EMBEDDED & DISTRIBUTED SEARCH

Android Security: An Overview Of Application Sandbox

The Problem:

Define a `policy` to control how various `clients` can `access` different `resources`.

A solution:

1. Each `resource` has an `owner` and belongs to a `group`.

SHARE

Labels

Android Security Framework

an owner but can belongs to multiple `groups`.

3. Each `resource` has a `mode` stating the `access permissions` allowed for its `owner`, `group` members and others, respectively.

In the context of operating system, or Linux specifically, the `resources` can be files, sockets, etc; the `clients` are actually processes; and we have three `access permissions`:read, write and execute.

Yes, this is just Linux's UID/GID based access control model, and the rules are enforced by Linux kernel. What we will discuss in this article is how it works Android. By the end of the article, we should be able to answer following questions.

1. How does Android set up the owner, groups and mode of a resource?

set up the owner and groups of a process?
3. What does it mean for users and apps? For example, is it possible for app1 access app2's data? Will a normal app be able to access device node directly?

The discussion here is based on the latest android master (Android N) but we'll mention some history in the hope it helps your understanding..

Android Users and Groups ID

Before we jumping in and answering above questions, let first take a look how the user and group are represented in Android. Yes, with an ID, obliviously. Here lists all the users and groups IDs for the system, their meaning and designated ranges for different purposes.

```
#define AID_ROOT
#define AID_SYSTEM
#define AID_RADIO
#define AID_BLUETOOT
#define AID_GRAPHICS
#define AID_INPUT
#define AID_AUDIO
#define AID_CAMERA
#define AID_LOG
#define AID_COMPASS
#define AID_MOUNT
#define AID_WIFI
#define ...
#define AID_WEBVIEW_
/* The 3000 series a
#define AID_NET_BT_A
#define AID_NET_BT
#define AID_APP
#define AID_USER
```

Then, we will look at the first question - How and when to set up the owner, groups and mode of a resource? Roughly speaking, there are two categories. The first is to set it when the file system is created; the second is to set it during the system init.

File System Configuration

When creating the file systems, following utilized to set the mode, uid and guid of corresponding directories and files. Since M, OEM are allowed to override those rules with customized configuration.

```
static const struct
    { 00770, AID_SYS
    { 00500, AID_ROC
    { 00771, AID_SYS
    { 00771, AID_ROC
    { 00771, AID_SYS
    { 01771, AID_SYS
    { 00775, AID_MED
    { 00771, AID_SYS
    { 00755, AID_ROC
    { 00755, AID_ROC
    { 00755, AID_ROC
    { 00755, AID_ROC
    { 00755, AID_ROC
    { 00777, AID_ROC
    { 00755, AID_ROC
};

static const struct
    { 00555, AID_ROC
    { 00644, AID_MED
    { 00755, AID_ROC
    { 00755, AID_ROC
    { 00755, AID_ROC
    { 00750, AID_ROC
    { 00755, AID_ROC
    { 00750, AID_ROC
    { 00640, AID_ROC
    { 00644, AID_ROC
};
```

Init and init.rc

The second place to set mode/uid/gid of a particular file or directory is the `init.rc`s, which will be read by `init` process - the first user space program will be executed after kernel is ready.

The full description of the init.rc and the boot process is outside of the scope of this article. As far as what is relevant to the discussing here, it boils down to use `chown` and `chmod` to set the owner and mode for a particular file and directory.

```
on post fs-data
    # We chown/chmod
    chown system sys
    chmod 0771 /data
```

Uevend and

Node

One thing we are of particular interest are the UID and GID of device node. Since device nodes are the interface to the system hardware resources, a failure to enforce permission control on device node indicates a big security vulnerability.

`ueventd` is responsible for taking are of assigning the correct mode, UID and GID to the device node. It starts very early and will parse `uventd.*.rc` and set up the mode/uid/gid of corresponding device node. This is the third place you can tweak the mode, uid and guid for a file but it is specific for the device node.

```
ueventd.rc
/dev/alarm
/dev/rtc0
/dev/tty0
/dev/graphics/*
/dev/input/*
/dev/eac
/dev/cam
...
```

covered three places where you can set the mode/uid/gid for files and directories, that is 1) when you creating the file system, 2) when the system start running and 3) a special handling of the device nodes using `ueventd`.

Now, it is time to look at another part of the story - how the uid/gid are set for processes. First, we will check the system processes. And, normal app processes.

UID/GID of System Process

At the late stage of the init process, the core system services, such as servicemanager, vold and surfaceflinger, will be started. The UID and GID of the system process are specified in its corresponding `.rc` file.

forsurfaceflinger, it's configuration is in the `surfaceflinger.rc`. It might worth note that, before M, the system process and its settings are all put into a centralized file called `init.rc`.

```
service surfacefling
    class core
    user system
    group graphics d
    onrestart restar
    writepid /sys/fs
```

As you can see, each process is assigned a `user` and multiple `groups`. For example, surfaceflinger's UID is system and it belongs to three groups: graphics, drmrpc and readproc.

