File DAC Configuration

When adding file system objects and services to the build, such items frequently need separate unique IDs, known as Android IDs (AIDs). Currently, many resources such as files and services use core, Android-defined AIDs unnecessarily; in many cases you can use OEM-defined AIDs instead.

In earlier versions of Android, extending the AIDs mechanism used a device-specific **`android_filesystem_config.h`** file to specify the filesystem capabilities and/or custom OEM AIDs. However, this system was unintuitive as it did not support using nice names for OEM AIDs, requiring you to specify the raw numeric for user and group fields without a way to associate a friendly name with the numeric AID.

Android 8.0 and higher includes a new AIDs mechanism for extending filesystem capabilities. This new method has support for the following:

- Multiple source locations for configuration files (enables extensible build configurations).
- Build-time sanity checking of OEM AID values.
- Generation of a custom OEM AID header that can be used in source files as needed.
- Association of a friendly name with the actual OEM AID value. Supports non-numeric string arguments for user and group, i.e. "foo" instead of "2901".

Additional improvements include the removal of the **`android_ids[]`** array from **`system/core/include/private/android_filesystem_config.h`**. This array now exists in Bionic as a fully private generated array, with accessors via **`getpwnam()`** and **`getgrnam()`**. (This has the side effect of producing stable binaries as core AIDs are modified.) For tooling and a README file with more details, refer to **`build/make/tools/fs_config`**.

Note: While you can still use the filesystem override method from previous Android releases (#older), you cannot use it simultaneously with the new AIDs mechanism. Using the new mechanism whenever possible is recommended.

Adding Android IDs (AIDs)

Android 8.0 removes the **`android_ids[]`** array from the Android Open Source Project (AOSP). All AID-friendly names are instead generated from the **`system/core/include/private/android_filesystem_config.h`** header file when generating the Bionic **`android_ids[]`** array. Any **`define`** matching **`AID_*`** is picked up by the tooling and * becomes the lowercase name.

For example, in **`private/android_filesystem_config.h`**:

```
#define AID_SYSTEM 1000
```

Becomes:

- Friendly name: system
- uid: 1000
- gid: 1000

To add a new AOSP core AID, simply add the **`#define`** to the **`android_filesystem_config.h`** header file. The AID will be generated at build and made available to interfaces that use user and group arguments. The tooling validates the new AID is not within the APP or OEM ranges; it also respects changes to those ranges and should automatically reconfigure on changes or new OEM-reserved ranges.

Configuring AIDs

To enable the new AIDs mechanism, set **`TARGET_FS_CONFIG_GEN`** in the **`BoardConfig.mk`** file. This variable holds a list of configuration files, enabling you to append files as needed.

Caution: Don't use **`TARGET_FS_CONFIG_GEN`** with the older **`TARGET_ANDROID_FILESYSTEM_CONFIG_H`** method from older Android releases! You will get an error.

By convention, configuration files use the name **`config.fs`**, but in practice you can use any name. **`config.fs`** files are in the Python ConfigParser ini format (https://docs.python.org/2/library/configparser.html) and include a caps section (for configuring file system capabilities) and an AIDs section (for configuring OEM-specific AIDs).

Configuring the caps section

The caps section supports setting file system capabilities (http://man7.org/linux/man-pages/man7/capabilities.7.html) on filesystem objects within the

build (the filesystem itself must also support this functionality).

Because running a stable service as root in Android causes a Compatibility Test Suite (CTS) (https://source.android.com/compatibility/cts/index.html) failure, previous requirements for retaining a capability while running a process or service involved setting up capabilities then using `setuid`/`setgid` to a proper AID to run. With caps, you can skip these requirements and have the kernel do it for you. When control is handed to `main()`, your process already has the capabilities it needs so your service can use a non-root user and group (this is the preferred way for starting privileged services).

The caps section uses the following syntax:

Section	Value	Definition
`[path]`		The filesystem path to configure. A path ending in / is considered a dir, else it's a file. It is an error to specify multiple sections with the same `[path]` in different files. In Python versions <= 3.2, the same file may contain sections that override the previous section; in Python 3.2, it's set to strict mode.
`mode`	Octal file mode	A valid octal file mode of at least 3 digits. If 3 is specified, it is prefixed with a 0, else mode is used as is.
`user`	AID_<user>	Either the C `define` for a valid AID or the friendly name (e.g. both `AID_RADIO` and `radio` are acceptable). To define a custom AID, see Configuring the AID section (#configuring-the-aid-section).
`group`	AID_<group>	Same as user.
`caps`	cap*	The name as declared in `system/core/include/private/android_filesystem_capability.h` without the leading `CAP_`. Mixed case allowed. Caps can also be the raw: • binary (0b0101) • octal (0455) • int (42) • hex (0xFF) Separate multiple caps using whitespaces.

For a usage example, see Using file system capabilities (#using-file-system-capabilities).

