Humboldt University Berlin
Computer Science Department
Systems Architecture Group

Rudower Chaussee 25
D-12489 Berlin-Adlershof
Germany

Phone: +49 30 2093-3400
Fax: +40 30 2093-3112
http://sar.informatik.hu-berlin.de

Self-Replication in J2ME MIDlets.

HU Berlin Public Report
SAR-PR-2006-04

March 2006

Authors:
Henryk Plötz, Martin Stigge, Wolf Müller, Jens-Peter Redlich

Self-Replication in J2ME MIDlets

Henryk Plötz, Martin Stigge, Wolf Müller, Jens-Peter Redlich
(ploetz|mstigge|wolfm|jpr)@informatik.hu-berlin.de
Systems Architecture Group[*], Computer Science Department,
Humboldt-University Berlin, Unter den Linden 6, 10099 Berlin, Germany

May 15, 2006

Abstract

A J2ME MIDlet is a Java application which runs in a very restricted environment, typically on a mobile phone. These don't necessarily have a file system, and even if they do, MIDlets usually don't have full access to it. Also they don't know where their code is stored in the file system, or if it's stored there at all. So for all practical purposes MIDlets can not directly get a copy of their code. Still, some applications such as social networking software might benefit greatly from the possibility of replicating the software from phone to phone. Because most mobile phones do not offer such functionality by themselves –probably for fear of software piracy– we'll have to implement that in our software and therefore need a copy of the code that is currently running.

1 Introduction

J2ME MIDlets run in a sandbox which imposes more restrictions than is common for Java programs. As a result it is normally not possible for a MIDlet to directly access its installation JAR file and therefore it can not generate a copy of itself, for example for sending to other phones.

But under certain cirumstances it would be favourable to have such a functionality. For example we're researching social networking software where widespread propagation of the software is a high priority. Having a way of offering to spread the software directly from mobile phone to mobile phone –without incurring any charges– should entice users to share the software with their friends or even random acquaintances, if it seems useful.

Shortly before the release of this paper another actual example came up: The new semacode.org (n.d.) J2ME standalone reader (version 1.6) prominently features a "Recruit!" soft-button which leads to a screen that says: "You can share Semacode with your friends. Use this to send them a text message with a free download link."

The Blooover II breeder edition (Herfurt, 2005) was the first MIDlet we saw which was capable of reproducing a copy of itself over the bluetooth interface. It does this by including a copy of the non-breeding edition as a resource in the

[*]http://sar.informatik.hu-berlin.de

installation JAR file, which can subsequently be accessed by the code and sent to another mobile phone. This approach has two major drawbacks:

1. It is only possible to breed exactly one new generation. The newly bred copies can not generate any further copies.
2. The installation JAR file doubles in size because a full new copy must be included.

This paper will try to address both issues one after the other: Section 3 solves the first problem in a rather traditional manner and section 4 solves the second problem through some trickery with the JAR file. We'll then conclude with some remarks on the problems with signed code in section 5.

An –at least superficial– understanding of the the JAR file format will be necessary to follow the course of this paper. For this reason we shall like to restate the most important facts about the ZIP file format first, since a JAR file is little else than a ZIP file with a different file name extension and a special META-INF directory inside.

2 Detour: The Zip File Format

The ZIP file format (ZIP spec, 2006) was originally devised by Phil Katz in 1989 for his then new PKZIP file compression utility. Its basic structure consists of one or more files concatenated together, each prepended with a local file header and possibly compressed, and a central directory of all files in the archive at the end. The end of the central directory is marked by a special "end of central directory" record which includes information about the central directory such as the number of entries, its length and its offset from the start of the file[1].

Each local file header includes information about the file in question such as the original file name, the compressed file size, the uncompressed file size, the file's CRC-32, the compression method that was used, a general purpose flag bit field, etc. One important point to notice: If bit 3 of the general purpose bit field is set, then the compressed size, uncompressed size and CRC-32 in the local file header will be set to zero and instead be included in a trailer (called "data descriptor") after the actual file contents. It is therefore not possible to read a ZIP file from the start in the general case, because one might need to jump over an unknown number of bytes in order to learn how many bytes one needed to jump over. The correct size and CRC-32 values will however be set in the central directory in any case.

