WILLIAM & MARY

CHARTERED 1693

CSCI 445:
Mobile Application Security

Lecture 18

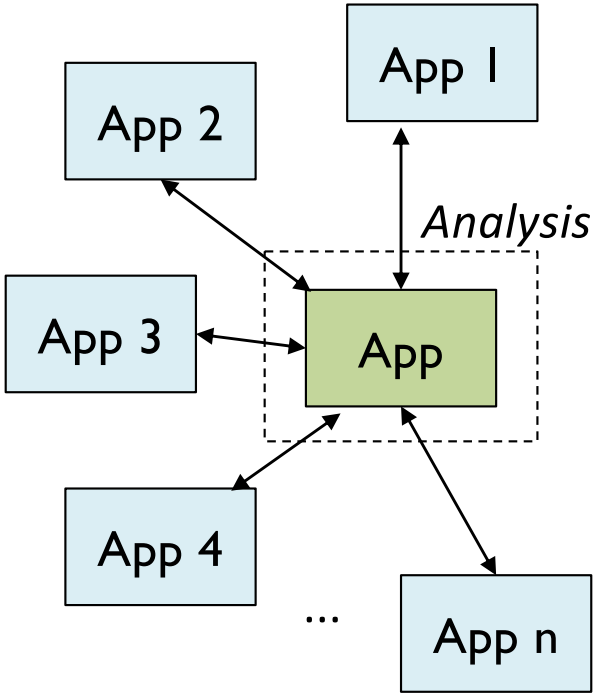
Prof. Adwait Nadkarni

Announcements

- Security Analysis *Workshop/Hackathon*!
  - Next Thursday, in class
- *Winner takes all: 3pt + on the class grade* (> half a step up)

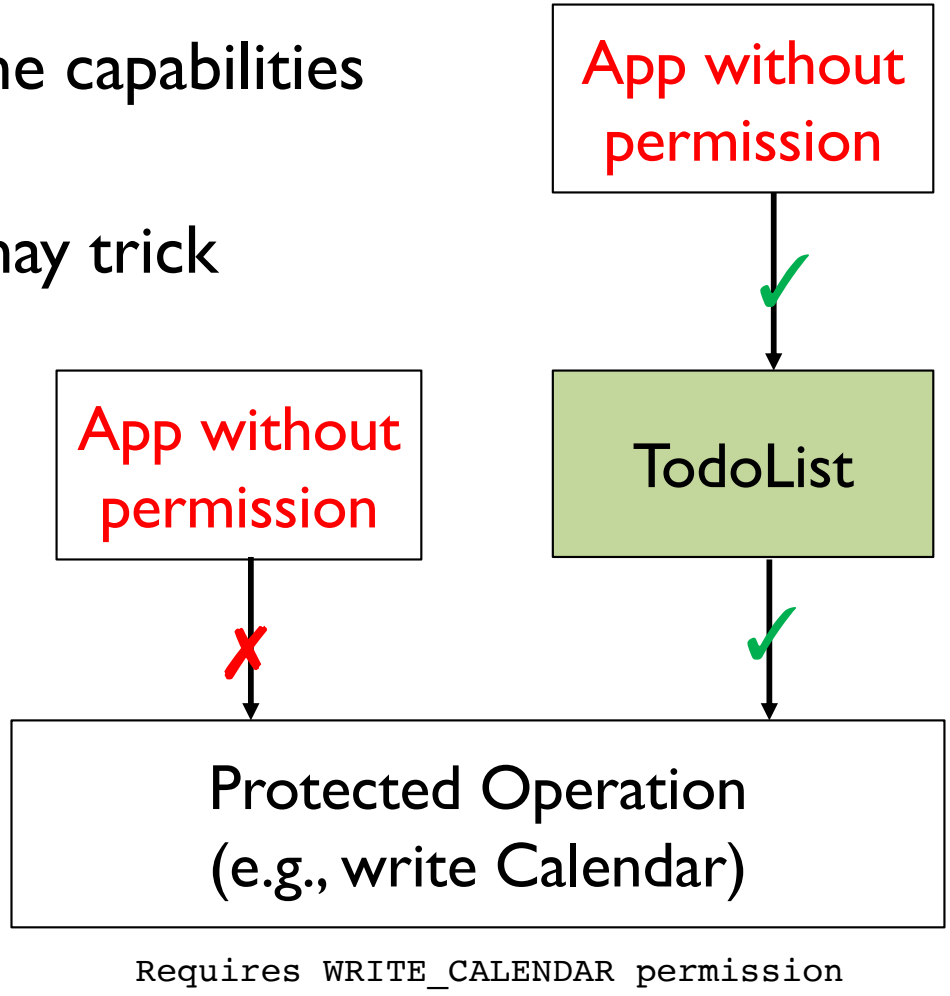
Is permission analysis enough?

