Application layout

Contents

Application bundles in the Apertis system may require several categories of storage, and to be able to write correct AppArmor profiles, we need to be able to restrict each of those categories of storage to a known directory.

This document is intended to update and partially supersede discussions of storage locations in theapplications[1] andsystem updates and rollback[2] design documents.

This document describes and provides rationale for the layout of and file types within an application bundle, suggested future directions, and details of functionality that is not necessarily long-term stable.

Requirements

Static files

- Most application bundles will contain one or more executable programs[3], in the form of either compiled machine code or scripts. These are read-only and executable, and are updated when the bundle is updated (and at no other time).
  - Some of these programs are designed to be run directly by a user. These are traditionally installed in `/usr/bin` on Unix systems. Other programs are *supporting programs*, designed to be run internally by programs or libraries. These are traditionally installed in `/usr/libexec` (or sometimes `/usr/lib`) on Unix systems. Apertis does not require a technical distinction between these categories of

[1] https://www.apertis.org/concepts/applications/
[2] https://www.apertis.org/concepts/system-updates-and-rollback/
[3] https://www.apertis.org/glossary/#program

program, but it would be convenient for them to be installed in a layout similar to the traditional one.

- Application bundles that contain compiled executables may contain *private shared libraries*, in addition to those provided by the platform[4], to support the executable. These are read-only ELF shared libraries, and are updated when the bundle is updated.
- Application bundles may contain support dynamically-loaded *plugins*.
- Application bundles may contain static *resource files* such as `.gresource` resource bundles, icons, fonts, or sample content. These are read-only, and are updated when the bundle is updated.
  - Where possible, application bundles should embed resources in the executable or library using GResource[5]. However, there are some situations in which this might not be possible, which will result in storing resource files in the filesystem:
    * if the application will load the resource via an API that is not compatible with GResource, but requires a real file
    * if the resource is extremely large
    * if the resource will be read by other programs, such as the icon that will be used by the app-launcher, the .desktop file describing an entry point, or D-Bus service files (used by dbus-daemon)
  - If a separate `.gresource` file is used, for example for programs written in JavaScript or Python, then that file needs to be stored somewhere.

Variable files

- The programs in application bundles may save variable data (configuration, state and/or cached files) for each user[6] (Applications design - Data Storage[7]).
  - *Configuration* is any setting or preference for which there is a reasonable default value. If configuration is deleted, the expected result is that the user is annoyed by the preference being reset, but nothing important has been lost.
  - *Cached files* are files that have a canonical version stored elsewhere, and so can be deleted at any time without any effect, other than performance, resource usage, or limited functionality in the absence of an Internet connection. For example, a client for "tile map"services like Google Maps or OpenStreetMap should store map tiles in its cache directory. If cached files are deleted, the expected result is that the system is slower or less featureful until an automated process can refill the cache.

[4] https://www.apertis.org/glossary/#platform
[5] https://developer.gnome.org/gio/stable/GResource.html
[6] https://www.apertis.org/glossary/#user
[7] https://www.apertis.org/concepts/applications/#data-storage

  - Non-configuration, non-cache data includes documents written by the user, database-like content such as a contact list or address book, license keys, and other unrecoverable data. It is usually considered valuable to the user and should not be deleted, except on the user's request. If non-configuration, non-cache data is unintentionally deleted, the expected result is that the user will try to restore it from a backup.
- Newport needs to be able to write downloaded files to a designated directory owned by the application bundle.
  - Because downloads might contain private information, Newport must download to a user- and bundle-specific location.

Upgrade, rollback, reset and uninstall

Store applications Suppose we have a store application bundle[8], Shopping List version 23, which stores each user's grocery list in a flat file. A new version 24 becomes available; this version stores each user's grocery list in a SQLite database.

- Shopping List can be installed and upgraded. This must be relatively rapid.
- Before upgrade from version 23 to version 24, the system should make version 23 save its state and exit, terminating it forcibly if necessary, so that processes from version 23 do not observe version 24 files or any intermediate state, which would be likely to break their assumptions and cause a crash.
  - This matches the user experience seen on Android: graphical and background processes from an upgraded `.apk` are terminated during upgrade.
- After upgrade from version 23 to version 24, the current data will still be in the version 23 format (a flat file).
- When a user runs version 24, the application bundle may convert the data to version 24 format if desired. This is the application author's choice.
- If a user rolls back Shopping List from version 24 to version 23, it is the application's responsibility to handle the now-converted saved data.
- Shopping List can be uninstalled. This must be relatively rapid. (Applications design[9] §4.1.4, "Store Applications —Removal")
- When Shopping List is uninstalled from the system, the system must remove all associated data, for all users.