This design choice was presumably made in order to be able to stream a new ZIP file out of stdout, but comes at a cost: To read a ZIP file one needs to jump to the end of it, find the "end of central directory" record to locate the beginning of the central directory and then jump to the central directory and read this in order to make sense of the rest of the file.

The records in the central directory are similar to the local file headers but there are some additional fields. The most important of these gives the offset to the start of that file's local header.

[1] Actually the start of the central directory is indicated as the offset from the beginning of the volume that the central directory starts on, but we will neglect the possibility of multi-volume archives for this article, as we will some other features of the ZIP file format such as digital signatures or the ZIP64 extension.

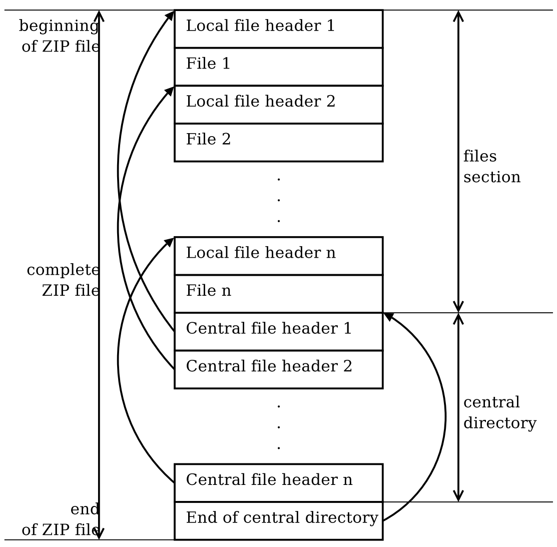

Figure 1: General overview of the ZIP file format

Note that all structures (local file headers, central directory records and end of central directory record) start with a magic word to make identifying them easy (or even possible at all). Table 1 in appendix B lists the magic words for the most important structures. Appendix A details their layouts. A general overview of the ZIP file format, also depicting which section links to which section, is shown in figure 1.

Note that this short overview only included the structures that are relevant to the subject at hand and left out a lot of details. Anyone who is truly interested in the ZIP file format should read its specification.

3 The Re-Insertion Technique

A J2ME MIDlet can access resources from its installation JAR file using

```
this.getClass().getResourceAsStream("/foo")
```

so it is easy for a MIDlet to produce a copy of something –for example a JAR file– that is included in its JAR:

1. Build the MIDlet suite, get an installation JAR file `a.jar`.
2. Copy `a.jar` into the MIDlet suite's resource directory and rebuild the suite, get an installation JAR file `b.jar`.
3. Distribute `b.jar`.
4. When run, `b.jar` can read in a copy of `a.jar` and for example send this using OBEX push to another mobile phone.

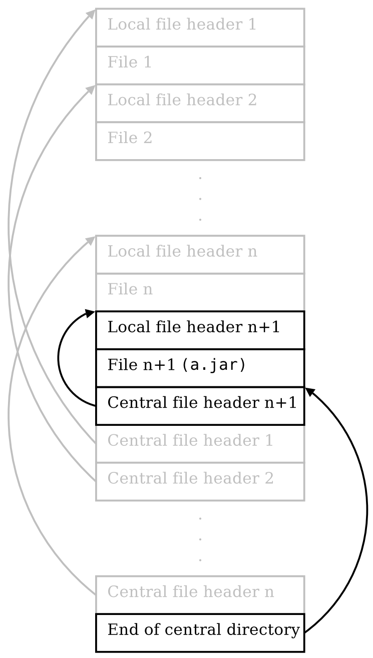

Figure 2: The ZIP file after reinsertion

5. When run, `a.jar` cannot read in a copy `a.jar`.

Note: Up to here this is about the same thing that Blooover does.

So what do we do about the problem in step five? Well, we do have a programming language at our disposition and can modify the in-memory copy of `a.jar` to our likings, right? So simply re-insert a copy of `a.jar` into our existing copy before passing it on.

Figure 2 shows what our ZIP file looks like after the reinsertion, emphasizing those structures that need to be changed or updated. What needs to be done, in order:

1. Locate the end of central directory record and the beginning of the central directory.

2. Just before the beginning of the central directory, insert a local file header describing our file `a.jar`: Give (at least) the file name, compressed size and uncompressed size (those two are equal), CRC-32 and compression method (stored, not compressed). Insert an unmodified copy of the file `a.jar`.