- Analyzing the permissions of *one* app
  - Does the app *need* the permissions requested?
  - Does the app request a *high-risk* permission?
    - Or permission combinations?
- *What are we missing?*
  - Multiple untrusted apps
  - Apps communicate!



Inter-app communication problems

- *Collusion:* Two apps may combine capabilities (e.g., location + Internet)
- *Confused Deputy:* An attacker may trick vulnerable apps



Analyzing Permission Re-delegation

Permission Re-delegation

Permission re-delegation occurs when an application with permission to access a resource makes a call on behalf of another application, which does not have that permission.

- A general case of the *collusion* and *confused deputy* problems.
- The permission *delegated* by the user to the privileged app (i.e., the *deputy*), is granted (i.e., *re-delegated*) to the adversary, *without the user's consent*
- Also called a '*capability leak*'

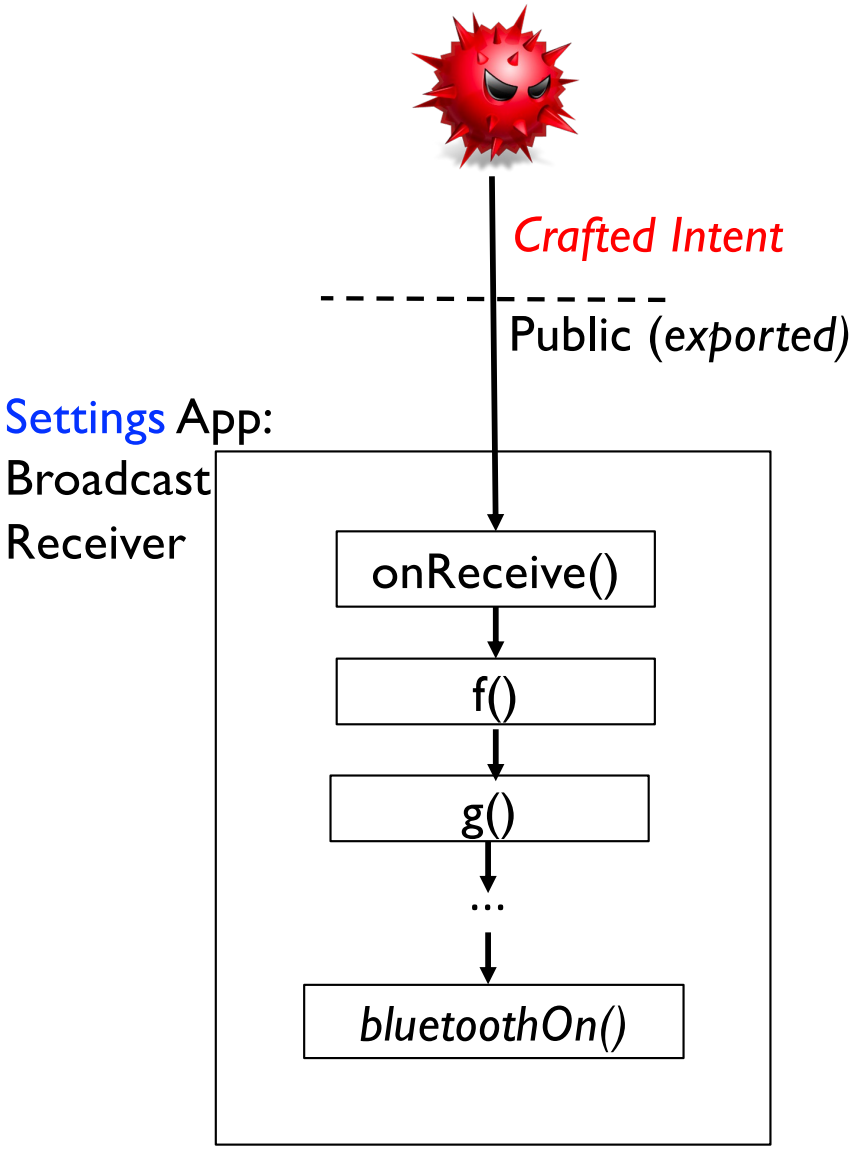
Detecting permission re-delegation/ capability leaks

