Creating Tar File Releases with Swbis

Making Source Packages with Embedded GPG Signatures
05 August 2006

by Jim Lowe

This manual is for the use of GNU Swbis (version 0.499) in the creation and verification of software releases containing embedded GPG signatures.

Table of Contents

1 Prerequisites

In general, you need Unix-like shell and utilites, including awk. A Posix shell is required, `bash` works fine. You will also need GNU tar version 1.14, 1.15.1 or 1.15.91, and GNU Privacy Guard (`gpg`). The GnuPG passphrase agent, (`gpg-agent`) is supported though optional. The features described in the last two chapters dealing with Automake and CVS require GNU swbis version 0.499.

2 Introduction

This document explains how to create and verify tar archives using GNU Swbis using particular methods and policy suited for free software distribution tar files. The primary motivation for using Swbis is that it can create packages with an embedded GPG signature.

The creation method described uses `swign` which employs `swpackage` and `tar` so that the archive is written entirely by `tar`. The packaging policy is designed so there are no package layout changes except for the addition of the meta-data directory '`catalog`'. This is accomplished by specifying the POSIX control directory as empty strings `""` and using a implementation extension option to set the path name prefix to the package *Name-Version*. The '`catalog`' directory conforms to the POSIX packaging standard ISO/IEC 15068-2:1999.

The verification methods described include a procedure that does not require any part of Swbis. It uses `tar`, `gpg`, and a few other GNU utilities plus a Ext2 compatible file system to verify the package data.

3 Making Distribution Tar Files

Making a distribution tar file first requires making a input file called a *Product Specification File* or PSF for short. It directs `swpackage` on what files to package, the package structure, and what control directory names to use. It also can contain meta-data (i.e. attributes) that are transferred into the package meta-data file named *INDEX*.

Here are examples that use a internally generated PSF to get started quickly, however, it is recommended that you provide your own PSF according to guidelines below.

Note that this will erase and replace a file named '`catalog`' which is the name of the ISO/IEC 15068-2 meta-data directory.

```
cd somepackage-1.0
swign -u "Your GPG Name" @- | tar tvf -
```

In this example `swign` generated a PSF since one was not supplied. Here is what it used.

```
swign --show-psf
distribution
dfiles dfiles

product
title somepackage version 1.0
description Source package for somepackage version 1.0
tag somepackage
revision 1.0
control_directory ""
fileset
tag somepackage-sources
control_directory ""
file_permissions -o jhl  -g jhl
directory .
file *
exclude catalog
```

If you already have a PSF named '`PSF`', here's how to use it with `swign`:

```
cd somepackage-1.0
swign -s PSF -u "Your GPG Name" @- | tar tvf -
```

The same package can be created with `swpackage`, however, it requires specifying more options and the archive is written by `swpackage` instead of `tar`, Here's how:

```
cd somepackage-1.0
swpackage -s PSF -gpg-name "Your GPG Name" \
--dir=somepackage-0.1 --sign --files @- |
tar tvf -
```

3.1 PSF Basics

For information about PSFs in general, see the info manual on the Swbis home page, or, the `sw` manual page. This section applies to all PSFs.

Here some basic information. The PSF consists of *object keywords* and *attribute keywords*. The most common *object keywords* are `distribution`, `bundle`, `vendor`, `product`, and `fileset`. All *attribute keywords* exist in an object context and some *attribute keywords* are used in several *object keywords* contexts. To disambiguate the following notation is used: *object_name.attribute*.

Comments are lines or parts of lines that begin with `#`. Whitespace in a PSF is not significant. Objects are terminated by the next object keyword. Unrecognized *attributes* are allowed but unrecognized *objects* are not allowed.

The `tag` attribute in all objects should not contain the following four characters ' ', ':', '.', ',' (space, colon, comma, period).

3.1.1 Name-Version-Release

Most GNU packages do not have a *Release* part, but how this is modeled in a PSF is described here anyway. The *Release* part is analogous to the `RPMTAG_RELEASE` and *debian_revision* attributes, hence GNU software releases do no have this part because GNU packages are the original 'upstream' releases relative to the packages of GNU/Linux distributions.

