not been noticed, anyway there was no error message). Lots of bugs and incorrect behaviours in the AMS.
- On the emulator: very broad range of response, there was a quite large number of crashes though.
- On the PC: usually an error message with the stack trace, useful for debugging.

We can notice that the behaviours are very different on each platform for the same input files. It means that for a given input file, different virtual machines will react differently. It is quite easy to craft a jar file that will run only on a telephone but not on an emulator or a PC with such results. The problem is that it can be exploited by malware authors if they study the behaviour of a telephone and compare it to the behavior of a debugger or a disassembler. It is even easier for debuggers because in order to debug a midlet, a debugger has to connect to a running emulator. So if the emulator does not handle the file successfully, the debugger will fail too. Disassemblers are harder to fool but in order to work, the class files have to be extracted from the JAR file and it is very easy to produce hard-to-parse JAR files.

To summarize: by studying the reactions to errors in input files for different systems (a target phone, an emulator, an extraction utility), **it is easy to produce a midlet that will run but cannot be analysed without modifications**. The solution would be to specify the exact behavior of an implementation when it encounters an error. In practice this is impossible to achieve, therefore there is a need for reliable and low-level analysis tools.

7 Conclusion

We have seen that the user must be aware of the specific risks of mobile Java. The threat is less serious than on PCs but it might evolve. And in case the situation evolves and malicious applications appear for J2ME, analysts need to be equipped with efficient tools. Finally, we have seen that a general but simple way to increase the security of J2ME is to adopt more transparent implementations and to take care when removing security measures.

The current security level of J2ME implementations is satisfying but likely to decrease. Java viruses for J2ME are probably going to appear, even as proof-of-concepts.

The last thing is that the landscape of Java malware is almost empty. In the short run, real J2ME malware is not likely to appear because it would need a lot more work than native malware, for less benefit.

Acknowledgements My thanks go to Mr. John Healy, lecturer at the Galway-Mayo Institute of Technology in Ireland, who directed and helped me throughout this study. I would also like to thank the Irish officers from USAC in Galway for their support and invaluable friendship, as well as Lieutenant-Colonel Filiol for his confidence in my work.

References

1. Holz, T., Van Sprundel I.: Recherche de vulnérabilités à l'aide du fuzzing, MISC - Le magazine de la sécurité informatique, 23, janvier (2006)
2. Java Verified:Unified Testing Criteria for Java(TM) Technology-Based Applications for Mobile Devices, Version 2.0, http://www.javaverified.com
3. J2SE Bug Report http://bugs.sun.com/bugatabase/view_bug.do?bug_id=6352834 (2005)
4. JSR 118 Expert Group Mobile Information Device Profile for J2ME, Version 2.0, http://www.jcp.org/en/jsr/detail?id=118 (2006)
5. Lindholm, T., Yellin, F.: The Java Virtual Machine Specification (Java Series), 2nd ed., Addison-Wesley, Reading (1996)
6. PKWARE : ZIP File Format Specification 6.2.1., http://www.pkware.com/(2006)
7. Reynaud-Plantey, D.: Reverse Engineering and Java Viral Analysis. In: Proceedings of the 2005 Virus Bulletin International Conference, pp. 121–126. http://www.esat.terre.defense.gouv.fr/cresat/articles/java_malware_analysis.pdf (2005)
8. Reynaud-Plantey, D.: J2ME Low-Level Security. http://tinapoc.sourceforge.net/ (2005)
9. Reynaud-Plantey, D.: J2ME Low Level Security: Implementation Versus Specification, Engineer Diploma Thesis. http://prdownloads.sourceforge.net/tinapoc/Reynaud_J2ME.pdf?download (2005)
10. Sun Microsystems Inc: Connected, Limited Device Configuration 1.1 (JSR-139). http://jcp.org/jsr/detail/139.jsp (2003)