- Goal: *Analyze apps to identify potential confused deputies*
- What to look at? *Class Exercise!*
  1. Permissions requested
  2. *Public* components:
     i. Activities
     ii. Services
     iii. Receivers
     iv. Providers
     - *Why prioritize these?*
       - *Background components*
  3. *What can you do with the access, i.e., the impact of the capability leak?*

Task: Analyze 1000+ apps for capability leaks

Let's define a practical approach!

1. Prioritize apps based on privilege
   - Apps with signature/system permissions (e.g., OEM apps)
   - Apps with certain *more* dangerous permissions
     - Are all dangerous permissions equal?
2. Identify public components
3. Find an execution path that uses the permission → Call graph!



- *What API to watch for?*
  - Permission Maps! (*for your analyses, even if you find this, along with step 1 and 2, it counts!*)

Need for a more *precise* approach

- Is the prior *basic* approach prone to FPs? Yes.
- **Protections in the Manifest:** Exported components may be *permission protected,* i.e., even if exported="true"
- **Authorization checks**: Developers may perform security checks in code; FPs
  - Rule A: If *any* check exists, mark as *negative*. Problem?
    - FNs
  - Rule B: Check for *specific permissions:* lower *FPs and FNs*

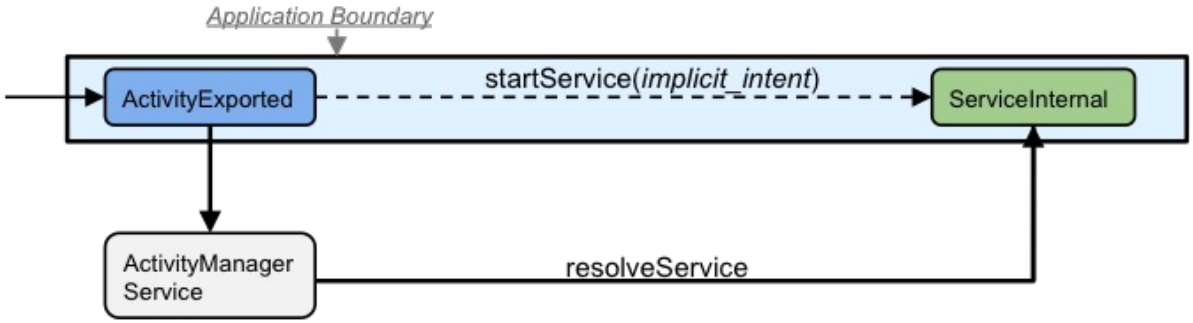
Challenges/Limitations

- *Scope:* Analysis only works for *Android* permissions
  - Detecting app-specific capability leaks is difficult. E.g., making Dropbox write files to public storage.
- *False Positives:* Access control checks in apps may not always be obvious
  - E.g., Apps may check for permissions, UIDs, PIDs, or some specific package/component name.
- *False Negatives:* App's authorization checks may look okay; but can't rule out false negatives without in-depth analysis

