2015 International Conference on Computational Science and Computational Intelligence

Java tool extensions for supporting multiple recommenders and distributed bundles

Marius Silaghi
Department of Computer Sciences and Cybersecurity
Florida Institute of Technology
Melbourne, Florida 32901
Email: msilaghi@cs.fit.edu

Khalid Alhamed
Information Technology Department
Institute of Public Administration
Riyadh, Saudi Arabia
Email: hamedk@ipa.edu.sa

Ryan Stansifer
Department of Computer Sciences and Cybersecurity
Florida Institute of Technology
Melbourne, Florida 32901
Email: ryan@cs.fit.edu

***Abstract*—A JAR (Java Archive) is typically used to incorporate code and associated resources into one file to distribute Java software. A cryptographically signed JAR file provides assurance about the authorship of the contents of the archive. We use Signed JAR files as part of a recommendation system. In this system different recommenders will evaluate the same software, and they need to sign the exact same JAR file. The user wants to verify that recommendations (i.e., signatures) received independently from multiple parties, e.g., for a software update, pertain to the exact same software. Related problems occur when users try to sign bundles consisting of files maintained on different servers. The tools in the Java Development Kit do not support this kind of application.**

We propose techniques to enable the signing of distribute bundles and techniques by which recommenders can sign software independently and such that verifiers are enabled to combine the recommendations. There changes to the Java `jarsigner` tool would avoid special purpose code which duplicates many of the same capabilities of the existing tools.

***Index Terms*—security, update, JAR, signature, recommendation.**

I. Introduction

In this work we refer to enabling the use of JAR files in a new type of application, namely as a vehicle to transmit digitally signed recommendations of software updates. A JAR file is an archive created using the same layout as PKZIP [2] files. Applets, Java class files (libraries), and stand-alone appliations are often disseminated using JAR files. Signed JAR files are used as a mechanism to recommend applets as trustworthy for accessing resources. Signed libraries are used for enabling strong cryptography that used to be export-controlled. Signed applications could be used, in principle, to enable trust in binaries downloaded from the Internet.

The scenario of the application that we want to enable consists of independent recommenders of software updates who, after testing the sources and compiling the software themselves, sign the resulting JAR files for distribution to potential adopters.

If the recommenders compile the same sources with the same compilers and with the same options, then the `.class` binaries will differ in their timestamps. Nothing else would be different. Even if the timestamps of all the files were set to a fixed value before archiving these binaries, the resulting JAR files (if they have a manifest) would still differ in the timestamps and (probably) the CRC[1] checkcodes of the `META-INF/` folder and `META-INF/MANIFEST.MF` file, as recorded in their local file headers and central directory record sections in the JAR file. (See Figure 1.) Without these timestamps, the JAR files could be compared directly or through their digests to detect changes.

If the JAR file is signed, this is done by adding the .SF hashes files, .DSA, .EC and .RSA key files in the `META-INF/` folder. These meta files further change the overall digest of the archives, making them no longer directly comparable, and, thus, difficult to check whether they contain exactly the same software or not.

On the other hand, the .SF meta-files contain the digests of the files in the JAR. This is exactly what we want to compare. Entries in the manifest file have a corresponding section in the .SF text file. The section consists of the file name and a digest. This digest is of appropriate lines of text from the manifest file—typically (1) the file name, (2) the base64-encode digest of the file contents, and (3) a blank line. However, users have to unzip the files and to compare the .SF files manually in order to check whether the content is identical. Since .SF files may contain only subsets of the archived files, the comparison has to be based on the content of a `MANIFEST.MF` file that describes all files.

II. Background

The Oracle distribution of Java provides a tool named `jar` that can be used to create and extract files from an archive. Java code is often distributed using special ZIP archives managed using the `jar` tool. JAR archives are special by the fact that they commonly contain meta-information in a `META-INF/` folder containing a text file `MANIFEST.MF`. This manifest file is updated each time the JAR file is modified [6]. As a consequence the JAR file will depend not only on the contents, but also on when it is made. In the next example we show that creating two JAR files with identical content yields different files.

```
$ which jar
```

[1]The limitations and variations of Cyclic Redundant Codes (CRC) check codes do not concern us here.

```
/data/jdk/jdk1.8.0_31/bin/jar
$ echo 'Hello World!' > file.txt
$ jar cfe test1.jar Main file.txt
$ jar cfe test2.jar Main file.txt
$ cmp -l test1.jar test2.jar
 11 225 230
 72 225 230
291 225 230
350 225 230
```

The Unix tool `cmp` reveals that a difference occurs at addresses 11, 72, 291 and 350. Position 11 is part of the file modification time in the local file header for the folder `META-INF`, while position 72 corresponds to a byte in the timestamp for `MANIFEST.MF`. The other two locations are bytes in the timestamps in the central directory file header for the same files. The layout of these timestamps in the JAR format are shown in Figure 1 from [2].

