Enhance Calling Definition Security for Android Custom Permission

Lanlan Pan (潘蓝兰)

abbypan@gmail.com

Outline

Introduction

- Custom permission is an important security feature of Android system to limit app's access to sensitive data.
- Any app can use custom permissions to share its resources with other apps, system provides the permission-based access control.

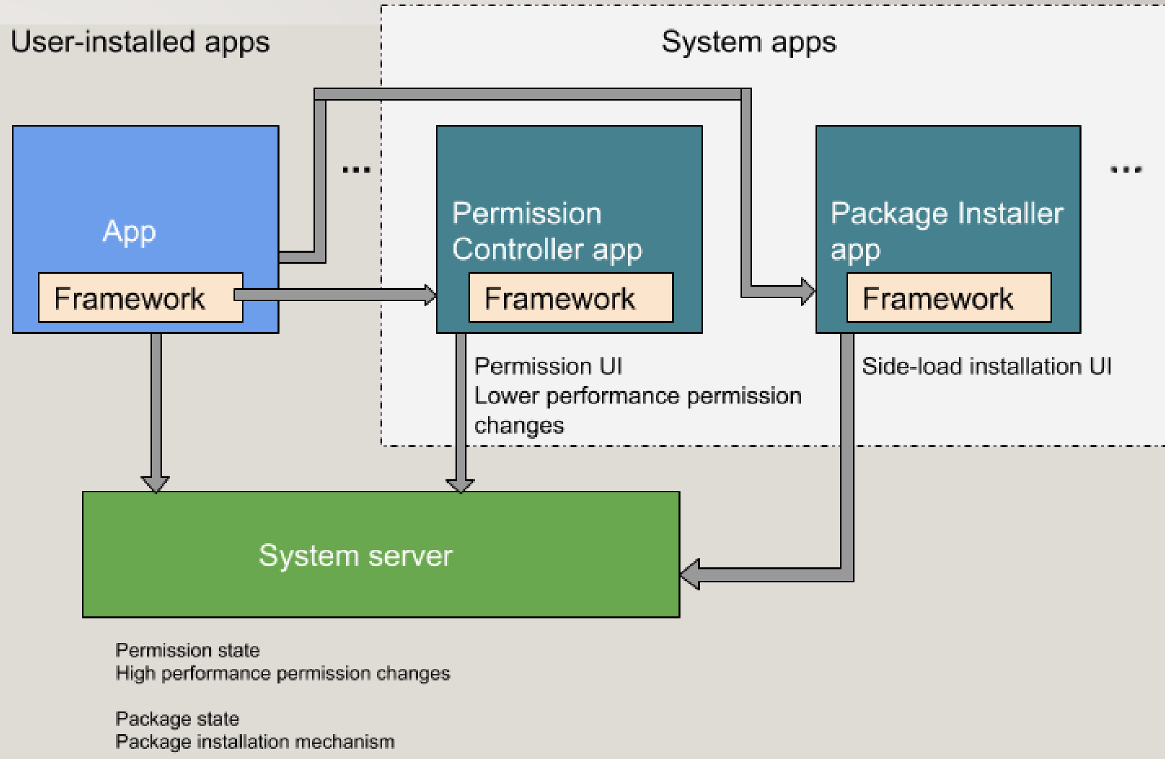
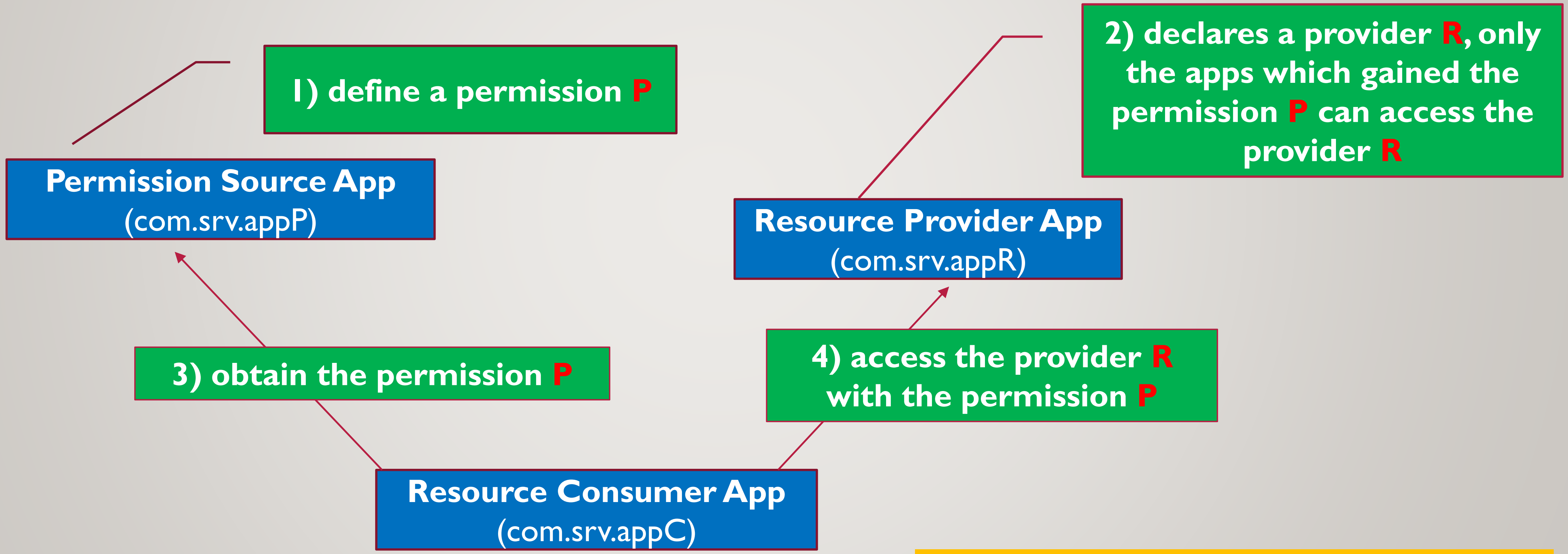

