# An In-Depth Introduction to the Android Permission Model

## and How to Secure Multi-Component Applications

**Jeff Six**
***jeffsix.com***

**OWASP**

3 April 2012

Presented at AppSecDC 2012

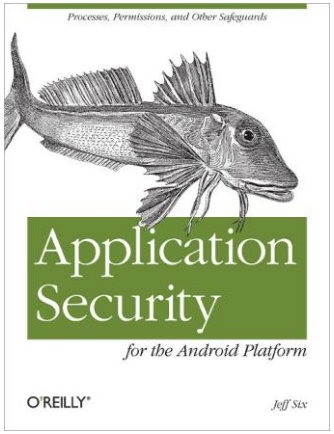


**The OWASP Foundation**

http://www.owasp.org

# About Jeff

- Jeff Six (http://jeffsix.com)
- MEE and BCpE, University of Delaware
- Eight+ years working computer and network security for NSA/DoD; five+ years for major financial institutions
- Author of *Application Security for the Android Platform*
  *(see me for discount coupons after the talk!)*
- Interests (Professional)
  - Computer, network, and application security
  - Android software development
  - STEM education
- Interests (Other)
  - Triathlon, SCUBA diving, other outdoor sports
  - Lifeguarding (19yr), LGI (16yr), EMS
- Family
  - 1 wife, 1 cat



# Introduction

- This talk introduces the application permissions system for the Android operating system/environment.
- It does not address the overall security model of Android, which is largely based on the underlying Linux kernel. Just know that each application (normally) runs under its own UID on an Android device and this is the fundamental construct for application separation.
- Application permissions allow components to control which other applications are allowed to interact with them.

# Android IPC Calls Using Intents

- Interprocess communication between applications is accomplished using the Binder interface and *Intents*.
- One application can create an Intent and send it to all apps running that are registered to receive that particular type of Intent (this is an *implicit Intent*). Intents can also be created that specify exactly which app should receive them (this is an *explicit Intent*).
- Each Android component can be either public or private. If it is public, other components can interact with it. If it is private, the only components that can interact with it are those that are part of the same app (or one that runs with the same UID).
  - Each component type is either public or private by default, depending on how it is used (if it specifies a filter to receive implicit Intents, then it is public…otherwise, it is private).


# Android Permissions Model

- The permissions model is based on permissions, which are constructs that various APIs require calling apps to have before they will provide certain services.
- Applications must declare (in their manifest) which permissions they request/require. When an application is installed, the Android system will present this list to the user and the user must decide to allow the installation or not.
  - This is an all-or-nothing decision; the user can install the app or not, but cannot choose to install it with reduced permissions.
  - This imparts a significant responsibility to both the developer (to accurately specify required permissions) and the user (to understand the risk involved and make an informed decision).

# Android Permissions Model

- The Android permissions model was designed with two fundamental goals.
  - **Inform the User** – By listing all "sensitive" operations that an application could perform, the user is more aware of the risks involved in installing the application. This assumes that the user will actually read the permission dialog and make a rationale yes/no install decision based on that information, which may or may not be true.
  - **Mitigate Exploits** – By limiting application access to sensitive APIs, the ability of an attacker to cause damage if an application is successfully exploited is somewhat mitigated (Dan Geer's "ding" versus "shatter").
- Most system permissions are all-or-nothing, by design. For example, an app gets unlimited Internet access or none at all.

# Specifying Required Permissions

- Making use of a system API that requires a permission simply requires you to specify that permission in your application's manifest (AndroidManifest.xml).
- For example, if you application needs access to the Internet, specify the INTERNET permission…

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
  package="com.example.testapps.test1">
        …
        <uses-permission android:name="android.permission.INTERNET" />
        …
</manifest>
```

# Permission Levels

- Android permissions fall into four levels.
  - **Normal** – These permissions cannot impart real harm to the user (e.g. change the wallpaper) and, while apps need to request them, they are automatically granted.
  - **Dangerous** – These can impart real harm (e.g. call numbers, open Internet connections, etc) and apps need to request them with user confirmation.
  - **Signature** – These are automatically granted to a requesting app if that app is signed by the same certificate (so, developed by the same entity) as that which declared/created the permission. This level is designed to allow apps that are part of a suite, or otherwise related, to share data.
  - **Signature/System** – Same as Signature, except that the system image gets the permissions automatically as well. This is designed for use by device manufacturers only.


# Custom Permissions

- Custom permissions can be used by developers to restrict access to various services/components.
- Any application that interacts with another application's component would need to possess the required permission for the call to succeed.
- First, a permission must be declared/created in an application's manifest.

```
<permission android:name="com.example.perm.READ_INCOMING_MSG"
  android:label="Read incoming messages from the EX service."
  android:description="Allows the app to access any messages received by
      the EX service. Any app granted this permission will be able to read all
      messages processed by the ex1 application."
  android.protectionLevel="dangerous"
  android:permissionGroup="android.permission-group.PERSONAL_INFO"
/>
```

# Enforcing Permissions Programmatically

- Applications can check to see if calling apps have permissions programmatically.