[8] https://www.apertis.org/glossary/#store-application-bundle
[9] https://www.apertis.org/concepts/applications/

  - If a multi-user system emulates a per-user choice of apps by hiding or showing apps separately on a per-user basis, it should delete user data at the expected time: if user 1 “uninstalls”Shopping List, but user 2 still wants it installed, the system may delete user 1’s data immediately.

- **Unresolved:** Are downloads rolled back?

Built-in applications By definition, built-in application bundles[10] are part of the same filesystem image as the platform. They are upgraded and/or rolled back with the platform. Suppose platform version 2 has a built-in application bundle, Browser version 17. A new platform version 3 becomes available, containing Browser version 18.

- The platform can be upgraded. This does not need to be particularly rapid: a platform upgrade is a major operation which requires rebooting, etc. anyway.
- Immediately after upgrade, the data is still in the format used by Browser version 17.
- Uninstalling a built-in application bundle is not possible (Applications design[11] §4.2.3, “Built-in Applications —Removal”) but it should be possible to delete all of its variable data, with the same practical result as if an equivalent store application bundle had been uninstalled and immediately reinstalled.
- Cache files for built-in applications are treated the same as cache files for Store applications, above.

Global operations User accounts can be created and/or deleted.

- Deleting a user account does not need to be as rapid as uninstalling an application bundle. It should delete that user’s per-user data in all application bundles.

A “data reset”operation affects the entire system. It clears everything.

- A “data reset”does not need to be as rapid as uninstalling an application bundle. It should delete all variable data in each application bundle, and all variable data that is shared by application bundles.

Unresolved: Does data reset uninstall apps?

Security and privacy considerations

- Given a bundle ID and whether the program is part of a built-in or store application, it must be easy to determine where it may write. Again, this is for services like Newport.

[10] https://www.apertis.org/glossary/#built-in-application-bundle
[11] https://www.apertis.org/concepts/applications/

- The set of installed store application bundles is considered to be confidential, therefore typical application bundles (with no special permissions) must not be able to enumerate the entry points, systemd units, D-Bus services, icons etc. provided by store application bundles. A permission flag could be provided to make an exception to this rule, for example for an application-launcher application like Android's Trebuchet.
- **Unresolved:** Are built-in bundles visible to all?

Miscellaneous

- Directory names should be namespaced by reversed domain names[12], so that it is not a problem if two different vendors produce an app-bundle with a generic name like "Navigation".
- Where possible, functions in standard open-source libraries in our stack, such as GLib and Gtk, should "do the right thing". For example, `g_get_cache_dir()` should continue to be the correct function to call to get a parent directory for an application's cache.
- Where possible, functions in other standard open-source libraries such as Qt and SDL should generally also behave as we would want. This can be achieved by making use of common Linux conventions such as the XDG Base Directory specification[13] where possible. However, these other libraries are likely to have less strong integration with the Apertis platform in general, so there may be pragmatic exceptions to this principle: full compatibility with these libraries is a low priority.

Writing application bundles

Application bundle authors should refer to the Flatpak documentation[14] for details on building Flatpak application bundles.

Unresolved design questions

Does data reset uninstall apps?

Does a data reset leave the installed store apps installed, or does it uninstall them all? (In other words, does it leave store apps'static files intact, or does it delete them?)

Appendix: comparison with other systems

Desktop Linux (packaged apps)

There are many possibilities, but a common coding standard looks like this:

- Main programs are installed in `$bindir` (which is set to `/usr/bin`)

[12] https://www.apertis.org/glossary/#reversed-domain-name
[13] http://standards.freedesktop.org/basedir-spec/basedir-spec-latest.html
[14] https://docs.flatpak.org/