User-installed apps
System apps
App
Framework
...
Permission Controller app
Framework
Package Installer app
Framework
...
Permission UI
Lower performance permission changes
Side-load installation UI
System server
Permission state
High performance permission changes
Package state
Package installation mechanism

Figure is from https://source.android.com/docs/core/permissions

App Roles in Custom Permission Scenario

- **Permission Source App** defines a custom permission.
  - com.srv.appP, signed by legitimate developer's private key.
- **Resource Provider App** declares to provide some resources to the apps which have gained the custom permission defined by permission source app.
  - com.srv.appR, signed by legitimate developer's private key.
- **Resource Consumer App** requests to consume some resources which provided by resource provider app.
  - com.srv.appC, signed by legitimate developer's private key.

Sample of Custom Permission Usage


1) define a permission P
Permission Source App (com.srv.appP)
2) declares a provider R, only the apps which gained the permission P can access the provider R
Resource Provider App (com.srv.appR)
3) obtain the permission P
4) access the provider R with the permission P
Resource Consumer App (com.srv.appC)

permission P: "com.srv.appP.JUSTFORTEST"
provider R: "com.srv.appR.sourceprovider"

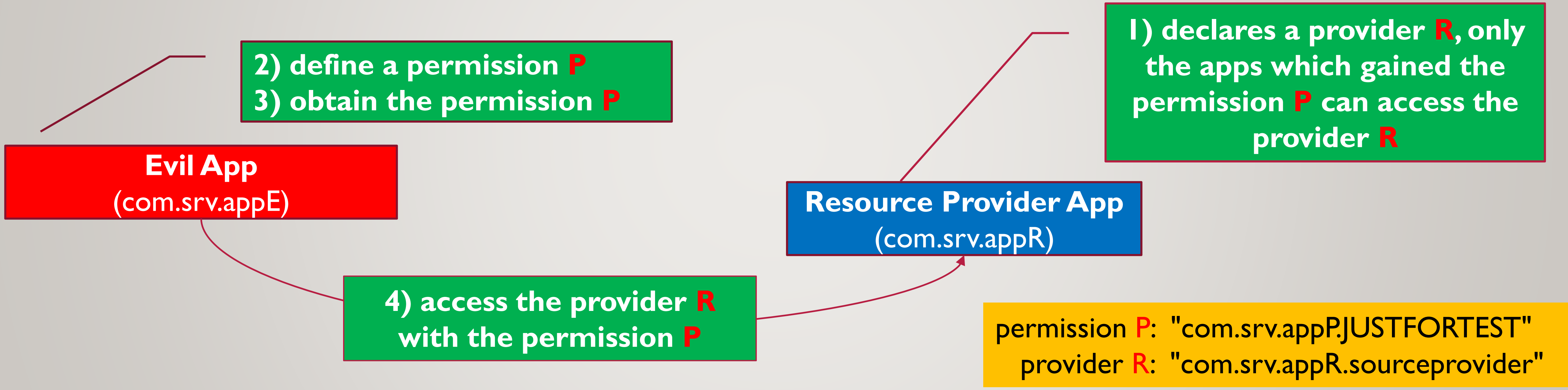
Permission Squatting Attack

- **Background:** Android system fully trusts the permission source app which first defined the custom permission by default.

- **Evil App** is developed by the attacker, makes permission squatting attacks, gets ahead of legitimate permission source app (com.srv.appP) to define the custom permission.
    - com.srv.appE, signed by attacker's private key.

Sample of Permission Squatting Attack

- User has installed resource provider app (com.srv.appR), but hasn't installed permission source app (com.srv.appP).
- Attacker developed an evil app (com.srv.appE), and coax user into installing evil app (com.srv.appE).


2) define a permission P
3) obtain the permission P
Evil App
(com.srv.appE)
1) declares a provider R, only the apps which gained the permission P can access the provider R
Resource Provider App
(com.srv.appR)
4) access the provider R with the permission P
permission P: "com.srv.appP.JUSTFORTEST"
provider R: "com.srv.appR.sourceprovider"

- Evil App can define and obtain the permission P, to access the provider R, which may lead to user data leakage.
- Moreover, because of permission definition collision, the user can't not install Permission Source App (com.srv.appP) if Evil App(com.srv.appE) has existed.

Evaluation of Permission Squatting Attack

We perform permission squatting attack evaluation on 6 mobile devices from the following manufacturers: Google, Huawei, OPPO, Samsung, Vivo, and Xiaomi.

Manufacturer	**Device Model**	**Operation System**	**Squatting Attack**
Google	Pixel 3	Android 12	✓
Huawei	P40	HarmonyOS 2.0.0	✓
OPPO	Find X5 Pro	Android 13	✓
Samsung	Galaxy S20 5G	Android 12	✓
Vivo	iQOO 7	Android 13	✓
Xiaomi	Mi 10	Android 12	✓

Code: https://github.com/JimmyChenX/SquattingAttackInAndroidCustomPermissions

Related Work

Existing Android custom permission security research has discussed the permission squatting attack, mostly focus on the permission vulnerabilities detection, and system access control policy improvement on permission.

- Reverse Domain Style Permission Name
- Naming Convention
- Disallow Custom Permission with The Same Name
- Revoke Permission
- Dynamic Enforcement

Reverse Domain Style Permission Name

- [Talegaon'19] suggest to regulate custom permission name with the reverse domain style.
  - For example, if application name "com.srv.appP" , then the permission defined by the app should follow "com.srv.appP.* " reverse domain style.

- Good
  - system can find the permission defined by other evil app quickly.
  - find if com.other.appX defined "com.srv.appP.JUSTFORTEST"
- Weak
  - the attacker can develop evil app with the same application name of legitimate app.