```
int canProcess = checkCallingPermission(
        "com.example.perm.READ_INCOMING_MSG");
if (canProcess != PERMISSION_GRANTED)
        throw new SecurityException();
```

- This will ensure that the calling process (via IPC) has the necessary permission. There are other forms of this call that can check to see if a specific PID/UID combination has a certain permission or if a specific package has that permission.

# Enforcing Permissions Programmatically

- One caveat…there is a method that will check to see if a calling process OR the currently active process has a specific permission.

```
int canProcess = checkCallingOrSelfPermission(
            “com.example.perm.READ_INCOMING_MSG”);
if (canProcess != PERMISSION_GRANTED)
            throw new SecurityException();
```

- This is a dangerous model, **and should not be used**, as it can lead to *permission leaking*. It can allow a process without a certain permission to call your code, which does have that permission, and trick it into performing an unauthorized operation.

# Enforcing Permissions via Manifest

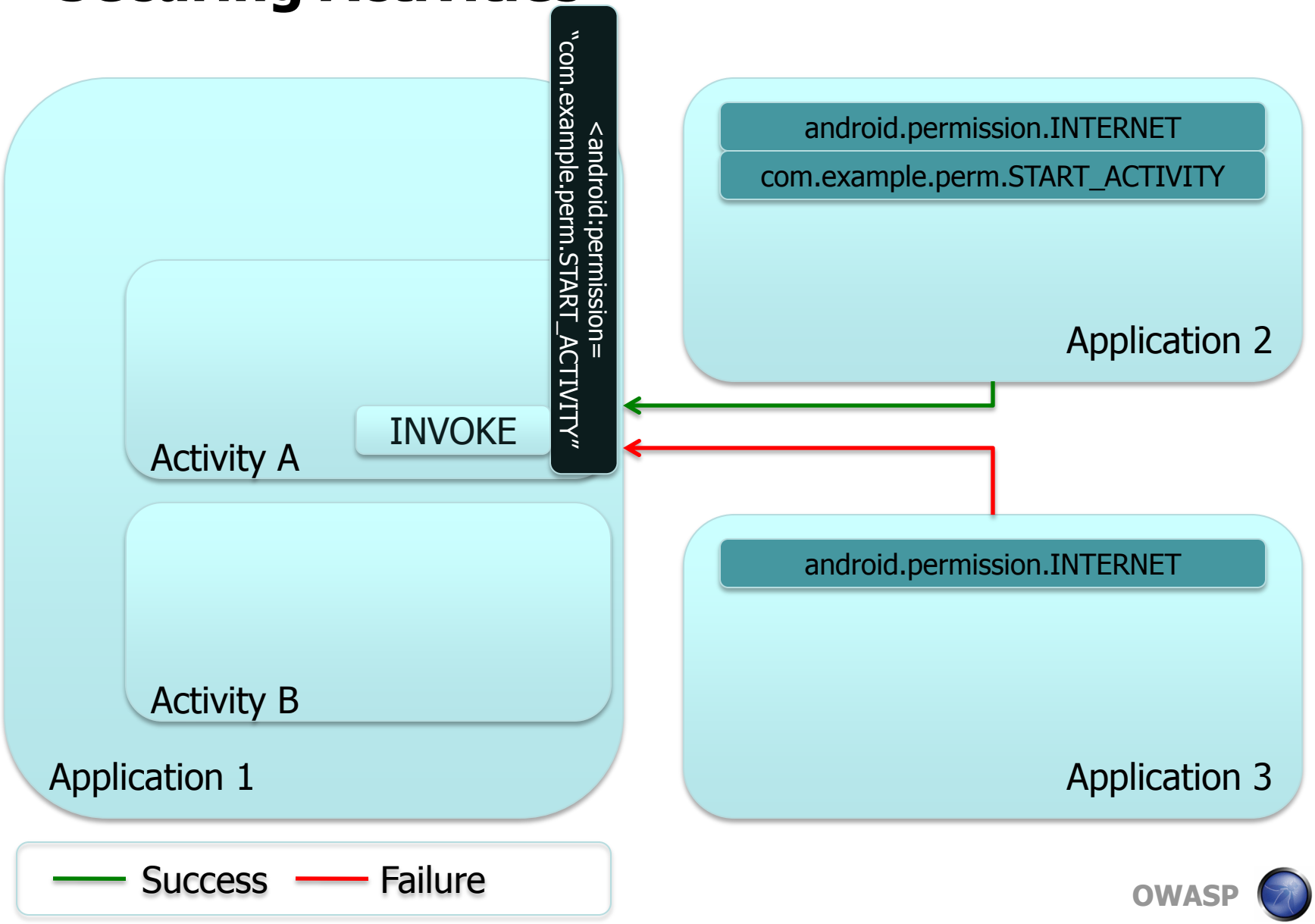
- It is also possible to enforce permissions to interact with certain components by specifying those permissions within the containing application's manifest.
- This method is generally preferred over programmatic checks, as permissions are more easily managed by including them all in a central place and decoupled from the application's source code.
- Each component type can specify required permissions using straightforward attributes in their manifest entries.

# Securing Activities

- An *Activity* is a component that represents a presentation layer experience for an Android application.
- You can use permissions to restrict what apps can cause an Activity to start by including a *permission* attribute in the manifest entry for the Activity in question.

