DPM Package Format

Overview

This document outlines the DPM package format.

```
package_name-1.0.x86_64.dpm
├── contents(.tgz)
│   ├── etc/
│   │   └── myapp/
│   │       └── conf.d/
│   │           └── main.config
│   └── usr/
│       └── bin/
│           └── myapp
├── hooks(.tgz)
│   ├── POST-INSTALL
│   ├── POST-INSTALL_ROLLBACK
│   ├── POST-REMOVE
│   ├── POST-REMOVE_ROLLBACK
│   ├── POST-UPDATE
│   ├── POST-UPDATE_ROLLBACK
│   ├── PRE-INSTALL
│   ├── PRE-INSTALL_ROLLBACK
│   ├── PRE-REMOVE
│   ├── PRE-REMOVE_ROLLBACK
│   ├── PRE-UPDATE
│   └── PRE-UPDATE_ROLLBACK
├── metadata(.tgz)
│   ├── ARCHITECTURE
│   ├── AUTHOR
│   ├── MAINTAINER
│   ├── DEPENDENCIES
│   ├── DESCRIPTION
│   ├── CONTENTS_MANIFEST_DIGEST
│   ├── LICENSE
│   ├── NAME
│   ├── PACKAGE_DIGEST
│   ├── HOOKS_DIGEST
```

```
│   ├── PROVIDES
│   ├── REPLACES
│   ├── SOURCE
│   ├── CHANGELOG
│   └── VERSION
└── [signatures(.tgz)]
    ├── contents.gpg.signature
    ├── hooks.gpg.signature
    └── metadata.gpg.signature
```

From the outside, a DPM package is just a gzipped tarball with a DPM filename extension.

Inside that extracted tarball, you'll see 4 more bundled inside it, named:

1. `signatures`
2. `contents`
3. `metadata`
4. `hooks`

These archives all serve different purposes and their content formats are very simple.

The `signatures`, `metadata`, and `hooks` archives are generated by the DPM package creation process largely automatically. After metadata is generated, the package creator can tailor items in the metadata prior to finalizing the package and wrapping it.

The package creation process is out of scope for this document, however, and will be discussed in more detail in dedicated documents on the workflow.

signatures

The signatures archive MAY contain GPG-signed cryptographic checksums of the other archives as signature files. These are optionally generated automatically during the package creation process, by the DPM package creation utility. The default checksum algorithm is SHA-256, but is configurable to future-proof DPM against evolving threat landscapes.

These signatures ARE NOT used to ***encrypt*** ANY payload of the package or provide ANY form of confidentiality to the package. They are *purely* used for ***integrity validation***. In other words, when present, the 3 signatures CAN BE used to validate that the contents of the package (the files being installed), the metadata (information about the package), and the hook scripts (operational triggers) have not been tampered with.

The method for this is by computing a local cryptographic checksum of the corresponding archive, identified by relying on reserved names in the package structure, and validating the signature of the archive checksum by comparing it against that locally computed version, using the package creator's public key stored in `/etc/dpm/keys/`.

The public key in `/etc/dpm/keys/` is usually populated at a prior point by DPM's companion utility, `DON` by pulling it from a remote repository where the package is located. Public keys can also be manually imported from a local file to this location using the DPM utility. This behaviour will be defined more clearly in the documentation for `DON`.

DPM can be configured to require or not require GPG integrity validation using these signatures.

If GPG integrity validation is disabled, it will prohibit the use of anything in this directory. Integrity validation can still be performed against the package metadata in the DPM database or backing tree during and after installation, but it will potentially be vulnerable to tampering and so is fundamentally less secure.

If GPG integrity validation is required, but GPG signatures are not present, DPM will refuse to install the package.

If GPG integrity validation is required, but one of the nested archives' signature fails validation, `DPM will refuse to install the package with an error message explaining which piece failed signature checks, and a warning that this could indicate tampering.

contents

The contents archive contains the actual files being deployed by the package. As

you can likely see already from the tree diagram, files in this archive are staged in their absolute path, adjusted from the system's root filesystem. So, for example, this package has an '`etc/myapp/conf.d/main.config`'. This is meant to arrive on the installing machine at `'/etc/myapp/conf.d/main.config'` when installed.

In some situations, it is necessary to override the top level directory target from the root of the filesystem to another directory, such as when using DPM to populate a sysroot when building a CHROOT environment, a container, or a new operating system. DPM provides that target override functionality as a commandline argument.