Naming Convention

- [Tuncay'18] introduce an internal naming convention, which enforces that all custom permission names are internally prefixed with the source id of the app that declares it.
  - the custom permission names are translated to source id : permission name.
  - To avoid the package name problem, they instead use the app's signature as the source id to prefix permission name.

- Good
  - avoid custom permission name collision.
- Weak
  - only works for the apps with same signature.
  - can't deal with the scenario when permission source app, resource provider app, and resource consumer app are signed by different private keys.

Disallow Custom Permission with The Same Name

- [Bagheri'18] suggest to disallow multiple apps that define a custom permission with the same name from simultaneously existing on the device.
    - For example, if app "com.srv.appP" has defined a permission "xxx", then any other app can not defined permission "xxx"

- Good
    - system can reject the permission defined by other evil app quickly.
    - find if com.other.appX defined "com.srv.appP.JUSTFORTEST"
- Weak
    - we can't assume that the legitimate app can be always installed before the evil app.
    - the attacker may uninstall legitimate app.

Revoke Permission

- [Li’21] suggest to revoke permission directly.
  - When the system removes a custom permission, its grants for apps should be revoked.
  - When the system takes the ownership of a custom permission, its grants for apps should be revoked.
  - During the permission update, its grants for apps should be revoked.

- Good
  - the custom permission access control policy is more clearly.
- Weak
  - attacker can attract users to grant the evil app’s custom permission again, with the same permission name, and users hardly to distinguish it.

Dynamic Enforcement

- [Hill'21] envision an approach in which both users and Android enterprise administrators can actively restrict the functionality of potential permission abusing apps by leveraging the dynamic permission updates provided by Android enterprise.

- Good
  - support dynamic permission policy update.
  - prevent the permission abusing, even when users has granted the permission.
- Weak
  - the management work is heavy, such as identify attack patterns and potential templates for counter-policies.

Our Proposal

There are few discussions about how to enhance resource provider app's self-protection on custom permission, which can help resource provider app to provide its resource to the right consumer app with the right permission defined by the right permission source app.

To address the permission squatting attack, we describe a permission source validation scheme for the resource provider app.

We add three permission attributes for the permission source validation:

- permission_srcSecondLevelDomainNames
- permission_srcProtectLevel
- permission_srcSignerCerts

permission_srcSecondLevelDomainNames

- permission_srcSecondLevelDomainNames means that, the resource provider app assumes that the permission source application name must belong to some second level domain names.
  - For example, Resource Provider App (com.srv.appR) declares a provider, the permission source application name should belong to "com.srv", or "com.service".
  - "com.srv.appP" matches "com.srv".

```
<string-array name="srcSecondLevelDomainNamesForP">
<item>com.srv</item>
<item>com.service</item>
</string-array>

<provider
android:authorities="com.srv.sourceprovider"
android:name="com.srv.appR.sourceprovider"
android:permission="com.srv.appP.JUSTFORTEST"
android:permission_srcSecondLevelDomainNames="@array/srcSecondLevelDomainNamesForP"
android:exported="true"
/>
```

permission_srcProtectLevel

- permission_srcProtectLevel means that, the resource provider app assumes that the permission source app should belong to one of the protect levels below:
  - permission_srcProtectLevel = signature: the permission source app should have the same signer certificate with the resource provider app.
  - permission_srcProtectLevel = privileged: the permission source app should be privileged app.
  - permission_srcProtectLevel = knownSigner: the permission source app's signer certificate digest should be listed in the permission_srcSignerCerts.