3. After that insert a central directory record for this file with the same information and additionally give the offset to the local file header (this is the same offset at which the central directory was previously found).

4. Revisit the end of central directory record and update the information: Add one to the number of entries, increase the size according to the size

of the new record from step three, adjust the offset according to the size of the new file and local file header from step two.

That's all there is to it. If this modified file is transmitted onto another mobile phone and run, it will be able to execute the same steps again, as will its offspring and so on.

This solution is not completely satisfactory though: Because the `a.jar` file needs to be included into `b.jar` the size of `b.jar` will be about double the size of `a.jar`. This can make the difference between a reasonably sized MIDlet and an unreasonably sized MIDlet, especially in situations where the users need to –or want to– download `b.jar` using their mobile phone services which often are billed by volume. The size of the modified `a.jar` is not so much of an issue, because sending over OBEX push is free. (Usually the modified `a.jar` will be bigger than `b.jar`, despite them having about the same contents, because we do not use any compression when reinserting `a.jar` into the modified `a.jar`, while the program that included `a.jar` in `b.jar` will have used compression.)

4 The Matryoshka Technique

So what can we do about the size problem? Looking at the ZIP specs and at our modification of `a.jar` we will notice two things:

1. The file that we inserted is just the same as the file we are inserting it into.

2. The ZIP data structures give explicit length and offset for *each* file in the archive and there's nothing that prevents us from creating overlapping files.

So, don't you just wonder whether we can create an entry for a file that overlaps the whole archive? We can, as it turns out, though the complete solution involves some fiddling with the CRC-32.

4.1 Approach

The steps needed differ somewhat from the previous technique and most importantly they are carried out right after building the JAR file, on the developer's machine, before distribution:

1. Locate the end of central directory record and the beginning of the central directory.

2. Just before the end of the central directory (the location really is arbitrary), insert a central file header describing a file that starts at offset zero and is [(size of the JAR file before modification) + (size of our new central directory record)] bytes long, with some file name `foo`, no compression. You can't fill out the CRC-32 field yet because it is going to change, so simply set it to zero. We'll get to this later.

3. At the very beginning insert a local file header describing the same file. Note that now the *complete* JAR file (minus the local file header) can be treated as a file `foo` within the JAR file. Pretty cool, eh?

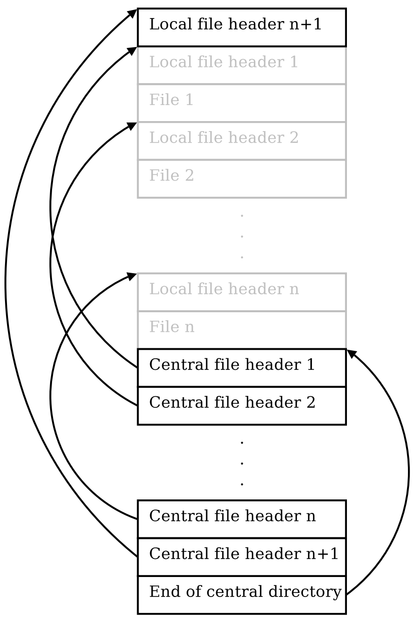

Figure 3: The ZIP file after applying the Matryoshka technique

4. Fix all the offsets that you broke by inserting the local file header at the beginning: Add the size of the header to each and every offset (central directory records and end of central directory record), except for our new central directory record.

5. Calculate the CRC-32 of the `foo` file (e.g. everything after our new local file header) and fill in the CRC-32 field in the new local file header. You cannot fill in the CRC-32 field in the central file header, because that would change the CRC-32.

The end result should look like figure 3.

In theory the MIDlet should now be able to read the file `foo`, just as it did `a.jar` in the previous section and acquire the complete JAR minus the local file header which it just needs to recreate by applying steps 3 and 5 again. No offsets need to be fixed, because the recreated local file header has the same length as the original one (of course the MIDlet needs to use the same file name again).