The *Name* becomes the *product.tag* attribute. The *Revision* becomes the *product.revision* attribute. The *Release* becomes the *product.vendor_tag* attribute.

3.1.2 The Distribution Object

In most PSFs this object can be left empty. An empty object consists of just the object keyword followed by a newline. Any control or package meta-data files that apply to the distribution can tbe included in this object.

For example:

```
distribution
  dfiles dfiles
  AUTHORS <AUTHORS
  COPYING <COPYING
```

sets the *dfiles* attribute to its default value of "dfiles". The '`AUTHORS`', and '`COPYING`' will be included as individual files in the package directory '`somepackage-0.1/catalog/dfiles`'.

3.1.3 The Vendor Object

Providing a `vendor` object is optional.

```
vendor
   the_term_vendor_is_misleading  true   #  True or False
   tag         tag                 #  Short name, Globally unique if possible
   title       title               #  Longer name
   description description         #  A Detailed Description
```

The `tag` and `vendor_tag` attributes should not contain a ' ', ':','.', ',' (space, colon, comma, period).

3.1.4 The vendor_tag Attribute

The *product*.`vendor_tag` attribute is version identifier attribute and is used to distinguish packages that have the same *product.tag* and *product.revision* attributes. It points to a *vendor* object with a matching *vendor.tag* attribute.

3.1.5 Package File Ownerships

File permissions can be set independent of the permissions of the source files. The default policy for `swign` is to use the current owner and group. For reasons explained later, setting the ownerships to 0/0 for the owner and group is helpful. This is done with the following line in the PSF.

```
file_permissions -o 0  -g 0
```

The resulting ownerships are equivalent to the GNU `tar` options `--numeric --owner=root --group=root`.

3.2 PSFs for Source Packages

Here is an example PSF for the '`somepackage`' package, version 1.0.

```
distribution
product
title The somepackage package
description Source package for somepackage
tag somepackage
revision 1.0
control_directory ""
fileset
tag somepackage-sources
control_directory ""
file_permissions -o 0  -g 0
directory .
file *
exclude catalog
```

`swign` version 0.483 and later has a attribute replacement feature for the *product.tag* and *revision* attributes. They are determined from the current directory which must have the form *tag-revision*. The replacement strings are `%__tag` and `%__revision`. Hence here is a file, call it '`PSF.in`', which will work for any future revision.

```
# PSF.in  -- 'swign' Input file
distribution
product
title The somepackage package
description Source package for somepackage
tag %__tag
revision %__revision
control_directory ""
fileset
tag somepackage-sources
control_directory ""
file_permissions -o 0  -g 0
directory .
file *
exclude catalog
```

Here's how to use 'PSF.in'

```
cd somepackage-1.0
swign -s PSF.in -u "Your GPG Name" @- | tar tvf -
# -or -
cat PSF.in | swign -s - -u "Your GPG Name" @- | tar tvf -
```

4 Verifying the Distribution

The swbis signature verification program, `swverify`, will verify a package in two forms 1) as a tar archive file, and 2) as a unpacked archive. The distribution can also be verified manually using the existing GNU tools `tar`, `gpg`, `md5sum` and `sha1sum` and a Ext2 compatible file system. Verifying a distribution requires comparing the archive digests (md5 and sha1) with the digests present in the authenticated GPG signed data stream.

4.1 Verifying the Tar Archive File

The `swverify` verifies the package in memory without installing the package in file system. If a package is signed, it will have the following files:

```
<path>/catalog/
<path>/catalog/INDEX
...
<path>/catalog/<dfiles>/md5sum
<path>/catalog/<dfiles>/sha1sum
<path>/catalog/<dfiles>/sig_header
<path>/catalog/<dfiles>/signature
...
```

For example:

```
swverify -d  @- <somepackage-1.0.tar.gz
    # - or -
swverify <somepackage-1.0.tar.gz
```

4.2 Verifying the Unpacked Archive

The ability to verify the unpacked form is subject to several limitations, chief among them is the package must unpack into a single directory, verification then takes place on that directory.