- For example, resource provider app R declares a provider, the permission source app should have the same signer certificate with the resource provider app.

```
<provider
android:authorities="com.srv.sourceprovider"
android:name="com.srv.appR.sourceprovider"
android:permission="com.srv.appP.JUSTFORTEST"
android:permission_srcProtectLevel="signature"
android:exported="true"
/>
```

permission_srcSignerCerts

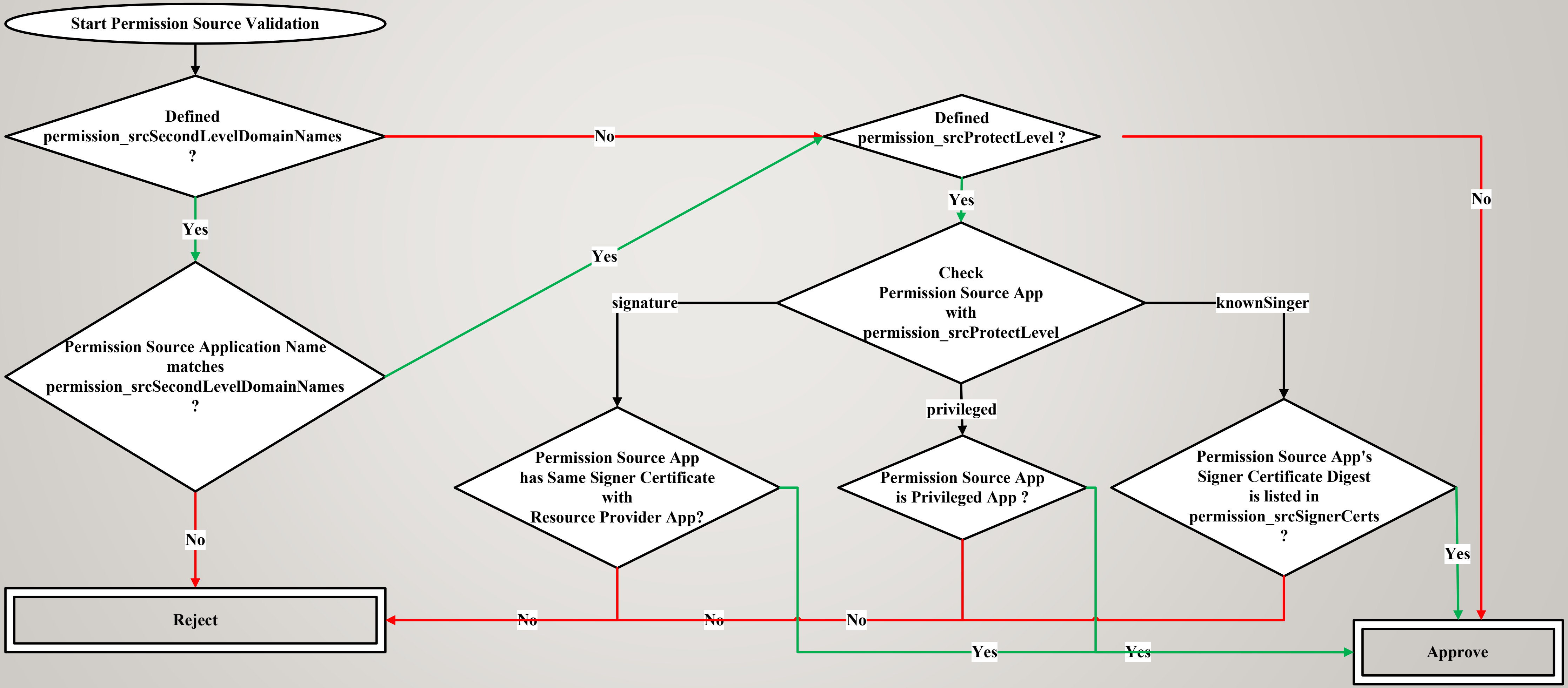
- permission_srcSignerCerts contains some signer certificate digests, which are used for the permission_srcProtectLevel = knownSigner scenario.
- The digest should be computed over the DER encoding of the signer certificate using the SHA-256 digest algorithm.

