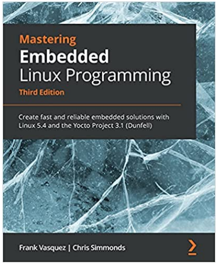
Why can’t my app open that file?

A deep dive into the Android app sandbox

Chris Simmonds

Droidcon London 2023

About Chris Simmonds



Mastering
Embedded
Linux Programming
Third Edition

Create fast and reliable embedded solutions with
Linux 5.4 and the Yocto Project 3.1 (Dunfell)

Frank Vasquez | Chris Simmonds

- Freelance consultant
- Author of *Mastering Embedded Linux Programming*
- Working with embedded Linux since 1999
- Android since 2009
- Speaker at many conferences and workshops
- Organiser of the AOSP and AAOS Meetup

"Looking after the Inner Penguin" blog at `https://2net.co.uk/`

Mastodon: @csimmonds@fosstodon.org `https://fosstodon.org/@csimmonds`

`https://uk.linkedin.com/in/chrisdsimmonds/`

Objectives of this talk

- Describe the Android application sandbox: why and how
- Give some insight into the internal workings of Android
- Show how platform apps can bypass (some of) these restrictions

What files can my app access?

Standard applications can access:

What files can my app access?

Standard applications can access:

- Private files in internal storage,

What files can my app access?

Standard applications can access:

- Private files in internal storage,
- Private files in external storage,

What files can my app access?

Standard applications can access:

- Private files in internal storage,
- Private files in external storage,
- Shared files in external storage,

What files can my app access?

Standard applications can access:

- Private files in internal storage,
- Private files in external storage,
- Shared files in external storage,
- ... and that's it

What files can my app access?

Standard applications can access:

- Private files in internal storage,
- Private files in external storage,
- Shared files in external storage,
- ... and that's it

Anything else will result in a FileNotFoundException

Good or bad?

Good, because

- improves system and user security: I can be sure that no other app can read my bank account details

Bad, because

- makes it harder to share data between apps
- prevents apps from accessing useful system files
    - a problem when designing dedicated embedded Android devices

The Android application sandbox

- The sandbox was part of the original design of Android, right from day 1
- Isolates applications from each other and from the operating system
  - each application can access its own memory and files, and no others(*)
- The sandbox uses Linux kernel features for
  - memory isolation, based on Linux processes virtual memory
  - file isolation, based on Linux User IDs (UID), Group IDs (GID), and file mode

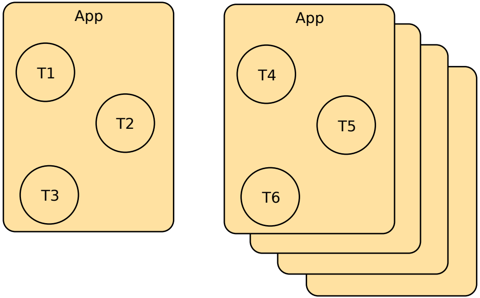
(*) in theory. There have been various loopholes, some of which get closed with each release

Linux processes

- A **Linux process** consists of
  - an area of virtual memory that contains the code, data, stack and heap
  - a Process Identifier (**PID**) that is allocated when the process is created
  - one or more threads, each identified by a Thread Identifier (**TID**)
  - an owner, indicated by a User Identifier (**UID**)
  - a group owner, indicated by a Group Identifier (**GID**)
  - zero or more supplementary GIDs
- The threads in a process share the address space, and so can share memory, but they cannot access any memory outside the process

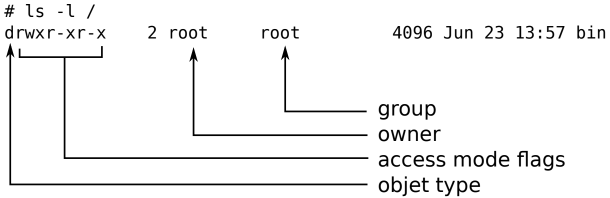
Sandbox: memory isolation

- Each Android app runs in a separate Linux process
  - therefore, threads in one app cannot read or write memory from another app



Linux file permissions and DAC

Each file has an owner (**UID**), a group (**GID**), and a set of permission flags, called the **mode**



```
# ls -l /
drwxr-xr-x    2 root     root         4096 Jun 23 13:57 bin
```

File mode flags

400	`r--------`	Owner permissions
200	`-w-------`	
100	`--x------`	
040	`---r-----`	Group permissions
020	`----w----`	
010	`-----x---`	
004	`------r--`	World permissions
002	`-------w-`	
001	`--------x`	

To read, write or execute a file, a process must match the mode flags

- if the UID of the process and file match, the first 3 flags ("owner")
- if the GID of the process and file match, the middle 3 flags ("group")
- otherwise, the last 3 flags ("world")