```
<activity  android:name=“.Activity1”
                    android.permission=“com.example.perm.START_ACTIVITY”>
        <intent-filter>
                    …
        </intent-filter>
</activity>
```

# Securing Activities



# Securing Services

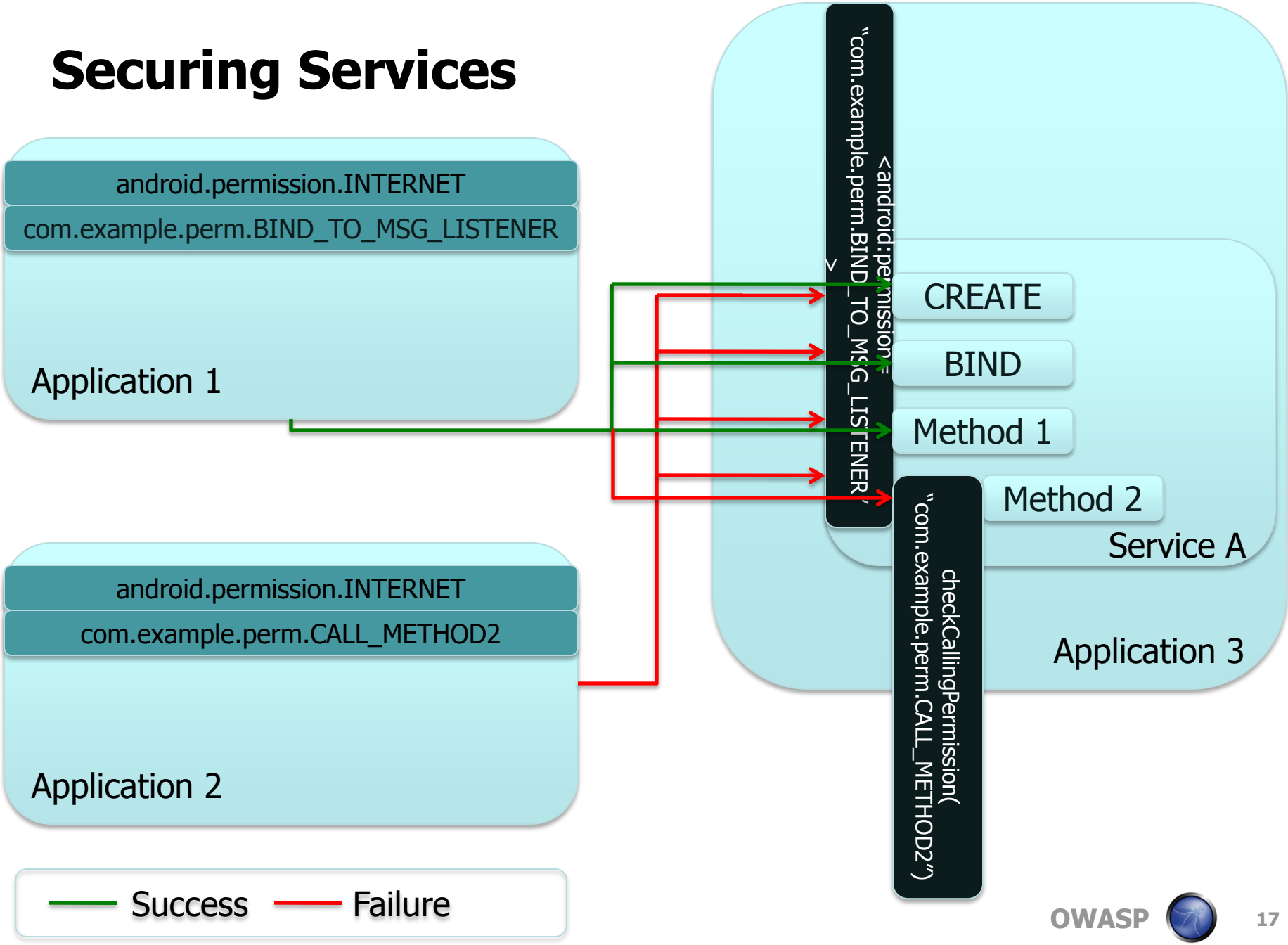
- A *Service* is a component that runs in the background and provides some services to other components.
- You can use permissions to restrict what apps can interact with a Service (creating an instance of it, binding to it, or calling methods on it) by including a *permission* attribute in the manifest entry for the Service in question.