For example

```
tar zxpf somepackage-1.0.tar.gz
swverify -d  @:somepackage-1.0
```

Verifying in this way requires that `tar` be able to re-create the exact byte stream that existed in the original distribution.

There are many constraints on the ability to verify the unpacked archive. These restrictions do not apply when verifying the archive file itself. Here they are:

- The file system must order directory entries like the Ext2 file system. (Ext3 file systems have this compatibility if dir_indexes are turned off. e.g. tune2fs -O ^dir_index /dev/device).
- The package must unpack into a single directory.
- The version of GNU `tar` must be compatible with the `swpackage` version used to make the package.
- The file owners in the package are present on the system with the same ids.

- Whether the package has file names longer than 99 bytes. (There have been intermittent deviations with GNU tar for certain long file names.)
- The package contains '`checkdigest`' script '`<path>/catalog/dfiles/checkdigest`'
- The package contains distribution file list '`<path>/catalog/dfiles/files`' (if the checkdigest script requires it, which it should).

4.2.1 The '`checkdigest`' script

The `checkdigest` script is an implementation extension verification hook. `swverify` will execute it after verifying the GPG signature and `swverify` exits with its exit status. It is intended to be a shell script that verifies the unpacked archive using existing GNU tools using the techniques described in the next section "Verifying Using Existing GNU Tools".

The file '`checkdigest.sh`' from the swbis distribution will work for any package.

To include a `checkdigest` script in the package, add the following line to the `distribution` object in the PSF.

```
checkdigest </usr/local/opt/src/checkdigest.sh  # For Example
```

4.3 Verifying Using Existing GNU Tools

Verifying manually is subject to the same constraints as verifying the unpacked archive, except for the existence of the '`checkdigest`' script and file list file '`catalog/dfiles/files`'. The steps below that check the payload message digests are typically the checks the '`checkdigest`' script would perform.

The first step is to unpack.

```
rm -fr somepackage-1.0
tar zxpf somepackage-1.0.tar.gz
```

The next step is try to re-create the signed byte stream and verify with `gpg` like this:

```
tar cf - --format=ustar -b1 --numeric --owner=root --group=root \
--exclude=catalog/dfiles/signature \
somepackage-1.0/catalog  |
gpg --verify somepackage-1.0/catalog/dfiles/signature -
```

Experimenting with the `--format`, `--numeric`, `--owner`, and `--group` options may be required to get a authentic byte stream. These options depend on how the distribution was created, specifically, the `swign --format` option and the PSF `file_permissions` directive. This is why a consistent file permissions policy and tar archive format are important.

Next, try to re-create the payload byte streams like this:

```
tar cf - --format=ustar -b1 --numeric --owner=root --group=root \
--exclude=somepackage-1.0/catalog \
--exclude=somepackage-1.0/catalog/\* somepackage-1.0 | md5sum
```

Then compare this md5 to the contents of '`somepackage-1.0/catalog/dfiles/md5sum`'. Do the same thing for the sha1 digest. If the package contains a symbolic link then you will not be able to re-create these digests because the modification time cannot be preserved for this file type. This may be a good reason source packages not contain symbolic links.

5 Adding New Signatures

If a package has a signature, the signature can be replaced or a new signature can be added keeping the old one.

Currently, swbis does not have a utility to make this easy, however, one is planned. The `swinstall` and `swverify` command currently support multiple signatures.

To replace the signature, all that is required is to replace the data part of the '`<path>/catalog/dfiles/signature`' file or add a new archive member, using the same tar header, placing it before or after the existing signature member.

The signature itself must be formatted in a particular way and have a length in bytes that matches its tar header, this is currently 1024 bytes. Every signature must have the same tar header (i.e. same name) and this tar header is stored in the '`<path>/catalog/dfiles/sig_header`' file.

Hence, to make a new signature member, take the data part of '`<path>/catalog/dfiles/sig_header`' (512 bytes) and append the 1024 bytes of the properly formatted signature, replace or add these 1536 bytes in the archive (this currently must be done by manually splitting the file into pieces, then concatenating it back together or by using a binary editor).