Configuring the AID section

The AID section contains OEM-specific AIDs and uses the following syntax:

Section	Value	Definition
`[AID_<name>]`		The **`<name>`** can contain characters in the set uppercase, numbers, and underscores. The lowercase version is used as the friendly name. The generated header file for code inclusion uses the exact **`AID_<name>`**. It is an error to specify multiple sections with the same **`AID_<name>`** (case insensitive with the same constraints as **`[path]`**).
`value`	<number>	A valid C style number string (hex, octal, binary and decimal). It is an error to specify multiple sections with the same value option **or** to specify a value that is outside of the inclusive OEM ranges (defined in **`system/core/include/private/android_filesystem_config.h`**): • AID_OEM_RESERVED_START(2900) - AID_OEM_RESERVED_END(2999) • AID_OEM_RESERVED_2_START(5000) - AID_OEM_RESERVED_2_END(5999)

For usage examples, see Defining an OEM-specific AID (#defining-an-oem-specific-aid) and Using an OEM-specific AID (#using-an-oem-specific-aid).

Usage examples

The following examples detail how to define and use an OEM-specific AID and how to enable filesystem capabilities.

Defining an OEM-specific AID

To define an OEM-specific AID, create a **`config.fs`** file and set the AID value. For example, in **`device/x/y/config.fs`**, set the following:

```
[AID_FOO]
value: 2900
```

After creating the file, set the **`TARGET_FS_CONFIG_GEN`** variable and point to it in **`BoardConfig.mk`**. For example, in **`device/x/y/BoardConfig.mk`**, set the following:

```
TARGET_FS_CONFIG_GEN += device/x/y/config.fs
```

Your custom AID can now be consumed by the system at large on a new build.

Using an OEM-specific AID

To access the **#define** value of your AID via C or C++ code, use the autogenerated header file by adding to your module's **Android.mk** and including the empty faux library. For example, in **Android.mk**, add the following:

```
LOCAL_STATIC_LIBRARIES := liboemaids
```

In your C code, **#include "generated_oem_aid.h"** and start using the declared identifiers. For example, in **my_file.c**, add the following:

```
#include "generated_oem_aid.h"

…

If (ipc->uid == AID_FOO) {
  // Do something
...
```

In Android 8.0, you must continue to use **oem_####** with **getpwnam** and similar functions, as well in places that handle lookups via **getpwnam** (such as init scripts). For example, in **some/init.rc**, use the following:

```
service foo /vendor/bin/foo_service
    user: oem_2900
    group: oem_2900
```

Using file system capabilities

To enable filesystem capabilities, create a caps section in the **config.fs** file. For example, in **device/x/y/config.fs**, add the following section:

```
[system/bin/foo_service]
mode: 0555
user: AID_FOO
group: AID_SYSTEM
caps: SYS_ADMIN | SYS_NICE
```

Note: The nice names **foo** and **system** could be used here as well.

After creating the file, set the **TARGET_FS_CONFIG_GEN** to point to it in **BoardConfig.mk**. For example, in **device/x/y/BoardConfig.mk**, set the following:

```
TARGET_FS_CONFIG_GEN += device/x/y/config.fs
```

When service **foo** is executed, it starts with capabilities **CAP_SYS_ADMIN** and **CAP_SYS_NICE** without **setuid** and **setgid** calls. In addition, the **foo** service's SELinux policy no longer needs **setuid** and **setgid**, so these capabilities can be removed from the SELinux policy for **foo**.

Configuring overrides (Android 6.x-7.x)

Android 6.0 relocated **fs_config** and associated structure definitions (**system/core/include/private/android_filesystem_config.h**) to **system/core/libcutils/fs_config.c** where they could be updated or overridden by binary files installed in **/system/etc/fs_config_dirs** and **/system/etc/fs_config_files**. Using separate matching and parsing rules for directories and files (which could use additional glob expressions) enabled Android to handle directories and files in two different tables. Structure definitions in **system/core/libcutils/fs_config.c** not only allowed runtime reading of directories and files, but the host could use the same files during build time to construct filesystem images as **${OUT}/system/etc/fs_config_dirs** and **${OUT}/system/etc/fs_config_files**.

While the override method of extending the filesystem has been superseded by the modular config system introduced in Android 8.0, you can still use the old method if desired. The following sections detail how to generate and include override files and configure the filesystem.

Generating override files

You can generate the aligned binary files **/system/etc/fs_config_dirs** and **/system/etc/fs_config_files** using the **fs_config_generate** tool in **build/tools/fs_config**. The tool uses a **libcutils** library function (**fs_config_generate()**) to manage DAC requirements into a buffer and defines rules for an include file to institutionalize the DAC rules.

To use, create an include file in **device/*vendor*/*device*/android_filesystem_config.h** that acts as the override. The file must use the **structure fs_path_config** format defined in **system/core/include/private/android_filesystem_config.h** with the following structure initializations for directory and file symbols:

- For directories, use **android_device_dirs[]**.

- For files, use `android_device_files[]`.