```
<service android:name=“.MailListenerService”
        android:permission=“com.example.perm.BIND_TO_MSG_LISTENER”
        android:enabled=“true”
        android:exported=“true”
        <intent-filter></intent-filter>
</service>
```

# Securing Services

- The permission specified in a Service's manifest entry is required to create, bind to, and call methods on, the Service. This general case does not allow you to specify different permissions for specific callable methods of a bound Service.
  - You can accomplish this (requiring different permissions for different callable Service methods) by inserting programmatic checkCallingPermission() calls in the various methods.
  - This is an example of where programmatic checks may be necessary and why the ability to call checkCallingPermission() at arbitrary points within an application is important.
  - Note that such checks would be in addition to the check performed for the permission specified in the manifest entry.

# Securing Services

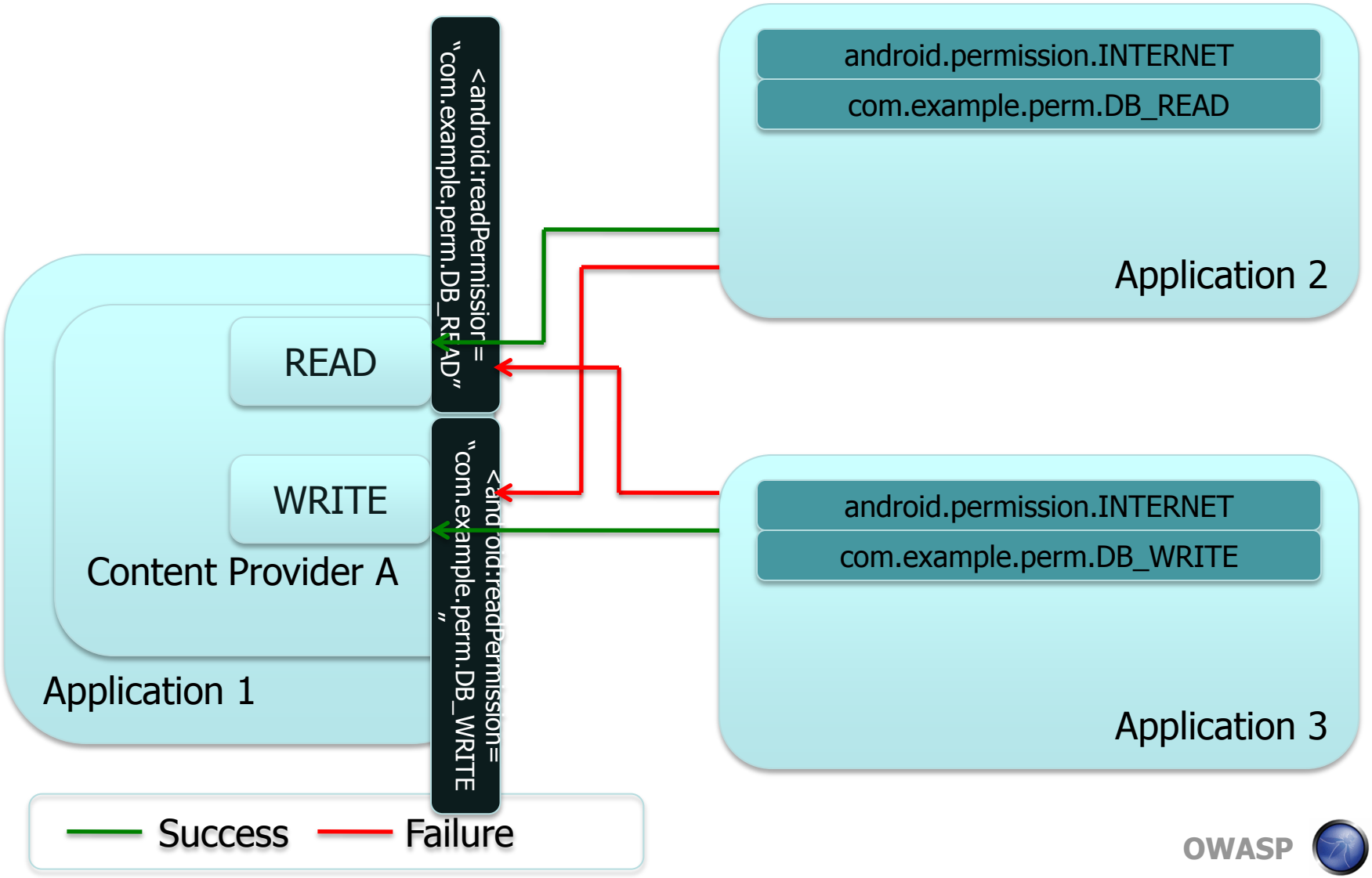


# Securing Content Providers

- A *Content Provider* is the standard way for Android apps to make data available to other apps, and is normally used as the front-end for a database or similar store.
  - Content Providers are not accessed via Intent-based IPC, but via connections to URIs that point into a Content Provider namespace.
- Content Providers can specify permissions required to either read or write to them (write does not imply read).