One can see in the following example that the folder's modification time is changed when the JAR manifest is updated:

```
$ echo Class-Path: a.jar > MANIFEST.txt
$ cp test1.jar test3.jar
$ jar ufm test3.jar MANIFEST.txt
$ jar tfv test1.jar | fgrep META
  0 Wed May 27 09:11:52 EDT 2015 META-INF/
 86 Wed May 27 09:11:52 EDT 2015 META-INF/MANIFEST.MF
$ jar tfv test3.jar | fgrep ME
  0 Wed May 27 09:11:52 EDT 2015 META-INF/
105 Wed May 27 09:20:38 EDT 2015 META-INF/MANIFEST.MF
```

The update to the manifest timestamp is performed even if the command line manifest update file is empty or contains the exact same files as the manifest already in the archive.

The Java Development Kit also provides a tool called `jarsigner` which creates the signature and signature block files in the `META-INF` folders. This signature is generated for a digest that is a fingerprint of files found in the archive. This fingerprint is the value of the manifest attribute named *x-Digest-Manifest*. Here *x* stands for a message digest algorithm, commonly SHA256. Since this attribute is a digest of the whole signature file, and in its turn this file contains the digest of each individual file being signed in the per-entry attribute named *x-Digest*, it is essential that the fingerprint be of the whole software bundle in the archive. Note that we refer to an exact fingerprint, not an approximate one as in [10]. A program that tests similarity of JAR files are mentioned in [3], but it does not address the manifest timestamps issue.

The `jarsigner` tool can be used to add multiple signatures to a JAR file, and can be used to verify each such signature separately, but it cannot tell whether two different archives contain the same software [6]. Note that this tool has an option `-tsa` for timestamps from authorities, which refers to a cryptographic protocol unrelated to the problem discussed here.

III. Tool details

In [1] we describe an application where independent recommenders evaluate and sign software updates.

To simplify the support of independent recommendations of a software in a JAR file as well as the support of distributed bundles, we propose to add new options to the `jarsigner` and `jar` tools.

A. Motivation

For example, an application like the DirectDemocracyP2P in [1] requires recommenders to generate a cryptographic digital signature for a combination of files maintained on different servers. These files can define items such as software packages, assessments, and recommender identity.

Assume that a recommender wants to generate one signature for the bundle of resources that describe his assessment of a binary release of the free open source code at github.com/ddp2p/DDP2P/, namely:

1) `http://john.rec.me/identity.id`
2) `http://ddp2p.net/specs/ddp2p-1.0.3.spec`
3) `http://mirror.net/updates/ddp2p-1.0.3.jar`
4) `http://john.rec.me/evals/ddp2p-1.0.3.eval`

Then the recommender can generate one digital signature for the digest of the stream of data obtained by concatenating the content obtained from these last 4 URLs and can store it as:

`http://john.rec.me/assesments/ddp2p-1.0.3.sign`

A verifier can further generate the digest value of the data from the release described by the 4 URLs and verify the combined data against the signature in `ddp2p-1.0.3.sign`.

The presence of JAR files introduces a complex problem, since the recommender cannot easily check that the content of, say:

`http://mirror.net/updates/ddp2p-1.0.3.jar`

is the same as the one generated from `github` sources for evaluation on a local computer. The problem is that the default timestamp of the `META-INF/MANIFEST` file and folder is the current date, and the content of the obtained jar-file depends on the date of its creation or of its time of update [6].