- For example, resource provider app R declares a provider, the permission source app's signer certificate digest should be listed in the permission_srcSignerCerts.

```
<string-array name="srcSignerCertsForP">
<item> 657d6f7c6295d453f027a8cc4ce528f411d95276cca140f540c53f396df1ceff </item>
</string-array>

<provider
android:authorities="com.srv.sourceprovider"
android:name="com.srv.appR.sourceprovider"
android:permission="com.srv.appP.JUSTFORTEST"
android:permission_srcProtectLevel="knownSigner"
android:permission_srcSingerCerts="@array/srcSignerCertsForP"
android:exported="true"
/>
```

Permission Source Validation for Resource Provider App



Start Permission Source Validation
Defined permission_srcSecondLevelDomainNames ?
No
Yes
Permission Source Application Name matches permission_srcSecondLevelDomainNames ?
Yes
No
Reject
Defined permission_srcProtectLevel ?
Yes
No
Check Permission Source App with permission_srcProtectLevel
signature
knownSinger
privileged
Permission Source App has Same Signer Certificate with Resource Provider App?
Permission Source App is Privileged App ?
Permission Source App's Signer Certificate Digest is listed in permission_srcSignerCerts ?
No
No
No
Yes
Yes
Yes
Approve

Discussion: Good

- Compared to reverse domain style permission name
  - permission_srcSecondLevelDomainName is more flexible for application name changing.
- Compared to naming convention
  - permission_srcProtectLevel = knownSigner could support the scenario when permission source app, resource provider app, and resource consumer app are signed by different private keys.
- Compared to disallow custom permission with the same name
  - permission_srcProtectLevel=signature/knownSigner could prevent the evil app from getting resource provider's data even when the evil app has successful declared the same permission name on Android system.
- Compared to revoke permission
  - permission_srcProtectLevel=signature/privileged/knownSigner could support more loose access control policy with less permission revoked work, because the system can confirm that the permission is updated by some legitimate permission source app through permission_srcProtectLevel check.

Discussion: Weak

- permission_srcSecondLevelDomainName: Resource Provider App should update its manifest when the Permission Source App's second level domain name is changed.

- permission_srcProtectLevel=signature/knownSigner: Resource Provider App should be resigned/update its manifest when the Permission Source App's signer certificate is changed.

- permission_srcSignerCerts: Resource Provider App should manage the Permission Source App's signer certificate digests on its manifest.

Conclusion

- Our Work
    - We describe a scheme to provide permission source validation for the resource provider apps, which can be resistant to permission squatting attack.
    - We define three permission attributes to enhance the calling context security for android custom permission, which are suitable for resource provider app's self-protection.
- Limitation
    - We don't discuss about the system how to identify evil app on app store, or the system how to make malware detection when installing app.
    - We don't mention about the attacker how to attract user to install evil app before legitimate permission source app.
- Future Work
    - Do more experiments on android system.
    - Suggest Google to implement our scheme in the future android version.

THANK YOU

Q&A

Reference

- [Talegaon'19] Talegaon, S., & Krishnan, R. (2020). A formal specification of access control in android. In Secure Knowledge Management In Artificial Intelligence Era: 8th International Confer-ence, SKM 2019, Goa, India, December 21–22, 2019, Proceedings 8 (pp. 101-125). Springer Singapore.
- [Tuncay'18] Tuncay, G. S., Demetriou, S., Ganju, K., & Gunter, C. (2018). Resolving the predicament of android custom permissions.
- [Bagheri'18] Bagheri, H., Kang, E., Malek, S., & Jackson, D. (2018). A formal approach for detection of security flaws in the android permission system. Formal Aspects of Computing, 30, 525-544.
- [Li'21] Li, R., Diao, W., Li, Z., Du, J., & Guo, S. (2021, May). Android custom permissions de-mystified: From privilege escalation to design shortcomings. In 2021 IEEE Symposium on Security and Privacy (SP) (pp. 70-86). IEEE.
- [Hill'21] Hill, M., Rubio-Medrano, C. E., Claramunt, L. M., Baek, J., & Ahn, G. J. (2021, September). Poster: DyPolDroid: User-Centered Counter-Policies Against Android Permission-Abuse Attacks. In 2021 IEEE European Symposium on Security and Privacy (EuroS&P) (pp. 704-706). IEEE.