```
<provider android:name="com.example.test.app1.MailProvider"
        android:authorities="com.example.test.app1.mailprovider"
        android:readPermission="com.example.perm.DB_READ"
        android:writePermission="com.example.perm.DB_WRITE">
</provider>
```

# Securing Content Providers



# Securing Content Providers

- This basic level of permissions would require read/write permissions to the entire Content Provider to be granted. As Content Providers typically expose access to an underlying database, this is not ideal. *URI Permissions* are also available, which allows permissions to be granted to specific URIs within the provider.
- To use this, a Content Provider must be configured to allow it. There are two ways of doing this (either the entire Provider at once or specific subbranches).
- Note that this configuration only *allows* permissions for other apps to access this content to be granted. The permissions are not automatically granted and the app must take this second, programmatic, step to actually share the data.

# Securing Content Providers

- One can either enable the granting of URI-based permissions throughout the entire Provider…

```
<provider android:name="com.example.test.app1.MailProvider"
        android:authorities="com.example.test.app1.mailprovider"
        android:readPermission="com.example.perm.DB_READ"
        android:writePermission="com.example.perm.DB_WRITE"
        android:grantUriPermission="true">
</provider>
```

- Or to specific subbranches within the Provider…

```
<provider android:name="com.example.test.app1.MailProvider"
        android:authorities="com.example.test.app1.mailprovider"
        android:readPermission="com.example.perm.DB_READ"
        android:writePermission="com.example.perm.DB_WRITE">
            <grant-uri-permission android:path="/attachments/">
</provider>
```

# Securing Content Providers

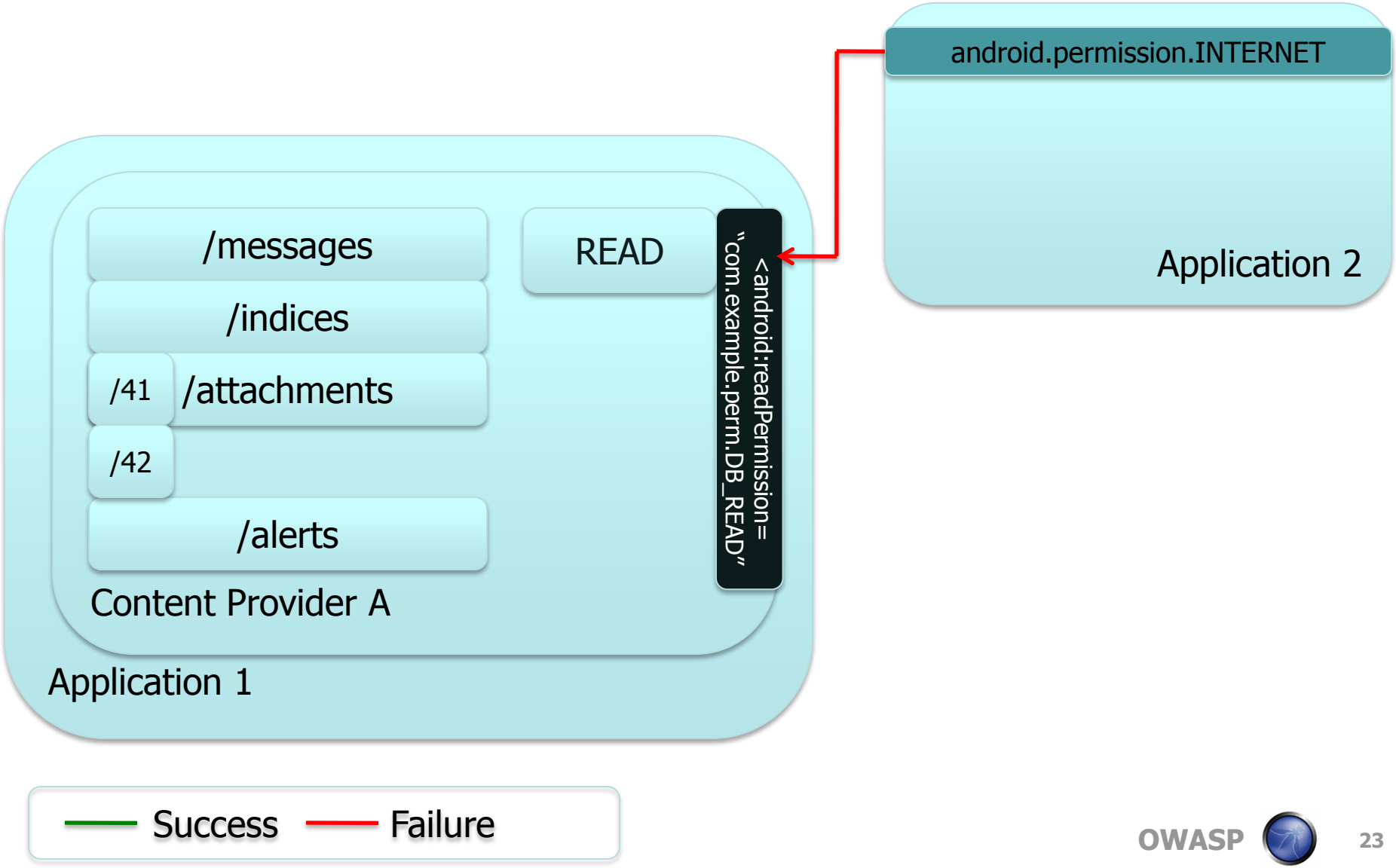
- To grant permissions to whatever application handles an implicit Intent, add that permission to the Intent.