Furthermore, a simple recompilation of the archive in the mirror would change its content due to the modified timestamp of its manifest, invalidating all signatures referring to it.

a) Solution with Current Tools: Currently, for a program *Main* consisting of files in the list *classes.list*, JAR files that are independent of the update and creation time can be obtained with the following combination of commands:

```
jar cfme /tmp/archive.jar MANIFEST.txt Main \
    @classes.list
cd /tmp
jar xf archive.jar
touch -d  '2009-06-15 13:45:30 Z' META-INF \
    META-INF/MANIFEST.MF
jar cMf archive.jar META-INF/MANIFEST.MF \
    @classes.list
```

These commands still do not solve the problem of integrating multiple recommenders with the `jarsigner` tool. To compare two archives signed by two independent recommenders with aliases `alice` and `bob`, one can use the next combination of commands:

```
mkdir clean; cd clean
jar xf ../archive_alice.jar
rm -r META-INF
jar cMf ../archive_base_alice.jar *
```

offset	*bytes*	*description*
0	4	file header
4	2	version
6	2	flags
8	2	compression method
10	2	last modification time
12	2	last modification date
14	4	CRC-32 of file
18	4	compressed size
22	4	uncompressed size
26	4	length of file name
30	2	length of additional field
32		name of file
		extra field
		compressed data

All multi-byte values are stored in little-endian byte order

offset	*bytes*	*description*
0	4	directory header
4	2	version made by
6	2	version required to extract
8	2	flags
10	2	compression method
12	2	last modification time
14	2	last modification date
16	4	CRC-32 of file
20	4	compressed size
24	4	uncompressed size
28	2	length of file name
30	2	length of additional field
32	2	length of comment
34	2	disk number
36	2	internal file attributes
38	4	external file attributes
42	4	location of file header
46		name of file
		extra field
		comment field
		compressed data

Fig. 1. Local file and central directory file headers in JAR/PKZIP format [2]

```
cd ..; rm -rf clean
mkdir clean; cd clean
jar xf ../archive_bob.jar
rm -r META-INF
jar cMf ../archive_base_bob.jar *
cd ..; rm -rf clean

if \
jarsigner -verify archive_alice.jar alice & \
jarsigner -verify archive_bob.jar bob & \
cmp -l archive_base_alice.jar archive_base_bob.jar\
; then echo success ; fi
```

B. Option for `jar`

We propose to extend the `jar` tool with an option T which specifies a timestamp for the created manifest file. Without specifying this option, the default timestamp is the current date, and the content of the obtained jar-file depends on the date of its creation or of its update [6].

The parameter of the `T` option, inserted in the list of arguments in the order of the appearance of `T` relatively to options *emf*, can consist of any time format [9], [5], or as the list of 16 hexadecimal values for the 4 bytes in the time and date fields of the local file header for ZIP files [4].

Example calls to `jar` with the new options are:

```
jar cfTe archive.jar 2009-06-15T13:45:30 Main \
      @classes.list
jar cmTef archive.jar MANIFEST.txt \
      2009-06-15T13:45:30 Main archive.jar \
      @classes.list
```

JAR files created with the proposed option T and with the same command line and with input files having the same timestamps are identical—regardless of when the JAR file is created. Such JAR files have the advantage that they can be freely combined with other systems into a data stream that can be digitally signed only once, as we require for recommendation systems.

C. Digest option for `jarsigner`

We also propose to extend the `jarsigner` tool with an option `-digest x`, to be used as:

```
jarsigner -digest x jar-file [base-name]
```

The parameter `x` is expected to be the standardized name of a `java.security.MessageDigest` algorithm. With this option, the tool would be expected to print, without modifying the file, the digest it would generate for the attribute *x-Digest-Manifest*.

When called with an optional parameter `base-name`, then the tool prints the digest of the manifest associated with the signature file `base-name.SF`. This is equivalent to the sequence of command:

```
zip -q -s archive.jar META-INF/base-name.SF \
  | grep Digest-Manifest:
```

However, the `-digest` option also displays a digest for JAR files that are not signed, which cannot be done now with the above commands. With this option made available, applications as the one mentioned in motivation can include the digest obtained in this way instead of the content of a JAR, when computing the digest to be signed for a distributed bundle.

D. URL option for `jarsigner`

To enable the inclusion of the signature for a distributed bundle inside the JAR archive, an attribute:

```
Include-URLs: http://john.rec.me/identity.id
http://ddp2p.net/specs/ddp2p-1.0.3.spec
http://mirror.net/updates/ddp2p-1.0.3.jar
http://john.rec.me/evals/ddp2p-1.0.3.eval
```

can be added to the main attributes in the manifest of the signature JAR. The digests for digital signature would then be computed by concatenating the streams obtained from these URLs to the content of the current JAR file.