This mechanism is known as Discretionary Access Control, **DAC**

Sandbox: file isolation

- Android apps have Linux User IDs!
- Each app is assigned a unique Linux UID by Package Manager when installed
  - i.e. Android uses a Linux UID to identify an application (known as an appId in 14+)
  - App UIDs are in the range 10,000 to 99,999 (so, a maximum of 89,999 apps installed at once?); UIDs 0 to 9,999 are reserved for the system
- Each app has a place to put private files, e.g. `/data/data/<package name>`
- Linux DAC ensures that no other app can access those files

Note: since DAC is enforced by the kernel, NDK libraries have exactly the same restrictions as byte code

Digression: AOSP build types

For some of the examples, I am using a **userdebug** build so that I can use a root shell to show things that are not normally visible

The Android platform is built from the Android Open Source Project (AOSP)

AOSP allows three build types:

- **user**: locked-down, production build
- **userdebug**: includes `su` command for root-access, good for debugging
- **eng**: similar to userdebug, but root access is the default

$ prompt = normal shell

prompt = root shell

DAC in action

The user ID is recorded by Package Manager when the app was installed:

```
$ dumpsys package packages
[...]
  Package [com.example.filedemo] (878585b):
    appId=10089
[...]
```

At run-time the app has UID 10089

```
$ ps -An | grep filedemo
USER           PID  PPID        VSZ    RSS WCHAN            ADDR S NAME
10089         2592   330   13817352 130652 0                   0 S com.example.filedemo
```

The internal storage for the app has UID 10089 and GID 10089 for persistent files, and 20089 for cached files

```
# ls -ln /data/data/com.example.filedemo/
total 24
drwxrws--x 3 10089 20089 4096 2023-10-25 09:54 cache
drwxrws--x 2 10089 20089 4096 2023-10-24 19:49 code_cache
drwxrwx--x 2 10089 10089 4096 2023-10-25 09:54 files
```

supplementary groups

Find the PID of the app:

```
$ ps -A | grep filedemo
u0_a116       2071   357 13672104 117356 0                  0 S com.example.filedemo
```

Look at the supplementary groups:

```
emulator64_x86_64:/ $ grep Groups /proc/2071/status
Groups: 9997 20116 50116
```

Meaning:

```
9997     shared between all apps in the same profile
20116    cached data
50116    apps in each user to share
```

What about Android Users?

- Jelly Bean 4.2 introduced multi-user Android on tablets; later releases extended support to other devices including phones and cars
- With the multi-user UI enabled, each user identifies themselves when they authenticate with the device (PIN, fingerprint, ...)
- But Linux UIDs are used already as appId, so how are real users accommodated?

Android user ID?

- Android maps ranges of 100,000 Linux UIDs onto each Android user ID (AUID)
  - but note that AUID 1 to 9 are missing(*)
- UID = AUID*100000 + appId
- For example, the filedemo app running with AUID 10 and appId 10089:

```
$ ps -An | grep filedemo
USER           PID  PPID      VSZ    RSS WCHAN            ADDR S NAME
1010089       3194   354 13725488 102688 0                   0 S com.example.filedemo
```

Or, without the n option, ps shows PID symbolically as u0_a89

```
u0_a89        3194   354 13725488 102688 0                   0 S com.example.filedemo
```

(*) I don't know why

File isolation for multi-user

- We need to isolate different users of the same app from each other
- For each user (AUID) there is a separate private data storage area in `/data/user/`
- For example,

```
/data/user/0/com.example.filedemo/files/myfile.txt
/data/user/10/com.example.filedemo/files/myfile.txt
```

- `/data/user/0` is a link to `/data/data` for backwards compatibility

```
 # ls -ln /data/user/0/com.example.filedemo/files/myfile.txt
-rw-rw---- 1 10116 10116 13 2023-10-26 16:26 /data/user/0/com.example.filedemo/files/myfile.txt
# ls -ln /data/user/10/com.example.filedemo/files/myfile.txt
-rw-rw---- 1 1010116 1010116 13 2023-10-26 16:32 /data/user/10/com.example.filedemo/files/myfile.txt
```

SELinux enters the picture

- Basic DAC permissions leave some loopholes
- So, we need a layer of Mandatory Access Control (**MAC**)
- Linux supports several MAC implementations: Android uses **SELinux**
  - SELinux = Security Enhanced Linux, written by the NSA
  - deployed in full enforcing mode since Android 5
- Note that DAC and MAC work together: a process has to pass both layers of security before it can access a file or other resource