To show the USER ID of a process, use `ps`

```
myDevice # ps
system    427   1
```

To show the Group IDs of a process, we can check the process' related proc file.

```
myDevice # cat /proc
Name:   surfacefling
```

```
Pid:    427
PPid:   1
TracerPid:  0
Uid:    1000    1000
Gid:    1003    1003
FDSize: 256
Groups: 1026 3009
```

We can see that surfaceflinger belongs to 1003 and 1026 groups, which are graphics and drmrpc respectively. (*1003 is the gid, 1026 and 3009 are the supplementary group it belongs to. See the proc main page for detail)

UID/GID of Normal App Process

Normal app will be assigned AID above 10000, and the GUID will be the same as AID.

To show the UID, use `ps`:

```
u0_a46     5833  1096
```

To check the GID, check its `proc` file:

```
myDevice:/ # cat /pr
Name:   android.came
State:  S (sleeping)
Tgid:   5833
Pid:    5833
PPid:   1096
TracerPid:  0
Uid:    10046   1004
Gid:    10046   1004
FDSize: 128
Groups: 3003
```

Note that the GID is the same as UID, which are 10046. It is easy to find out how it is related to the name u0_a46.

You may have noticed that there is supplementary groups ID the camera2 process belongs, 3003 (i.e `AID_INET`). It is related with what permission this app has been granted.

Permiss

GUID for Apps

If an application requests certain permission and is granted, the corresponding group ID will be added to the process of the application. Part of the mapping between the permission and group id is shown as below:

```
<permission name="an
    <group gid="net_
</permission>
<permission name="an
    <group gid="medi
    <group gid="sdca
</permission>
<permission name="an
    <group gid="inet
</permission>
```

We will use above camera2 app as an example to show how the permission is related to the group it is assigned.

To show the permissions granted for camera2 application, use

```
shell dumpsys package
com.android.camera2

    install permission
    // other permissio
    android.permission
    gids=[3003]
```

As we can see, since camera2 app is granted INTERNET permission, which maps to the `inet` group, it has the supplementary groups 3003.

Apps With Special UID

One particular interest is to assign an app a special UID so it will be allowed to access resource that otherwise won't be able to access. By special UID, we usually mean the UID defined for system, i.e those belong to the range of 1000 to 1999. Can that be achieved?

Yes, we can do that by declaring android:sharedUserId="

the AndroidManifest.xml. In addition, the application also should be signed with the platform key by adding `LOCAL_CERTIFICATE := platform` in the Android.mk.

One example is the NFC app. Instead of having a normal u0_axx UID, Nfc app has the User ID nfc.

```
     nfc       4414
```

And that is AID_NFC, 1027.

```
  arche:/ # cat /proc/
  Name:   com.android.
  Tgid:   4414
  Pid:    4414
  PPid:   1096
  Uid:    1027    1027
  Gid:    1027    1027
  Groups: 3001 3002 30
```

And that means the nfc app can access following device node directly!

```
  myDevice:/ # ls -l /
  crw-rw---- 1 nfc nfc
```

Here is an example how that is achieved in nfc app.

```
    package="com.and
    android:sharedUs
```

Being able to access the device node directly means lots of trusts; that is the reason the application request system UID must also be signed with platform certification.

```
LOCAL_PACKAGE_NAME :
LOCAL_CERTIFICATE :=
```

Now, it is time to have some exercises.

Exercises

1. Can app1 can access app2's data?

Normally, they can't.

```
drwxr-x--x u0_a21
drwxr-x--x u0_a22
```

App's uid/gid are unique and the mode is set to "rw" only for the owner.

the data of app2.

But it can be done by sharing same uid and signed with same certification, as we discussed in Apps with Special UID.

---

2. Whether a process can access a certain device node?

It depends.

case 1 : same UID

```
root@myBoard:/ # ll
crw-rw-rw- system
```

`surfacflinger` can access `/dev/ion` because surfaceflinger's user is system and so is the /dev/ion.

A recap that the mode/uid/gid of `/dev/ion` is set in the `ueventd.device.rc`. `/dev/ion 0666 system media`

```
root@myBoard:/ # ll
crw-rw----   root
```

Despite that UID of video0 and the mediaserver are different (root and media respectively), but since they belongs to the same group (camera), and also the permission for group member is "rw", so `mediaserver` can read and write `/dev/video0`node.

UID and Binder call

So far, we limit our definition of resources to be files. However, it can be something else, such as the ability to trigger certain system action, or more general, to do a Binder call.

For each binder call, at the server side, you can get its calling PID and UID, which can be used determine whether the denied. This is the most basic but fundamental practice in Android to ensure the IPC security.

Summary

UID/GID based security control is a type of Discretionary Access Control(DAC). It is the fundamental part of Android's sandbox and security model to ensure the data and system security, so it's important to understand how it works.

Since Android 4.3, SELinux, as an implementation of Mandatory Access Control(MAC), has been utilized to overcome the limitation of DAC and to further improve the security of Android. We can talk it about it someday as well.

EDIT: Well, SELinux is discussed here.

FRAMEWORK

SHARE

Popular posts from this blog

Android Camera2 API Explained

Compared with the old camera A ...

SHARE

READ MORE

Java Collec

Framework Cheat Sheet

Java Collections Framework (J ...

SHARE
READ MORE

Android UI Internal : UI Composition

SurfaceFlinger

SurfaceFlinger is an Android sy ...

SHARE

READ MORE

Total Pageviews

Archive

Labels