ARF: Identifying Re-Delegation Vulnerabilities in Android System Services

Sigmund Albert Gorski III and William Enck
North Carolina State University
Android Platform
Android Middleware

Application

Middleware

Linux Kernel

App1

App2

API

API

API

AC

AC

AC

Protected Operation

Protected Operation

Driver
The Middleware is Interconnected
Why Are These Interactions Complex?

- **Path**: A interconnected relationship between two or more APIs
- **Deputy**: The first API in a path
- **Target**: The last API in a path
Why Are These Interactions Complex?

- Authorization checks are always performed on the calling identity of the most immediate caller.
- The authorization checks of $E_1$ must be at least as restrictive as those of $E_2$. 

[Diagram showing the flow from Third Party App A (caller) through Middleware to API $E_1$ (deputy) to API $E_2$ (target)]
Re-Delegation Example

```java
Deputy
void enqueueToast(String pkg, ITransientNotification callback,
    int duration) {
    int callingPid = Binder.getCallingPid();
    long callingId = Binder.clearCallingIdentity();
    try {
        ...
        keepProcessAliveIfNeededLocked(callingPid);
        ...
    } finally {
        Binder.restoreCallingIdentity(callingId);
    }
}
void keepProcessAliveIfNeededLocked(int pid) {
    ...
    mAml.setProcessImportant(token, pid, toastCount > 0, "toast");
    ...
}
```
Re-Delegation Example

```java
void enqueueToast(String pkg, ITransientNotification callback,
                 int duration) {
    int callingPid = Binder.getCallingPid();
    long callingId = Binder.clearCallingIdentity();
    try {
        ...
        keepProcessAliveIfNeededLocked(callingPid);
        ...
    } finally {
        Binder.restoreCallingIdentity(callingId);
    }
}

void keepProcessAliveIfNeededLocked(int pid) {
    ...
    mA.mAm.setProcessImportant(token, pid, toastCount > 0, "toast");
    ...
}
```
Not As Simple as It Seems

Android 8.1.0
Interconnected API Graph
Challenges

• How do we compare the authorization checks?
• How do we efficiently analyze the heavily interconnected relationships of APIs?
Related Works Focus on Apps

- CHEX (CCS’12): looked at permission re-delegation vulnerabilities in third-party applications
- Woodpecker (NDSS’12): stock versions of system applications
- SEFA (CCS’13): vendor modified versions of system applications
- Felt et al. and Quire (USENIX’11): developed systems to prevent permission re-delegation in third-party applications
ARF Key Insights

• Assume system applications to be benign as malicious system applications represent a larger security concern (third-party applications considered malicious)
• Only need to consider permission checks since third-party application access is primarily enforced with permissions (permission re-delegation)
• Not all possible paths need to be analyzed for re-delegation (exposed single edge paths → exposed multi-edge paths)
ARF Design Overview

- **Step 1:** Run ACMiner (CODASPY ’19) to mine authorization checks and retrieve API edge relationships.
- **Step 2:** Analyze the single edge paths for re-delegation instances.
- **Step 3:** Identify multi-edge paths containing vulnerable single edge paths and analyze them for re-delegation instances.
Path Reduction

- All paths containing multiple edges consist of two or more single edge paths
- **No Re-Delegation**: The authorization checks of a target must be a subset of those in the deputy

\[ \text{No Re-Delegation} := AC(E_1) \supseteq AC(E_3) \]
Path Reduction

No Re-Delegation := $\text{AC}(E_1) \supseteq \text{AC}(E_2) \land \text{AC}(E_2) \supseteq \text{AC}(E_3)$

- **No Re-Delegation**: Each single edge path that constructs a multi-edge path must also not contain a re-delegation instance
Path Reduction

Re-Delegation: At least one single edge path that constructs a multi-edge path must contain a re-delegation instance.
Path Reduction

Re-Delegation := AC(E₁) ⊈ AC(E₂) ∨ AC(E₂) ⊈ AC(E₃)

2,816 paths with permission re-delegation

15,483 paths with permission re-delegation

Reduction in Paths with Permission Re-Delegation AOSP 8.1.0
Eliminating Non-Vulnerable Paths

- Still too many paths. Can we reduce them further?