```
uri = “content://com.example.test.provider1/attachments/42”;
Intent intent = new Intent(Intent.ACTION_VIEW);
intent.setFlags(Intent.FLAG_ACTIVITY_NEW_TASK);
intent.addFlags(Intent.FLAG_GRANT_READ_URI_PERMISSION);
intent.setDataAndType(uri, “image/gif”);
startActivity(intent);
```

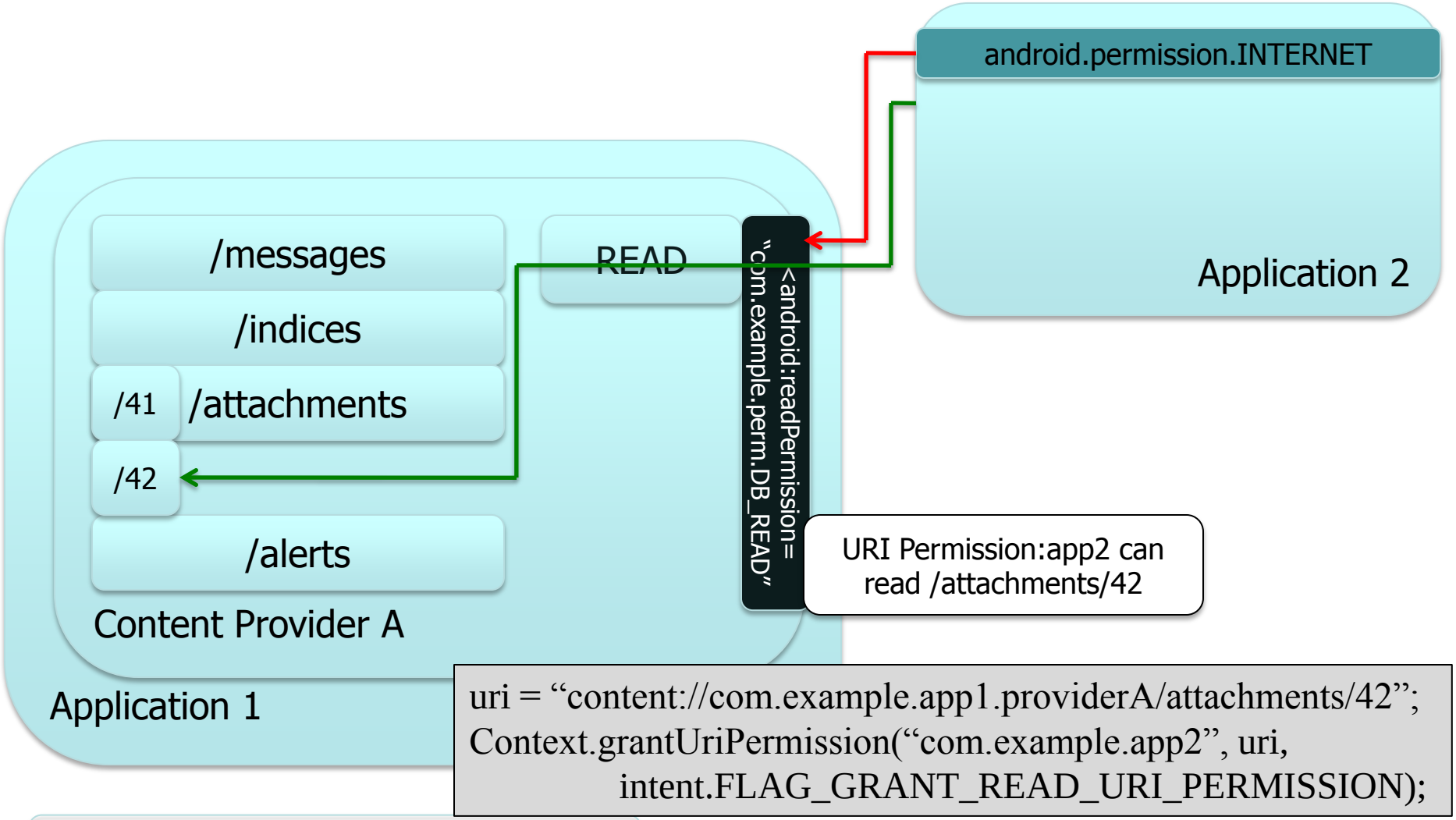
- To grant permissions to a specific application on the device, add the permission to that application directly.