SELinux context

- SELinux contexts are of the form **user:role:type:sensitivity[:category]**
- Each process has an SELinux context, shown with `ps -Z`:

```
$ ps -AZ
LABEL                              USER       PID  PPID  S NAME
u:r:platform_app:s0:c512,c768      u0_a98     753   373  S com.android.systemui
u:r:priv_app:s0:c512,c768          u0_a91    1090   373  S com.android.launcher3
u:r:system_app:s0                  system    1890   373  S com.android.localtransport
u:r:untrusted_app_25:s0:c512,c768  u0_a86    2282   373  S com.android.deskclock
```

- Each file also has an SELinux context `ls -Z`:

```
# ls -Z1 /data/user/0/com.example.filedemo/
u:object_r:app_data_file:s0:c116,c256,c512,c768 cache
u:object_r:app_data_file:s0:c116,c256,c512,c768 code_cache
u:object_r:app_data_file:s0:c116,c256,c512,c768 files
```

SELinux policy

- (Most) Android apps belong to one of these SELinux types
  - `untrusted_app`: a regular app, including all user-installed apps
  - `platform_app`: a pre-installed app, signed with the platform keys
  - `system_app`: a pre-installed app with UID = 1000 (system)
  - `priv_app`: a pre-installed app which can be granted permissions with protection level `signature|privileged`

SELinux policy

- SELinux policy is coded in AOSP: you can't change it at runtime
- The policy for each type is in a type enforcement (.te) file

Here is an example of the policy for an untrusted_app in `untrusted_app.te`

```
# Some apps ship with shared libraries and binaries that they write out
# to their sandbox directory and then execute.
allow untrusted_app_all privapp_data_file:file { r_file_perms execute };
allow untrusted_app_all app_data_file:file     { r_file_perms execute };
```

SELinux policy for untrusted_app

- For `untrusted_app`, the policy depends on the targetSdkVersion:

Policy file	**targetSdkVersion range**
`untrusted_app.te`	34 and later
`untrusted_app_32.te`	from 32 to 33
`untrusted_app_30.te`	from 30 to 31
`untrusted_app_29.te`	29 only
`untrusted_app_27.te`	from 26 to 27
`untrusted_app_25.te`	25 and earlier

Changes over the years

- API level 19 and higher: app doesn't need to request any storage-related permissions to access app-specific directories within external storage. The files stored in these directories are removed when your app is uninstalled
- API level 28 or lowerr: your app can access the app-specific files that belong to other apps, provided that your app has the appropriate storage permissions
- API level 29 and higher: apps are given scoped access into external storage, or scoped storage, by default. When scoped storage is enabled, apps cannot access the app-specific directories that belong to other apps
- API level 30 and higher: apps cannot create their own app-specific directory on external storage

`https://developer.android.com/training/data-storage/app-specific`

Breaking the rules: platform apps

- You can break the rules if you are the platform developer
- Usecases
  - (mass market) platform developer/integrator for phone/TV/Automotive OEM
  - (specialized hardware) embedded Android devices - smart white boards, room access/booking systems, test and measurement, PoS, Advertising

Platform apps

- Platform key is the key used to sign `/system/framework/framework-res.apk`
- Any app signed with the same key becomes a **platform_app**
- Platform apps can use low-level platform APIs by adding `platform_apis: true`, and removing `sdk_version`

```
android_app {
[...]
    certificate: "platform",
    platform_apis: true,
}
```

System apps

- A system app is a platform app with UID system (1000)
- Just add `android:sharedUserId="android.uid.system"` to `AndroidManifest.xml`
- SELinux domain `system_app`
- Can access files and resources with UID and GID system

Privileged apps

- A privileged app is a platform app with the privileged flag set in `Android.bp`

```
privileged: true,
```

- SELinux domain is `priv_app`
- Privileged apps can be granted permissions with protection level `"signature|privileged"`

Persistent apps

- A platform app can be made a persistent app by adding `android:persistent="true"` to `AndroidManifest.xml`
- Persistent apps are started early: before the HOME activity is started, and way before `BOOT_COMPLETED`
- Persistent apps are restarted if they crash
- Examples: SystemUI, phone

The AOSP and AAOS Meetup

If you are interested in these low-level details of Android then you might be interested in this meetup group

On-line meetup up every 2 months - next is Wednesday 15th November

Sign up: `https://www.meetup.com/the-aosp-and-aaos-meetup/`

Details of past meetups, including slides and videos: `https://aospandaaos.github.io/`

Questions?

Slides: `https://2net.co.uk/slides/sandbox-csimmonds-droidcon-london-2023.pdf`

Mastodon: @csimmonds@fosstodon.org `https://fosstodon.org/@csimmonds`

`https://uk.linkedin.com/in/chrisdsimmonds/`