<table>
<thead>
<tr>
<th>Common Causes of Non-Vulnerable Paths</th>
<th>Approx. Paths Removed</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Deputy or Target is an Authorization check</td>
<td>812</td>
</tr>
<tr>
<td>2. Target is Used in an Authorization Check</td>
<td>464</td>
</tr>
<tr>
<td>3. Deputy Does Not Require Multi-User Enforcement</td>
<td>102</td>
</tr>
<tr>
<td>4. Deputy Already Has Multi-User Enforcement</td>
<td>106</td>
</tr>
<tr>
<td>5. Deputy is Restricted to a Special Caller</td>
<td>671</td>
</tr>
<tr>
<td>6. Deputy and Target Have Equivalent Calling Identities</td>
<td>188</td>
</tr>
<tr>
<td>7. Target Can Only Be Called By Apps Compiled for Older Android Versions</td>
<td>54</td>
</tr>
</tbody>
</table>
Eliminating Non-Vulnerable Paths

- Explored the 2816 paths to find patterns in non-vulnerable paths
- Found 7 non-vulnerable path patterns
- Implemented techniques to remove paths from ARF output
Implementation

- ARF is constructed on top of ACMiner and uses the Soot static analysis framework to perform some path eliminations
- Consists of 3931 lines of java code
- ARF is applicable to all versions of Android (AOSP and OEMs)
Findings

- ARF identified vulnerabilities in 170 paths distributed across 86 unique deputies for Android 8.1.0.
- 320 reported paths had a deputy performing its role correctly.

<table>
<thead>
<tr>
<th>Type of Vulnerability</th>
<th>Number of Vulnerable Deputies</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. System State Modifications</td>
<td>6</td>
</tr>
<tr>
<td>2. Information Leaks</td>
<td>2</td>
</tr>
<tr>
<td>3. Disrupt Device Usability</td>
<td>8</td>
</tr>
<tr>
<td>4. Multi-User Enforcement</td>
<td>3</td>
</tr>
<tr>
<td>5. Keep the Device Awake</td>
<td>55</td>
</tr>
<tr>
<td>6. Non Re-Delegation Vulnerabilities</td>
<td>14</td>
</tr>
</tbody>
</table>
Re-Delegation Highlights

- **Impact**: Apps can keep a service running in the foreground
- **Problem**: Missing a check for the system level permission `SET_PROCESS_LIMIT`
- **Status**: Reported to Google
Re-Delegation Highlights

- **Impact:** Apps can determine if the device has a password, lock pattern, etc.
- **Problem:** No check for permission ACCESS_KEYGUARD_SECURE_STORAGE
- **Status:** Reported to Google

```java
Deputy

```java
boolean isDeviceSecure(int userId) {
    userId = ActivityManager.handleIncomingUser(getCallingPid(),
                                                     getCallingUid(), userId, false, true, "", null);
    long token = Binder.clearCallingIdentity();
    try {
        ...
        return mLockPatternUtils.isSecure(userId);
    }
    finally {
        Binder.restoreCallingIdentity(token);
    }
}

boolean isSecure(int u) {
    int m = getKeyguardStoredPasswordQuality(u);
    return isLockPatternEnabled(m,u) || isPasswordEnabled(m,u);
}
```
Re-Delegation Highlights

- **Impact:** Apps can discover the cache size of any other app on a volume
- **Problem:** No check for permission PACKAGE_USAGE_STATS
- **Status:** Reported to Google

```
1 long getFreeBytes(String volumeUid, String callingPackage) {
2     long token = Binder.clearCallingIdentity();
3     try {
4         File path = mStorage.findPathForUid(volumeUid);
5         if (isQuotaSupported(volumeUid, callingPackage)) {
6             final long total = getCacheBytes(volumeUid, callingPackage);
7             final long reserved = mStorage.getStorageCacheBytes(path, 0);
8             final long cClearable = Math.max(0, cTotal - cReserved);
9             return path.getAvailableSpace() + cClearable;
10         }
11         return path.getAvailableSpace();
12     } finally {
13         Binder.restoreCallingIdentity(token);
14     }
15 }
```

/*... Note: no permissions are required when calling these
 * APIs for your own package or UID. However, requesting
 * details for any other package requires the permission
 * PACKAGE_USAGE_STATS, which is a system-level permission
 * that will not be granted to normal apps. ...*/

```
Interactions With Google

• High severity and will be patched in the June 2019 security update
  – `getPermittedAccessibilityServicesForUser`: Allow any user to get a list of the accessibility services enabled for another user
  – `isPackageDeviceAdminOnAnyUser`: Allow any user to determine if an app is acting as a device administrator for another user
  – `areNotificationsEnabledForPackage`: Allow any user to determine if another user has enabled notifications for an app
  – `isSeparateProfileChallengeAllowed`: Enable any user to check if another user has a work profile security lock (e.g., password)
• Others considered moderate or low severity and will be fixed in the next major release of Android
Thank You!

- ARF’s key contribution is our reduction of the paths a domain expert needs to review for permission re-delegation vulnerabilities
  - Overall 97% reduction
- We plan on open sourcing ARF at a later date
- https://wspr.csc.ncsu.edu/arf/

Sigmund Albert Gorski III
PhD Student, WSPR Lab
NC State University
W: https://sigmundgorski.com
E: sagorski@ncsu.edu