```
uri = “content://com.example.test.provider1/attachments/42”;
Context.grantUriPermission(“com.example.test.app2”, uri,
        intent.FLAG_GRANT_READ_URI_PERMISSION);
```

# Securing Content Providers



# Securing Content Providers



# Securing Broadcast Intents

- *Broadcast Intents (Implicit Intents)* are messages sent out, across the system, to all apps that would like to receive them.
- A developer can restrict which apps are allowed to receive a broadcast by requiring a permission to do so…

```
Intent bcastIntent = new Intent(MESSAGE_RECEIVED);
context.sendBroadcast(bcastIntent, “com.example.perm.MSG_NOTIFY_RCV”);
```

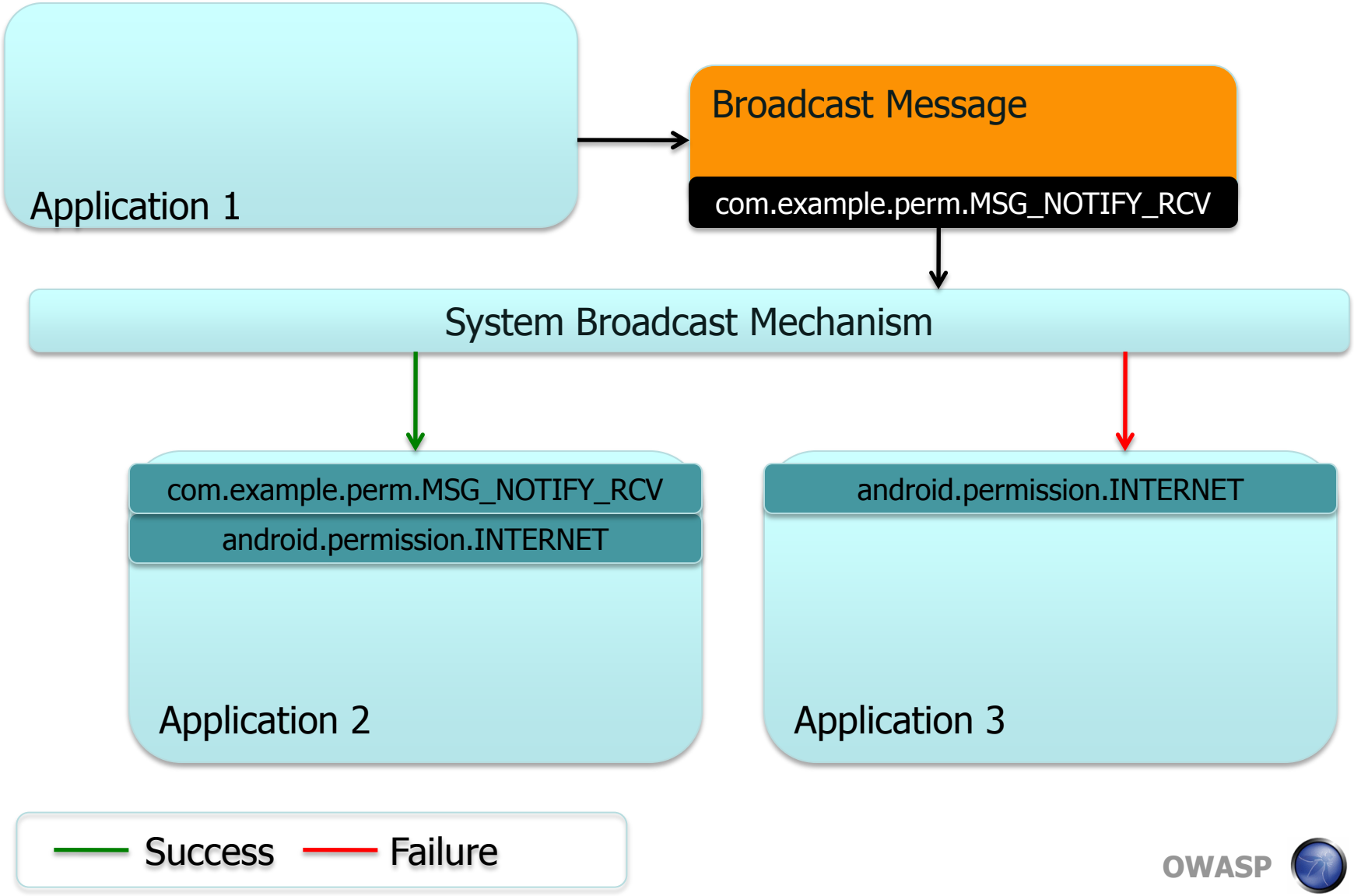
- One can also configure a broadcast receiver to only receive broadcasts from apps that have a permission…