Fortunately, swbis does have a utility to reproduce the signed data. `gpg` and `dd` will be used to make a signature and format it like this:

```
# First, grab the sig_header
tar zxpf somepackage-1.0.tar.gz -O \*/catalog/dfiles/sig_header |
dd bs=512 count=1 of=/tmp/newsig

# Now, make the new signature
# Note:  'swverify -WC' writes the signed data to stdout
swverify -WC <somepackage-1.0.tar.gz |
gpg --armor -sb -o - | dd bs=1024 conv=sync count=1 >>/tmp/newsig
```

For example, a package with two (2) signatures looks like this:

```
somepackage-1.0/catalog/
somepackage-1.0/catalog/INDEX
...
somepackage-1.0/catalog/dfiles/sig_header
somepackage-1.0/catalog/dfiles/signature
somepackage-1.0/catalog/dfiles/signature
...
```

Since all but the last signature is lost when unpacked, the last signature should be the considered the primary one.

6 Guidelines for GNU Source Packages

Here are itemized guidelines for GNU packages:

- Use GNU tar version 1.15.x, GNU swbis version 0.483 or later versions.
- Use the default `swign` format option '`--format=ustar`'. This corresponds to tar option '`--format=ustar`'.
- Do not include symbolic or hard links in the distribution, make them when configuring if needed.
- Try not to make file names longer than 99 bytes because this will make verification of the unpacked directory form a little problematic until some bugs in swbis and tar are fully converged.
- Set the file ownerships in the package to numeric root/root. Using the `file_permissions -o 0 -g 0` directive in the PSF is the easiest way to do this.
- Do include a '`checkdigest`' script. The file '`./bin/checkdigest.sh`' from the swbis distribution should work for any package.

Here is an example PSF.

```
# PSF.in  -- Example 'swign' Input file for GNU packages.
# Occurrences of %__tag and %__revision will be replaced
# by values determined from the name of the current directory
# that has the form:  tag-revision
distribution
  # dfiles dfiles            # dfiles is the default
  AUTHORS <./AUTHORS         # optional
  COPYING <./COPYING         # optional
  checkdigest <./var/checkdigest.sh  # or wherever it is on your system
  tag %__tag-%__revision   # Optional, this will set '--dir' option of
                             # of swpackage.
vendor
   the_term_vendor_is_misleading True
   tag GNU
   title GNU's Not Unix
description "The GNU Project was launched in 1984 to develop a complete UNIX-like
operating system which is free software: free as in freedom, not price.
See http://www.gnu.org."

product
  title GNU %__tag
  vendor_tag GNU
  description Source package for %__tag  # More can be added
  tag %__tag                 # This is the package name
  revision %__revision       # This is the package version
  control_directory ""
  fileset
     tag source
     control_directory ""
     file_permissions -o 0  -g 0
     directory .
     file *
     # exclude RCS   # Not supported yet by swign
     # exclude CVS   # Not supported yet by swign
     exclude catalog  # required
```

Here is how to use the PSF to create a package with an embedded GPG signature.

```
cd somepackage-1.0
swign -s PSF.in  -u "Your GPG name" @- | gzip -9 >../somepackage-1.0.tar.gz
# Then do a couple quick tests
swverify -d @- <../somepackage-1.0.tar.gz

# If a checkdigest script was included and the file system is Ext2
# compatible then the following should work, try it
swverify -d @.

# For some newer file system you must use the --order-catalog option
```

```
swverify --order-catalog -d @.
```

To make a nearly identical package using `swpackage`

```
# First, the replacement macros must be processed by swign
swign -s PSF.in --show-psf |
swpackage -s - --gpg-name="Your GPG name" \
--dir-owner=0 --dir-group=0 --files --sign @- |
gzip -9 >../somepackage-1.0.tar.gz
```

There are differences between `swign` and `swpackage`. `swign` uses `swpackage` but uses `tar` to write the final archive hence it is more fail safe against bugs. `swign` modifies the '`./catalog/`' making '`.`' immediately verifiable with `swverify` and is simpler to use.

That's it. You now have a tar archive with one or more embedded signatures, that is created using `tar`, is verifiable with existing tools, compatible with current practice, and conforms to the POSIX packaging standard.