- Supporting programs are installed in `$libexecdir` (which is set to either `/usr/libexec` or `/usr/lib`), often in a subdirectory per application package
- Public shared libraries are installed in `$libdir` (which is set to either `/usr/lib` or `/usr/lib64` or `/usr/lib/$architecture`)
  - Plugins are installed in a subdirectory of `$libdir`
  - Private shared libraries are installed in a subdirectory of `$libdir`
- `.gresource` resource bundles (and any resource files that cannot use GResource) are installed in `$datadir`, which is set to `/usr/share`
- System-level configuration is installed in a subdirectory of `$sysconfdir`, which is set to `/etc`
- System-level variable data is installed in `$localstatedir/lib/$package` and `$localstatedir/cache/$package`, with `$localstatedir` set to `/var`
- There is normally no technical protection between apps, but per-user variable data is stored according to the XDG Base Directory specification[15] in:
  - `$XDG_CONFIG_HOME/$package`, defaulting to `/home/$username/.config/$package`, where $username is the user's login name and $package is the short name of the application or package
  - `$XDG_DATA_HOME/$package`, defaulting to `/home/$username/.local/share/$package`
  - `$XDG_CACHE_HOME/$package`, defaulting to `/home/$username/.cache/$package`
- The user's home directory, normally `/home/$username`, is shared between apps but private to the user
  - It is usually technically possible for one app to alter another app's subdirectories of `$XDG_CONFIG_HOME` etc.
- There is no standard location that can be read and written by all users, other than temporary directories which are not intended to be shared

Debian Policy §9.1 "File system hierarchy"[16] describes the policy followed on Debian and Ubuntu systems for non-user-specific data. It references the Filesystem Hierarchy Standard, version 2.3[17].

Similar documents:

- The Filesystem Hierarchy Standard, version 3.0[18] has not yet been adopted by Debian Policy.
- The GNU Coding Standards[19] use a similar layout by default.
- systemd's proposals for file hierarchy[20] have been partially adopted by Linux distributions.

[15] http://standards.freedesktop.org/basedir-spec/basedir-spec-latest.html
[16] https://www.debian.org/doc/debian-policy/ch-opersys.html#s9.1
[17] http://www.pathname.com/fhs/pub/fhs-2.3.html
[18] http://refspecs.linuxfoundation.org/FHS_3.0/fhs/index.html
[19] https://www.gnu.org/prep/standards/html_node/Directory-Variables.html#Directory-Variables
[20] http://www.freedesktop.org/software/systemd/man/file-hierarchy.html

Android

- System app packages (the equivalent of our built-in application bundles[21]) are stored in `/system/app/$package.apk`
- Normal app packages (the equivalent of our store application bundles[22]) are stored in `/data/app/$package.apk`
- Private shared libraries and plugins (and, technically, any other supporting files) are automatically unpacked into `/data/data/$package/lib/` by the OS
- Resource files are loaded from inside the `.apk` file (analogous to GResource) instead of existing as files in the filesystem
- Per-user variable data is stored in `/data/data/$package/` on single-user devices
- Per-user variable data is stored in `/data/user/$user/$package/` on multi-user devices
- There is no location that is private to an app but shared between users.
- There is no location that is shared between apps but private to a user.
- `/sdcard` is shared between apps but not between users. Large data files such as music and videos are normally stored here.

systemd "revisiting Linux systems"proposal

The authors of systemd propose a structure like this[23]. At the time of writing, no implementations of this idea are known.

- The static files of application bundles are installed in a subvolume named `app:$bundle_id:$runtime:$architecture:$version`, where:
  - `$bundle_id` is a reversed domain name identifying the app bundle itself
  - `$runtime` identifies the set of runtime libraries needed by the application bundle (in our case it might be `org.apertis.r15_09`)
  - `$architecture` represents the CPU architecture
  - `$version` represents the version number
- That subvolume is mounted at `/opt/$bundle_id` in the app sandbox. The corresponding runtime is mounted at `/usr`.
- User-specific variable files are in a subvolume named, for example, `home:user:1000:1000` which is mounted at `/home/user`.
- System-level variable files go in `/etc` and `/var` as usual.
- There is currently no concrete proposal for a trust boundary between apps: all apps are assumed to have full access to `/home`.
- There is no location that is private to an app but shared between users.
- There is no location that is shared between apps and between users, other than removable media.

[21] https://www.apertis.org/glossary/#built-in-application-bundle
[22] https://www.apertis.org/glossary/#store-application-bundle
[23] http://0pointer.net/blog/revisiting-how-we-put-together-linux-systems.html

References

- Applications design document[24] (v0.5.4 used)
- Multimedia design document[25] (v0.5.4 used)
- Security design document[26] (v1.1.3 used)
- System Update and Rollback design document[27] (v1.6.2 used)

[24] https://www.apertis.org/concepts/applications/
[25] https://www.apertis.org/concepts/multimedia/
[26] https://www.apertis.org/concepts/security/
[27] https://www.apertis.org/concepts/system-updates-and-rollback/