In practice that doesn't work, because the CRC-32 in the central directory doesn't match the CRC-32 of the file `foo`. You can not easily avoid this problem because updating the CRC-32 in the central directory would mean changing the CRC-32 of the file `foo` necessitating a new update in the central directory, ad infinitum.

4.2 Fixing The CRC-32

So, we need to be able to set the CRC-32 before the data that it is computed over is completely available. Luckily CRC-32 is a simple linear operation that can easily be reversed. We simply think of a value, put this into central directory for the CRC-32 field and then modify our data so that it checksums to that chosen value.

anarchriz (1999) offers an approach that modifies the data's last four bytes to fix the CRC-32. To apply this work to our file we add a ZIP file comment of five bytes to the end of the file and update the end of central directory record as well as the local file header and the central file to reflect this change. The last four of these bytes are overwritten with the CRC-32 compensation data and the first byte we leave at zero. This prevents the printing of this comment in some programs (a zero byte marks the end of a string in C) and doesn't hurt.

Stigge (2006) has an even better approach that enables us to put the four compensation bytes anywhere in the data, not solely at the end. We haven't yet used the last mod file time/date fields (which together give us four bytes) so we can put the compensation data there.

5 Coping With Digital Signatures

Digitally signing code is important, especially in this context where the code is received from some (untrusted) third party and not from the original developer.

5.1 Java 2 Standard Edition

Code signing in the Java 2 Standard Edition follows the "Signed JAR File" part of the JAR File Specification (2003). Signing in this standard consists of:

- Additional entries in the MANIFEST.MF file listing the hashes of all files that are to be signed.
- Additional pairs of files, one for each signer: A signature file that lists all the files that are to be signed by that signer and their hashes and a digital signature file with that signer's signature over the corresponding signature file.

It follows naturally that both techniques that have been discussed so far do not interfere with the J2SE signing and verification process. Two points of advice:

- Sign first and then modify the signed JAR, especially when using the matryoshka technique. Otherwise the `jarsigner` tool will destroy the matryoshka property when repacking the file.
- You may want to put the virtual `foo` file into the META-INF directory inside the JAR (e.g. `META-INF/foo`), because these files will be ignored when signing and verifying.

5.2 Java 2 Micro Edition

The Java 2 Micro Edition –following the "Wireless is Different" doctrine– uses a different verification scheme. A signed MIDlet is split into two different files:

- A JAR file with the code, and

- a JAD file with some meta-information and the signature over the complete JAR file.

The common way to install a signed MIDletis to download the JAD file via (WAP or HTTP) over (GSM or GPRS or UMTS), get the JAR file's URL from the JAD and then download the JAR file (also over the air interface). That last part is precisely what we want to avoid when using phone to phone replication.

We're facing two fundamental problems when trying to apply our self-replication techniques to signed MIDlets:

1. Any modification of the JAR file will invalidate the signature, so the file must be signed after modifying. It is therefore not possible to incorporate the signing JAD file into the signed JAR file.

2. It is not possible install a signed MIDlet in a standardized way over a short-range communications technology (Bluetooth, infrared). Installing or offering to install the unsigned MIDlet was easy because that just involved sending the bare JAR file using OBEX push. But in order to send a signed MIDlet it is necessary to somehow transmit the JAD together with the JAR and then get the target phone to recognize their relationship. For some phones this is possible by sending both files with OBEX push and then instructing the receiving user to visit his or her inbox and install the received JAD file. For some phones it is not possible at all, or only through proprietary protocols[2].

We cannot really do anything about the second problem. But we can try to overcome the first problem when we pretend that we solved the second problem.

The one issue that is still left is: How do we come up with a JAD file that is a proper signature for the JAR file that we want to transmit (which should preferably be a copy of the JAR file that is currently running)?

Because we can't include the JAD in the JAR (unless we find some way to break the digital signature in a similar way that we broke the CRC-32, which is neither easy nor what we want), we'll have to store it in the MIDlet's data storage area (RMS, Record Management System). There several possibilities to get it there:

- If the program that is to be replicated needs online connectivity anyway, we could use this connection to get a copy of the JAD into the RMS.

- If the receiving user is advised to run the MIDlet immediately after replication, we could let the old and new MIDlet instances communicate and thus transfer the the JAD into the new instance's RMS.