Since URLs tend to change names, an alternative to this solution is to list the current name for URLs in the desired order on the command line at creation and verification of the signature, using the option `-URL`:

```
jarsigner -URLs http://john.rec.me/identity.id
  -URL http://ddp2p.net/specs/ddp2p-1.0.3.spec
  -URL http://mirror.net/updates/ddp2p-1.0.3.jar
  -URL http://john.rec.me/evals/ddp2p-1.0.3.eval
  ddp2p-sign-john.jar alias-john
```

E. Merge option for `jarsigner`

To better support multiple independent recommenders, we propose to extend the `jarsigner` tool with an option `-merge jar-file2`, to be used as:

```
  jarsigner -merge jar-file2 [-options]
jar-file
```

b) Test 1:: With such a call, the `jarsigner` would first check that the main attributes [7], [8] of the two jar files would be identical, (i.e., having the same *x-Digest-Manifest-Main-Attributes* attribute, if such an attribute would be computed).

c) Test 2:: Then, the files with the same path in `jar-file2` and in `jar-file` are checked to have the same content: (i.e., the same `x-Digest` attribute, if such an attribute would be computed). A failure occurs if the manifests use different digest algorithms.

d) Operation: Finally, the signature and signature block files in *jar-file2* are added to *jar-file*. If signature files with the same base file name were present in both jar files, but with different sets of signed files, then the base file name of the version in *jar-file2* is modified by adding a numerical suffix that does not generate conflicts with the signatures in *jar-file*.

On failure of the first test, the `jarsigner` tool quits with error code -1. If a failure happens during the comparison at Test 2 an error code specifying the index of the failing entry in the generated manifest of the `jar-file`.

IV. Conclusions

We have discussed problems that occur when trying to sign distributed bundles consisting of files maintained on different servers, as well as when trying to combine recommendations (i.e., digital signatures) generated independently for the same JAR files by distinct recommenders.

A solution was shown which is based on current version of the Java tools. We have proposed a set of alternative modifications to the Java tools `jar` and `jarsigner` which can make it possible to easily digitally sign distributed bundles and to combine independently generated signatures for JAR files.

References

[1] Khalid Alhamed, Marius-Calin Silaghi, Ihsan Hussien, Ryan Stansifer, and Yi Yang. "Stacking the deck" attack on software updates: Solution by distributed recommendation of testers. In *IAT*, pages 293–300. IEEE Computer Society, 2013. ISBN 978-1-4799-2902-3.

[2] Florian Buchholz. The structure of a pkzip file. https://users.cs.jmu.edu/buchhofp/forensics/formats/pkzip.html, 2015.

[3] Allan Godding and Edward Duong. Jar identification and duplication detection. https://comp4104a2.googlecode.com/svn/COMP4900/paper.pdf. Date accessed: May 2015.

[4] Paul Lindner. Zip: Registration of a new mime content-type/subtype. http://www.iana.org/assignments/media-types/application/zip, 2015.

[5] Microsoft. Standard data and time format strings. https://msdn.microsoft.com/en-us/library/az4se3k1\%28v=vs.110\%29.aspx, 2015.

[6] Oracle. Packaging programs in jar files. https://docs.oracle.com/javase/tutorial/deployment/jar/, 2015a.

[7] Oracle. Jar file specification. http://docs.oracle.com/javase/8/docs/technotes/guides/jar/jar.html, 2015b.

[8] Oracle. jarsigner. https://docs.oracle.com/javase/8/docs/technotes/tools/unix/jarsigner.html, 2015c.

[9] Oracle. Using predefined formats. https://docs.oracle.com/javase/tutorial/i18n/format/dateFormat.html, 2015.

[10] Muhammad Shafique and Dwight Deugo. finGAD: A jar file fingerprint generator and detector. In Hamid R. Arabnia and Hassan Reza, editors, *Proceedings of the 2009 International Conference on Software Engineering Research & Practice, SERP 2009, July 13-16, 2009, Las Vegas, Nevada, USA, 2 Volumes*, pages 70–76. CSREA Press, 2009. ISBN 1-60132-129-5.