There's really not much to this section. It is meant to be designed in such a way that all the package creator really has to do is stage a directory tree with files where they will need to go when they end up installed on the target machines installing the package, and then populate the metadata and optionally sign the package.

NOTE: For various efficiency reasons, the contents archive is added last when creating the package, so that metadata, hook, and signature extraction can occur more rapidly due to the sequential compression method used.

hooks

Hooks in DPM are very similar to hooks in `Git`. In the case of DPM, there are reserved filenames in this directory that represent operations that DPM performs, and are triggered when that action is performed.

For example, if something needs to happen on the system prior to install, the `PRE-INSTALL` hook is triggered prior to the package installation, so, the contents of what is in that file will be executed as a shell script. Conversely, if that fails or returns a non-zero exit code, the `PRE-INSTALL_ROLLBACK` hook is triggered, so that cleanup from whatever was being done there can occur.

This is powerful and dangerous, as your rollback script must be able to check for the conditions it needs to clean up, so be aware of that if you decide to make this complicated.

You may not add files to the hooks archive. This is by design to keep you from bundling entire unnecessary runtimes inside the package.

The burden of idempotency of these operational triggers is deferred to the package creator.

metadata

To make things easy, the DPM creation process automatically creates the metadata directory and contents so that package setup is straight forward without much guesswork. Once generated, the metadata archive contains a list of of reserved file names:

- CONTENTS_MANIFEST_DIGEST
- DEPENDENCIES
- PROVIDES
- REPLACES
- PACKAGE_DIGEST
- ARCHITECTURE
- AUTHOR
- DESCRIPTION
- LICENSE
- NAME
- SOURCE
- VERSION

CONTENTS_MANIFEST_DIGEST

The `CONTENTS_MANIFEST_DIGEST` reads like a table of contents and lists a few whitespace delimited columns containing details about every file to be installed by the package: `control designation`, `file checksum`, and `absolute filepath`.

Using the same example package structure, this is what the content digest looks like:

```
C $CHECKSUM 0755 user:group /usr/bin/myapp
```

```
N $CHECKSUM 0644 user:group /etc/myapp/conf.d/main.config
```

The first column, `control designation`, determines whether the file is designated as “controlled” (‘`C`‘) or “not controlled” (‘`N`‘). During package updates and package removals, “controlled” files are replaced or removed. Inversely, during updates and removals, “not controlled” files (‘`N`‘) are left in place. The intended use case for this is configuration files or any other data files that should be left alone during updates. In the event that a “not controlled” file produces a file conflict with what is being deployed, the file in the updated package will be saved to the same location with an appended suffix `.dpmnew`. This should be familiar behaviour. This ensures that reinstallation or updates of packages will not overwrite configuration files that can sometimes be difficult to recreate or that recreating would expand the scope of a standard patching window to reduce touch time on the host. By default, during package metadata generation, all files in the contents are designated as “controlled”, and the package creator will need to manually specify files that should be “not controlled”. This is deliberate to reduce the amount of time it takes to spec a package with the added benefit of producing a soft captological barrier to the creation of an overabundance of files on the system which are not controlled by the package manager but which are installed by the package manager.

The second column is for the cryptographic `checksum` of the file. By default this is a SHA-256 fingerprint of the file, but the algorithm is configurable to future-proof DPM against an evolving cryptographic landscape. It should be noted that this checksum is only calculated against the file contents.

The third column is the `permissions` intended for the file after install. This is prepopulated with what exists in the contents at the time of package metadata generation time and exists in the format for easy override.

The fourth column is the combination of `user` and `group` separated by a `colon character` that should own the file. This user:group pair is also prepopulated at package metadata generation time and also is easy to override simply by changing the value before finalizing the package creation.

The fifth column is the derived `path` of the file when the root of the filesystem is not modified. It should be the path of the file in the contents archive prefixed with a preceding forward-slash.

DEPENDENCIES

The `dependencies` file is a line-delimited list of expressions that serve as rules for dependencies on other packages.

If there are no dependencies, this file can be blank.

Following suit with our example, here's some sample contents of a DEPENDENCIES file:

```
glibc > 2.21
glibc < 2.39
libstdc++ > 0
```

The pattern for these rules is deliberately inflexible:

1. Rules are delimited by newline.
2. Rules are 3-term expressions.
3. The first term of the expression is the package name that is required.
4. The second term is an operator. Available operators are '<', '>', '==', '!=', '>=', '<='.
5. The third and final term is a version number.

So, reading our example, this package requires a version of glibc greater than 2.21, but less than 2.39, and requires any version of libstdc++ to be installed.