```
IntentFilter intentFilter = new IntentFilter(MESSAGE_RECEIVED);
UIMailBroadcastReceiver rcv = new UIMailBroadcastReceiver();
context.registerReceiver(rcv, intentFilter,
        “com.example.perm.MSG_NOTIFY_SEND”, null);
```

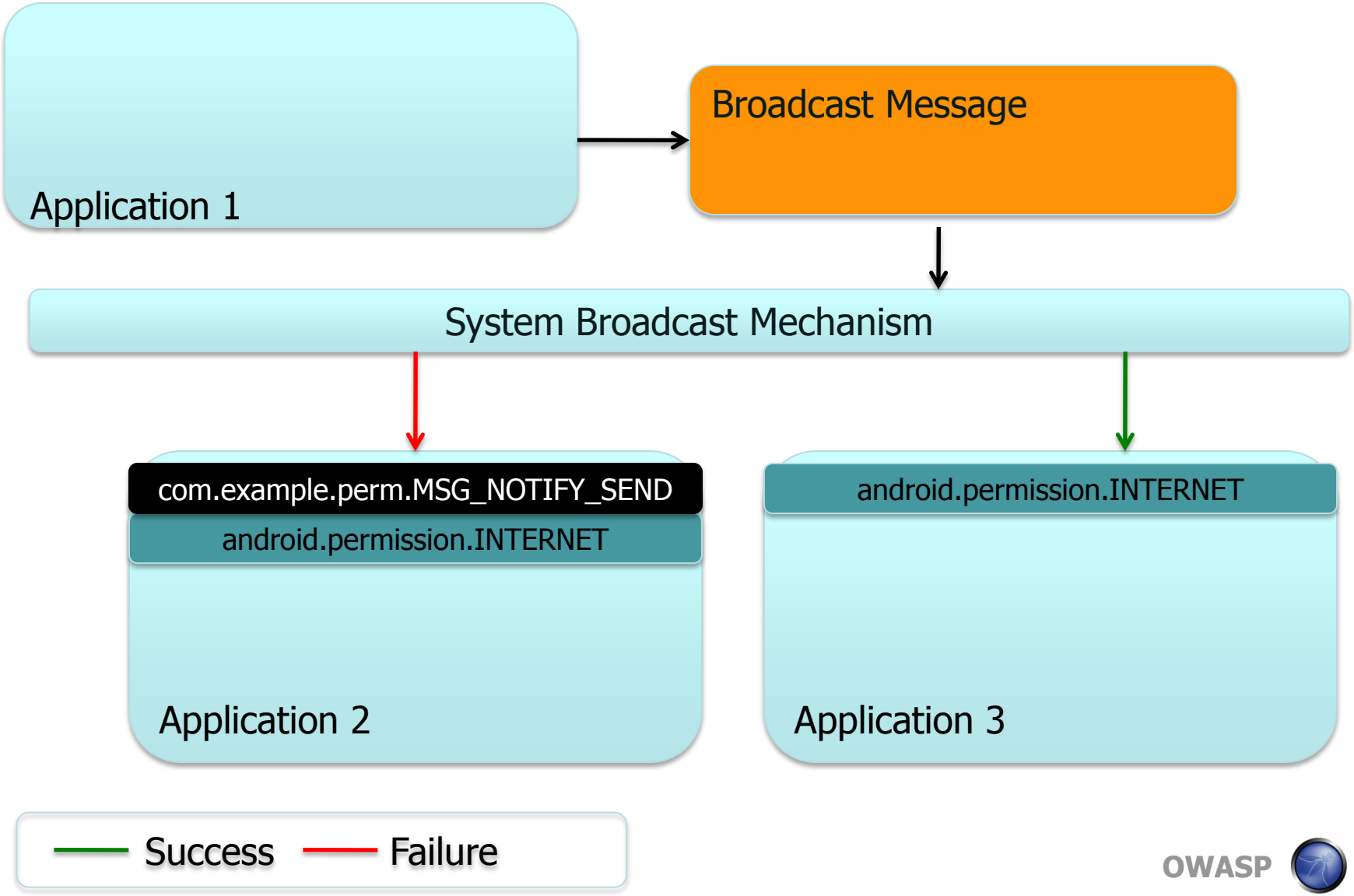
# Securing Broadcast Intents

- The two-sided nature of Broadcast Intents, broadcasting them out and listening for them, allow developers to properly secure both ends.
  - By requiring a permission to receive specific Broadcast Intents, an originator can specify which applications are allowed to be notified when specific messages are broadcast.
  - By requiring a permission from Broadcast Intent senders, applications that are listening for broadcasts can choose to only accept such messages from specific applications.

# Securing Broadcast Intents



# Securing Broadcast Intents



# And That's It…Thanks For Attending!

Questions? Comments?