- For the first generation MIDlet, we could apply a similar technique to that seen in section 3: We generate a first MIDlet version `a.jar`, sign that in order to get `a.jad`, then include `a.jad` into the JAR file while simultaneously applying the Matryoshka technique, resulting in `b.jar` and finally sign that to get `b.jad`.

  The first-generation user then downloads `b.jar` and `b.jad` from the website and runs it. This MIDlet can now use the Matryoshka code to retrieve

[2]The Device Explorer software that comes with the Development Kit for our W800i seems to be able to install a JAR/JAR combination.

a copy of `b.jar`, then extract `a.jad` and undo all modifications on its `b.jar` copy to generate a copy of `a.jar` (for which it has the matching `a.jad`).

6 Demonstration

We have prepared a proof of concept MIDlet which is available from our website at http://sar.informatik.hu-berlin.de/self-replication. This MIDlet, when downloaded and run on a mobile phone, will reconstruct its install JAR and then print this JAR's size and CRC-32. Use the "Search" button to search for neighbouring devices that offer to accept an OBEX push and then send a copy of the JAR to one of those devices (you may be asked to grant security permissions to the MIDlet in order to do that). The received copy should be bit-idententical to the original file (e.g. same size and same CRC-32). If the receiving device is a mobile phone it should be possible to install and/or execute the received copy right away.

Note: We have only successfully tested this MIDlet with a Sony-Ericsson W800i and similar Sony-Ericsson devices. A lot of other mobile phones (e.g. the Siemens S65) seem to have an incomplete implementation of JSR-82 (the Bluetooth API for J2ME that we are using to send the OBEX push) and are therefore not capable of running our MIDlet (though they can receive it just fine). It may be possible to work around this limitation by writing a custom OBEX implementation that uses just the basic JSR-82 features, but this is outside of our scope.

A Structures

A.1 Local File Header

Offset	Size	Meaning
0	4	Local file header magic word
4	2	Version needed to extract
6	2	General purpose bit flag
8	2	Compression method
10	2	Last mod file time
12	2	Last mod file date
14	4	CRC-32
18	4	Compressed size
22	4	Uncompressed size
26	2	File name length
28	2	Extra field length
30	variable	File name
variable	variable	Extra field

A.2 Central directory record

Offset	Size	Meaning
0	4	Central file header magic word
4	2	Version made by
6	2	Version needed to extract
8	2	General purpose bit flag
10	2	Compression method
12	2	Last mod file time
14	2	Last mod file date
16	4	CRC-32
20	4	Compressed size
24	4	Uncompressed size
28	2	File name length
30	2	Extra field length
32	2	File comment length
34	2	Disk number start
36	2	Internal file attributes
38	4	External file attributes
42	4	Relative offset of local header
46	variable	File name
variable	variable	Extra field
variable	variable	File comment

A.3 "End of central directory" record

Offset	Size	Meaning
0	4	End of central directory magic word
4	2	Number of this disk
6	2	Number of the disk with the start of the central directory
8	2	Total number of entries in the central directory on this disk
10	2	Total number of entries in the central directory
12	4	Size of the central directory
16	4	Offset of start of central directory with respect to the starting disk number
20	2	ZIP file comment length
22	variable	ZIP file comment

B Tables

Structure	Magic word (in hex)
Local file header	`50 4b 03 04`
Central directory record	`50 4b 01 02`
End of central directory record	`50 4b 05 06`

Table 1: Magic words for different structures in the ZIP file format

C References

anarchriz. (1999, April). *CRC and how to reverse it.* (Available from http://www.woodmann.com/fravia/crctut1.htm)

APPNOTE.TXT – .ZIP file format specification. (2006). (Version: 6.2.2 (revised: January 6, 2006) available from http://www.pkware.com/business_and_developers/developer/popups/appnote.txt)

Herfurt, M. (2005). Blooover - J2ME phone auditing tool [Computer program]. (Available from http://trifinite.org/trifinite_stuff_blooover.html)

(n.d.). (http://www.semacode.org)

Stigge, M. (2006, May). Reversing Crc – Theory and Practice.

Jar *File Specification.* (2003). (Available from http://java.sun.com/j2se/1.5.0/docs/guide/jar/jar.html)

Reports published by Humboldt University Berlin, Computer Science Department, Systems Architecture Group.