7 Using Automake and Swbis

This section describes an Automake target to include in the top level Makefile.am file. To use it, you must get your package working with Automake and able to create a distribution using one of the standard distribution targets such as `dist-gzip` that is already part of Automake.

This example target is called `dist-swbis`. The target is designed to be symmetric with the other standard Automake targets such as `dist-gzip`. It uses the `swign` program. The '`PSF.in`' file must use the `%__tag` and `%__revision` macros described above. The passphrase input options and identity is controlled by environment variables: SWPACKAGEPASSFD, GNUPGNAME, GNUPGHOME.

```
dist-swbis: distdir
        (cd $(distdir) && swign -s PSF.in --name-version=$(distdir) @-) | GZIP=$(GZIP_ENV) g
        $(sw_am__remove_distdir)
# Provide am__remove_distdir ourselves since am__remove_distdir may be a
# private automake variable.
sw_am__remove_distdir = \
  { test ! -d $(distdir) \
      || { find $(distdir) -type d ! -perm -200 -exec chmod u+w {} ';' \
               && rm -fr $(distdir); }; }
```

Here is an example invocation using the environment variable controls:

```
export SWPACKAGEPASSFD=agent; export GNUPGNAME="Your Name"; make dist-swbis
```

To input your passphrase from the tty, unset SWPACKAGEPASSFD or set it to "tty".

The result should be a file named '*distdir*`.tar.gz`' that has the same layout as the package produced by `dist-gzip` excpept this package will carry around your GPG signature in the additional ./catalog meta-data directory.

The file should then be verified:

```
swverify -d @- <distdir.tar.gz
```

That's it.

8 Using CVS and Swbis

This section describes how to use swbis to place GPG signatures into a source code management repository such as CVS. The application of swbis simply involves adding the '`./catalog/`' directory and its contents to the repository and is not specific to any particular SCM. The files in the '`./catalog/`' directory are either directories or ascii text regular files.

The first step is to perfectly sync-up with the repository. Empty directories should be removed and created on the fly by the Makefiles. Stray junk files in the working directory and repository need to be deleted from both. Failure to do this will result in failed verification although the partial success can still be useful. (Also the RCS style file Id's used by CVS may interfere the verification of the file digests.)

Step two is to initialize, add, and commit the '`./catalog/`' directory in the top level module. Just make the regular files empty for now. The order does not matter. The files are:

```
catalog/
catalog/INDEX
catalog/dfiles/
catalog/dfiles/INFO
catalog/dfiles/checkdigest
catalog/dfiles/md5sum
catalog/dfiles/sha1sum
catalog/dfiles/adjunct_md5sum
catalog/dfiles/files
catalog/dfiles/sig_header
catalog/dfiles/signature
catalog/pfiles/
catalog/pfiles/INFO
catalog/INFO
```

Next, checkout the '`./catalog/`' directory. Treat it just like any other directory except you will be using the `swign` command to generate its contents. Then at any point of your choosing sign your working directory by running `swign` and then commit all of your changes to the repository including the '`./catalog/`' directory.

Here's how to sign:

```
make distclean;
SWPACKAGEPASSFD=agent; GNUPGNAME="Your Name"  swign --name-version=module_name-
1.2.3 -s PSF.in --no-remove @.
```

This `swign` invocation will only alter files in '`./catalog/`'.

Note that the `--no-remove` option is required as this prevents the SCM control files from being deleted. Also, the `--name-version` option is required.

The '`PSF.in`' file has several specializatons. The `%__tag` and `%__revision` macros must be used and the `exclude` directive must exclude the SCM's working directory control files. The '`PSF.in`' file must also specify a `checkdigest` script as this is required to verify the directory form of a package. The '`checkdigest.sh`' file from swbis version 0.496 is a working example of this script.

Next, you should tag this point so it can retrieved in the future. Now, export (to exclude the SCM's control files) the module to a new directory and run `swverify` with the `--scm` option (The swverify version must be at least 0.496).

```
cvs export -r your_tag_name module_name
cd module_name
swverify -d --scm @.
```

That's it.