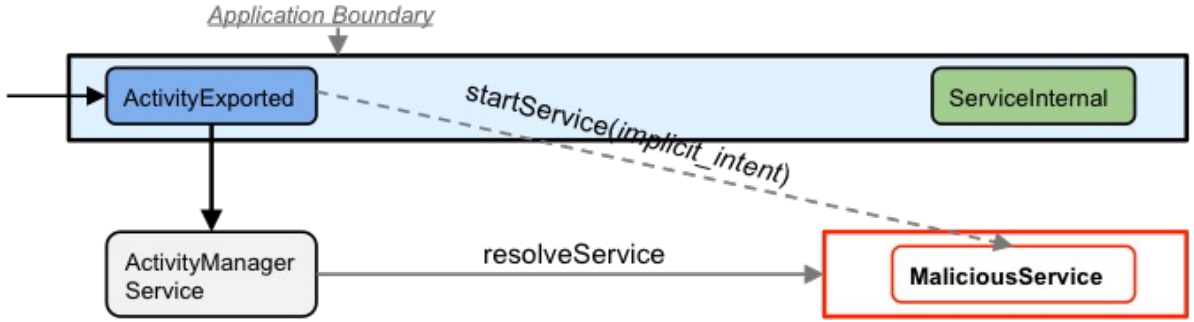
Analyzing Inter-app communication

Intent Hijacking

- Recall: an *implicit intent* is an intent message where Android's ActivityManager selects the target.

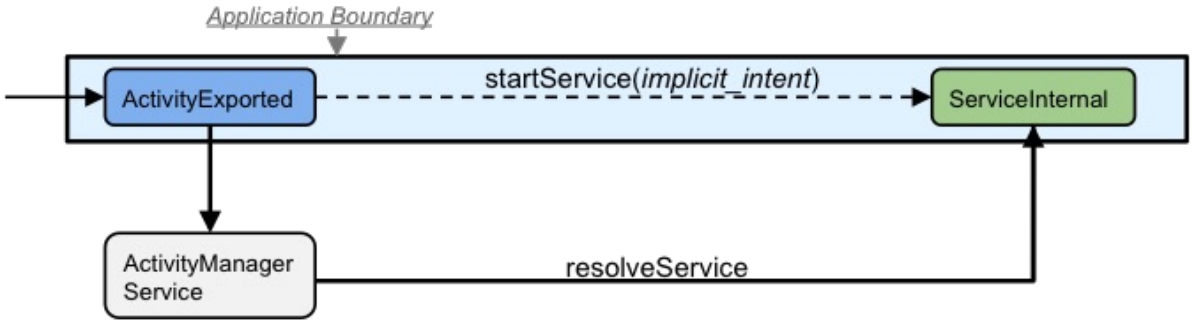


- *Intent Hijacking*: the ActivityManager is tricked into selecting a malicious target component



Intent Hijacking

- Recall: an *implicit intent* is an intent message where Android's ActivityManager selects the target.



- But, what if there is more than one match?
  - Activity: Ask the user!
  - Service: ?
    - *Random choice*

Broadcast Theft

- Anyone who registers for a broadcast can receive
  - No *hijacking* necessary
- What can we use to control who receives the broadcast?
  - Permissions!

```
<!--  Declaring the permission -->
<permission android:name="com.example.project.permission.BroadcastPerm"
            android:label="broadcastPerm"
            android:protectionLevel="signature/system">
</permission>
```

```
Intent broadcast = new Intent("com.example.project.Broadcast");
//Use the API: sendBroadcast (Intent intent, String receiverPermission)
sendBroadcast(broadcast, "com.example.project.permission.BroadcastPerm");
```

Basic analysis

- For each *intent object*, what do you look for?
  - Is the call using this *intent* "explicit"?
  - Does the *intent* have an action, flags, *extra data*?
- How to check for these characteristics?
  - Simple string/signature matching? *May* work in simple cases.
  - In most cases, data flow analysis may be required for a *practical* precision (as an intent can be modified over time).

```
String className = "A.class";
Intent intent = new Intent(className)
```

  - For few (or specific) apps, manual analysis is appropriate after some initial triaging.

The End