Published in ProAndroidDev

Ashu Tyagi Follow

Sep 26, 2020 · 5 min read · Listen

Save

Binders In Android (part I)

Photo by Jonathan Kemper on Unsplash

Introduction

First of all, why do we even need to know about binder? Have you ever thought what prevents us from tricking the system into hiding the soft-input keyboard which is being used by some other application, or releasing a wake lock acquired by another application, or from hiding another application's windows from the screen? Basically, how do Android's core system services respond to requests made by third-party applications in a way that is secure?

The answer to all of these question is very simple: T***he Binder.***

In the Android platform, the binder is used for everything that happens across processes in the core platform. Now we have good enough reasons to know about the binder so let's dive into it.

Inter-Process Communication

Binder is an inter-process communication (IPC) mechanism. Before getting into detail about how Binder works, let's briefly review IPC.

Processes in Android have separate address spaces and a process cannot directly access another process's memory (this is called *process isolation*). This is usually a good thing, both for stability and security reasons: multiple processes modifying the same memory can be catastrophic, and you don't want a potentially rogue process that was started by another user to dump your email by accessing your mail client's memory. However, if a process wants to offer some useful service(s) to other processes, it needs to provide some mechanism that allows other processes to discover and interact with those services. That mechanism is referred to as *IPC*.

Binder

Because the standard IPC mechanisms weren't flexible or reliable enough, a new IPC mechanism called *Binder* was developed for Android. Binders are the cornerstone of Android's architecture. It abstracts the low-level details of IPC from the developer, allowing applications to easily talk to both the System Server and others' remote service components.

Binder Implementation

As mentioned earlier, a process cannot access another process's memory. However, the kernel has control over all processes and therefore can expose an interface that enables IPC. In Binder, this interface is the */dev/binder* device, which is implemented by the Binder kernel driver. The *Binder driver* is the central object of the framework, and all IPC calls go through it.

But how is data actually passed between processes? The Binder driver manages part of the address space of each process. The Binder driver-managed chunk of memory is read-only to the process, and all writing is performed by the kernel module. When a process sends a message to another process, the kernel allocates some space in the destination process's memory and copies the message data directly from the sending process. It then queues a short message to the receiving process telling it where the received message is. The recipient can then access that message directly (because it is in its own memory space).

Higher-level IPC abstractions in Android such as ***Intents*** (commands with associated data that are delivered to components across processes), ***Messengers*** (objects that enable message-based communication across processes), and ***ContentProviders*** (components that expose a cross-process data management interface) **are built on top of Binder.** Additionally, service interfaces that need to be exposed to other

processes can be defined using the *Android Interface Definition Language (AIDL)*, which enables clients to call remote services as if they were local Java objects.

Binder Tokens

An interesting property of Binder object is that each instance maintains **a unique identity across all processes in the system**, no matter how many process boundaries it crosses or where it goes. This facility is provided by the Binder kernel driver as explained above.

Each `Binder`'s object reference is assigned either:

1. A **virtual memory address** pointing to a Binder object in the *same* process, or
2. A **unique 32-bit handle** (as assigned by the Binder kernel driver) pointing to the `Binder`'s virtual memory address in a *different* process.

The Binder kernel driver maintains a mapping of local addresses to remote Binder handles (and vice versa) for each `Binder` it sees, and assigns each `Binder`'s object reference its appropriate value to ensure that equality will behave as expected even in remote processes. The Binder's unique object identity rules allow them to be used for a special purpose: as shared, security access tokens.

Binders follow capability-based security model in which programs are granted access to a particular resource by giving them an unforgeable *capability* that both:

1. References the target object and,

2. Encapsulates a set of access rights to it.

Because capabilities are unforgeable, the mere fact that a program possesses a capability is sufficient to give it access to the target resource; there is no need to maintain access control lists (ACLs) or similar structures associated with actual resources. So it establishes the fact that:

> Binders are globally unique, which means if you create one, nobody else can create one that appears equal to it.

For this reason, the application framework uses Binder tokens extensively in order to ensure secure interaction between cooperating processes: a client can create a `Binder` object to use as a token that can be shared with a server process, and the server can use it to validate the client's requests without there being any way for others to spoof it.

Let's see some code to understand better

```
1
2  import android.content.Context
3  import android.os.Bundle
4  import android.os.PowerManager
5  import androidx.appcompat.app.AppCompatActivity
6
7  class ExampleActivity : AppCompatActivity() {
8
9      private lateinit var wakeLock: PowerManager.WakeLock
10
11     companion object {
12         private const val TIMEOUT_DURATION = 6000L
```



```
18         wakeLock = powerManager.newWakeLock(PowerManager.PARTIAL_WAKE_LOCK, "MyAp
19         wakeLock.acquire(TIMEOUT_DURATION)
20     }
21
22     override fun onDestroy() {
23         super.onDestroy()
24         wakeLock.release()
```

> *Looking at the PowerManager source code helps us understand what's happening under the hood (note that some of the source code has been cleaned up for the sake of brevity):*

```
@SystemService(Context.POWER_SERVICE)
public final class PowerManager {

    final IPowerManager mService;

    public final class WakeLock {
        @UnsupportedAppUsage
        private int mFlags;
        @UnsupportedAppUsage
        private String mTag;
        private final
        private final
        private final String mTraceName;

        WakeLock(int flags, String tag, String packageName) {
            // Create a token that uniquely identifies this wake lock.
            mToken = new Binder();
            mFlags = flags;
            mTag = tag;
            mPackageName = packageName;
            mTraceName = "WakeLock (" + mTag + ")";
        }

        public void acquire() {
            // Send the power manager service a request to acquire a wake
            // lock for the application. Include the token as part of the
            // request so that the power manager service can validate the
            // application's identity when it requests to release the wake
            // lock later on.
            synchronized (mToken) {
                try {
                    mService.acquireWakeLock(mToken, mFlags, mTag);
                } catch (RemoteException e) {
                    throw e.rethrowFromSystemServer();
                }
            }
        }

        public void release(int flags) {
```

```
40              // Send the power manager service a request to release the
41              // wake lock associated with 'mToken'.
42              synchronized (mToken) {
43                  try {
44                      mService.releaseWakeLock(mToken, flags);
45                  } catch (RemoteException e) {
46                      throw e.rethrowFromSystemServer();
47                  }
48              }
```

2. The client application creates and acquires a wake lock in `onCreate()`. The `PowerManager` sends the `WakeLock`'s unique `Binder` token as part of the `acquire()` request. When the `PowerManagerService` receives the request, it holds onto the token for safe-keeping and forces the device to remain awake.

3. The client application releases the wake lock in `onDestroy()`. The `PowerManager` sends the `WakeLock`'s unique `Binder` token as part of the request. When the `PowerManagerService` receives the request, it compares the token against all other `WakeLock` tokens it has stored, and only releases the `WakeLock` if it finds a match. This additional "validation step" is an important security measure put in place to guarantee that other applications cannot trick the `PowerManagerService` into releasing a `WakeLock` held by a different application.

Wrapping Up

Binder tokens are the driving force behind interaction in the framework. Although their existence for the most part is hidden from developers, Binder tokens are used extensively in the system for security.

Two other notable Binder features are reference counting and death notification (also known as link to death). Reference counting

guarantees that Binder objects are automatically freed when no one references them and is implemented in the kernel driver. We'll cover this in part 2 of this article.

References:

- https://www.androiddesignpatterns.com/2013/07/binders-window-tokens.html
- https://g.co/kgs/6Pf71w

Android | Android App Development | AndroidDev | Linux | Technology