Regarding the package name, it can be substituted for names listed in the "`REPLACES`" or "`PROVIDES`" metadata for another package. This is useful for metapackages, or virtual packages, or transitionary packages that replace legacy packages under a new name. This concept will be explained in more detail in the section dedicated for those files.

PROVIDES

The `provides` file is a line-delimited list of "virtual package names" or "aliases" for the package that can be used to identify the package.

For example, let's say you have a collection of packages that all represent different programs in a larger system with many components. This is actually

seen often in Desktop Environments. You could have a mostly empty package called, for example, `XFCE`, which is a popular desktop environment that lists `xfce-terminal` — which is its terminal emulator and is packaged separately — as a dependency. Then, you can list the other packages that make up the XFCE suite of packages, all listed as dependencies to your mostly empty XFCE package.

In that mostly empty file, you could list `XFCE` in the provides file so that other packages can depend on the whole XFCE environment being installed or not.

Technically you could do this with just the package name, but this is a cleaner way to do it, so that you could have the name of the package be `xfce-desktop-suite` that "provides" `xfce` or `xfce4-desktop`, allowing a package to have multiple aliases.

This also allows you to have interchangeable components that meet the same dependencies be used in an intermixed fashion without creating package naming conflicts. For example, let's say a package depended on `xfce4-terminal` or `sakura` terminal to be installed and could work with either, both packages could list `standalone-terminal` in its `provides` list and your package that just needed one of them could use the one you preferred because the dependency was met.

This feature is prone to complexity and can do some powerful things if used by package maintainers correctly, such as creating system profiles for complex metapackage installations that are highly tailorized to the system.

This also, for example, allows `apache2` and `httpd` to be used interchangeably in some cases.

REPLACES

The replaces file serves a different purpose. This is a newline delimited list of packages that the package replaces altogether and would meet as a dependency to another package in those packages' stead.

So, if a package is renamed between versions, you would list the old name in this file so that installing this package would meet the dependency listed if another package uses the old name still. This allows multiple packages to be listed for when projects consolidate functionality from multiple libraries while maintaing

some degree of api compatibility, so that one package meets what used to be the dependencies that several packages used to meet.

PACKAGE_DIGEST

The `PACKAGE_DIGEST` file is a cryptographic checksum of the sorted entries in `CONTENTS_MANIFEST_DIGEST`. This is used for file integrity validation and tamper indicators when evaluating the health of installed packages.

HOOKS_DIGEST

Like the PACKAGE_DIGEST, the `HOOKS_DIGEST` file is a cryptographic checksum of the files in the `HOOKS` archive for integrity validation.

ARCHITECTURE

This is a one line file that contains a string identifier for the package architecture if applicable. (example: `x86_64`)

CHANGELOG

The `CHANGELOG` is a multiline list of changes to the package over time as versions increment. It is not meant to be a changelog for the software itself but for the packaging process for that software over time.

New entries begin with an asterisk (`'*'`), a timestamp in YYYY-MM-DD, a name, and an email address.

AUTHOR, MAINTAINER DESCRIPTION, LICENSE, NAME, SOURCE, VERSION

Like the `ARCHITECTURE` file, these are all, except `DESCRIPTION`, one-line files that contain the values their names represent.

DESCRIPTION can be multiple lines.

In the case of `AUTHOR` and `MAINTAINER` it's highly recommended to use a name with an email address (e.g. "`Chris Punches <chris.punches@silogroup.org>`") and in the case of SOURCE it's highly

recommended to use a URL (e.g. "https://source.silogroup.org/SILO-GROUP/rex").

Package Maintainer Guidance: SELinux

While technically not part of the format specification, in some cases you may need to create new SELinux file context definitions or type enforcement rules for your paths in your package.

DPM is agnostic to these.

The recommended path for solving this is to have package creators who need this create an optional second sidecar package exclusively for SELinux policy deployment, as is done in many RPM-based distributions.

So, for `myapp-1.0.x86_64.dpm` you might also have a `myapp_selinux-1.0.noarch.dpm`.

This `myapp_selinux` package would contain the respective `.te` and `.cf` files being deployed to accompany your package for selinux-enabled hosts.

It is not recommended to use `DEPENDENCIES` on this secondary package structure, so that if your target system does not use the same paths, or doesn't have SELinux infrastructure installed, you can still use the main program without issue, and if the target machine does have that infrastructure they can solve their label issue by installing the optional selinux sidecar package.

This approach provides the added benefit of clearly separating SELinux policy management from individual application deployments to allow coverage of a much broader range of target systems.