When not using `android_device_dirs[]` and `android_device_files[]`, you can define `NO_ANDROID_FILESYSTEM_CONFIG_DEVICE_DIRS` and `NO_ANDROID_FILESYSTEM_CONFIG_DEVICE_FILES` (see the example (#older-example) below). You can also specify the override file using `TARGET_ANDROID_FILESYSTEM_CONFIG_H` in the board configuration, with an enforced basename of `android_filesystem_config.h`.

Including override files

To include files, ensure that `PRODUCT_PACKAGES` includes `fs_config_dirs` and/or `fs_config_files` so it can install them to `/system/etc/fs_config_dirs` and `/system/etc/fs_config_files`, respectively. The build system searches for custom `android_filesystem_config.h` in `$(TARGET_DEVICE_DIR)`, where `BoardConfig.mk` exists. If this file exists elsewhere, set board config variable `TARGET_ANDROID_FILESYSTEM_CONFIG_H` to point to that location.

Configuring the filesystem

To configure the filesystem in Android 6.0 and higher:

1. Create the `$(TARGET_DEVICE_DIR)/android_filesystem_config.h` file.
2. Add the `fs_config_dirs` and/or `fs_config_files` to `PRODUCT_PACKAGES` in the board configuration file (e.g., `$(TARGET_DEVICE_DIR)/device.mk`).

Override example

This example shows a patch for overriding the `system/bin/glgps` daemon to add wake lock support in the `device/`*`vendor`*`/`*`device`* directory. Keep the following in mind:

- Each structure entry is the mode, uid, gid, capabilities, and the name. `system/core/include/private/android_filesystem_config.h` is included automatically to provide the manifest #defines (`AID_ROOT`, `AID_SHELL`, `CAP_BLOCK_SUSPEND`).
- The `android_device_files[]` section includes an action to suppress access to `system/etc/fs_config_dirs` when unspecified, which serves as an additional DAC protection for lack of content for directory overrides. However, this is weak protection; if someone has control over `/system`, they can typically do anything they want.

```
diff --git a/android_filesystem_config.h b/android_filesystem_config.h
new file mode 100644
index 0000000..874195f
--- /dev/null
+++ b/android_filesystem_config.h
@@ -0,0 +1,36 @@
+/*
+ * Copyright (C) 2015 The Android Open Source Project
+ *
+ * Licensed under the Apache License, Version 2.0 (the "License");
+ * you may not use this file except in compliance with the License.
+ * You may obtain a copy of the License at
+ *
+ *      http://www.apache.org/licenses/LICENSE-2.0
+ *
+ * Unless required by applicable law or agreed to in writing, software
+ * distributed under the License is distributed on an "AS IS" BASIS,
+ * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or
+ * implied. See the License for the specific language governing
+ * permissions and limitations under the License.
+ */
+
+/* This file is used to define the properties of the filesystem
+** images generated by build tools (eg: mkbootfs) and
+** by the device side of adb.
+*/
+
+#define NO_ANDROID_FILESYSTEM_CONFIG_DEVICE_DIRS
+/* static const struct fs_path_config android_device_dirs[] = { }; */
+
+/* Rules for files.
+** These rules are applied based on "first match", so they
+** should start with the most specific path and work their
+** way up to the root. Prefixes ending in * denotes wildcard
+** and will allow partial matches.
+*/
+static const struct fs_path_config android_device_files[] = {
+  { 00755, AID_ROOT, AID_SHELL, (1ULL << CAP_BLOCK_SUSPEND),
"system/bin/glgps" },
+#ifdef NO_ANDROID_FILESYSTEM_CONFIG_DEVICE_DIRS
+  { 00000, AID_ROOT, AID_ROOT, 0, "system/etc/fs_config_dirs" },
+#endif
+};


diff --git a/device.mk b/device.mk
index 0c71d21..235c1a7 100644
--- a/device.mk
```

```
+++ b/device.mk
@@ -18,7 +18,8 @@ PRODUCT_PACKAGES := \
     libwpa_client \
     hostapd \
     wpa_supplicant \
-    wpa_supplicant.conf
+    wpa_supplicant.conf \
+    fs_config_files

 ifeq ($(TARGET_PREBUILT_KERNEL),)
 ifeq ($(USE_SVELTE_KERNEL), true)
```

Migrating filesystems from earlier releases

When migrating filesystems from Android 5.x and earlier, keep in mind that Android 6.x:

- Removes some includes, structures, and inline definitions.
- Requires a reference to **`libcutils`** instead of running directly from **`system/core/include/private/android_filesystem_config.h`**. Device manufacturer private executables that depend on **`system/code/include/private_filesystem_config.h`** for the file or directory structures or **`fs_config`** must add **`libcutils`** library dependencies.
- Requires device manufacturer private branch copies of the **`system/core/include/private/android_filesystem_config.h`** with extra content on existing targets to move to **`device/`*`vendor`*`/`*`device`*`/android_filesystem_config.h`**.
- As Android reserves the right to apply SELinux Mandatory Access Controls (MAC) to configuration files on the target system, implementations that include custom target executables using **`fs_config()`** must ensure access.

Last updated August 28, 2017.