账号安全方案？

AccountManagerService负责管理手机中用户的在线账户，主要的工作涉及账户的添加、删除和AuthToken的获取和更新

# TODO

<https://blog.csdn.net/dzkdxyx/article/details/78569867>

<https://blog.csdn.net/dzkdxyx/article/details/78632945>

http://wiki.jikexueyuan.com/project/deep-android-v2/content-account.html

# 启动流程

## startService

面看下AccountManagerService的初始化，进入SystemServer中的startOtherServices方法看到如下代码：

mSystemServiceManager.startService(ACCOUNT\_SERVICE\_CLASS);

其中ACCOUNT\_SERVICE\_CLASS为com.android.server.accounts.AccountManagerService$Lifecycle，SystemServiceManager的startService里面会通过反射创建AccountManagerService.Lifecycle对象，并调用其onStart方法：

## Lifecycle.onStart

public static class Lifecycle extends SystemService {

public void onStart() {

mService = new AccountManagerService(getContext());

publishBinderService(Context.ACCOUNT\_SERVICE, mService);

}

## New AccountManagerService

可以看到创建AccountManagerService对象，并发布到服务总管里面。下面看下AccountManagerService构造函数：

public AccountManagerService(Context context) {

this(context, context.getPackageManager(), new AccountAuthenticatorCache(context));

}

public AccountManagerService(Context context, PackageManager packageManager,

IAccountAuthenticatorCache authenticatorCache) {

mContext = context;

mPackageManager = packageManager;

mAppOpsManager = mContext.getSystemService(AppOpsManager.class);

mMessageHandler = new MessageHandler(FgThread.get().getLooper());

mAuthenticatorCache = authenticatorCache;

mAuthenticatorCache.setListener(this, null /\* Handler \*/);

## AccountAuthenticatorCache

这里又创建了AccountAuthenticatorCache对象并设置监听器，AccountAuthenticatorCache是android平台中账户验证服务（AAS）的管理中心，AAS是应用程序中定义的服务，对它的定义有一定的要求，后面会看到。接着看下AccountAuthenticatorCache的构造函数

### AccountAuthenticatorCache

/\* package private \*/ class AccountAuthenticatorCache

extends RegisteredServicesCache<AuthenticatorDescription>

public AccountAuthenticatorCache(Context context) {

super(context, AccountManager.ACTION\_AUTHENTICATOR\_INTENT,

AccountManager.AUTHENTICATOR\_META\_DATA\_NAME,

AccountManager.AUTHENTICATOR\_ATTRIBUTES\_NAME, sSerializer);

}

这里又调用父类RegisteredServicesCache构造函数，传递的参数如下定义：

public static final String ACTION\_AUTHENTICATOR\_INTENT =

"android.accounts.AccountAuthenticator";

public static final String AUTHENTICATOR\_META\_DATA\_NAME =

"android.accounts.AccountAuthenticator";

public static final String AUTHENTICATOR\_ATTRIBUTES\_NAME = "account-authenticator";

后面解析XML要用到，所以AAS的配置文件也需要符合这个要求。

### RegisteredServicesCache

Asdf

public RegisteredServicesCache(Context context, String interfaceName, String metaDataName,

String attributeName, XmlSerializerAndParser<V> serializerAndParser) {

mContext = context;

mInterfaceName = interfaceName;

mMetaDataName = metaDataName;

mAttributesName = attributeName;

mSerializerAndParser = serializerAndParser;

migrateIfNecessaryLocked();

IntentFilter intentFilter = new IntentFilter();

intentFilter.addAction(Intent.ACTION\_PACKAGE\_ADDED);

intentFilter.addAction(Intent.ACTION\_PACKAGE\_CHANGED);

intentFilter.addAction(Intent.ACTION\_PACKAGE\_REMOVED);

intentFilter.addDataScheme("package");

mContext.registerReceiverAsUser(mPackageReceiver, UserHandle.ALL, intentFilter, null, null)

#### migrateIfNecessaryLocked

其中migrateIfNecessaryLocked用户在系统目录下创建文件：

File systemDir = new File(getDataDirectory(), "system");

File syncDir = new File(systemDir, REGISTERED\_SERVICES\_DIR);

AtomicFile oldFile = new AtomicFile(new File(syncDir, mInterfaceName + ".xml"))

#### mPackageReceiver

接着注册了包添加删除广播。猜测应用安装或卸载会在这里进行账户信息添加和删除。下面看下这个广播接收器：

private final BroadcastReceiver mPackageReceiver = new BroadcastReceiver() {

@Override

public void onReceive(Context context, Intent intent) {

final int uid = intent.getIntExtra(Intent.EXTRA\_UID, -1);

if (uid != -1) {

handlePackageEvent(intent, UserHandle.getUserId(uid));

}

}

};

##### handlePackageEvent

接着进入handlePackageEvent：

private final void handlePackageEvent(Intent intent, int userId) {

// Don't regenerate the services map when the package is removed or its

// ASEC container unmounted as a step in replacement. The subsequent

// \_ADDED / \_AVAILABLE call will regenerate the map in the final state.

final String action = intent.getAction();

// it's a new-component action if it isn't some sort of removal

final boolean isRemoval = Intent.ACTION\_PACKAGE\_REMOVED.equals(action)

|| Intent.ACTION\_EXTERNAL\_APPLICATIONS\_UNAVAILABLE.equals(action);

// if it's a removal, is it part of an update-in-place step?

final boolean replacing = intent.getBooleanExtra(Intent.EXTRA\_REPLACING, false);

if (isRemoval && replacing) {

// package is going away, but it's the middle of an upgrade: keep the current

// state and do nothing here. This clause is intentionally empty.

} else {

int[] uids = null;

// either we're adding/changing, or it's a removal without replacement, so

// we need to update the set of available services

if (Intent.ACTION\_EXTERNAL\_APPLICATIONS\_AVAILABLE.equals(action)

|| Intent.ACTION\_EXTERNAL\_APPLICATIONS\_UNAVAILABLE.equals(action)) {

uids = intent.getIntArrayExtra(Intent.EXTRA\_CHANGED\_UID\_LIST);

} else {

int uid = intent.getIntExtra(Intent.EXTRA\_UID, -1);

if (uid > 0) {

uids = new int[] { uid };

}

}

generateServicesMap(uids, userId);

}

##### generateServicesMap

当包正在删除且部分在更新情况不做任何处理，接着进入generateServicesMap：

private void generateServicesMap(int[] changedUids, int userId) {

if (DEBUG) {

Slog.d(TAG, "generateServicesMap() for " + userId + ", changed UIDs = " + changedUids);

}

final ArrayList<ServiceInfo<V>> serviceInfos = new ArrayList<ServiceInfo<V>>();

final List<ResolveInfo> resolveInfos = queryIntentServices(userId);

for (ResolveInfo resolveInfo : resolveInfos) {

try {

ServiceInfo<V> info = parseServiceInfo(resolveInfo);

if (info == null) {

Log.w(TAG, "Unable to load service info " + resolveInfo.toString());

continue;

}

serviceInfos.add(info);

} catch (XmlPullParserException|IOException e) {

Log.w(TAG, "Unable to load service info " + resolveInfo.toString(), e);

}

}

首先传递userid通过过PMS查询应用包服务信息。接着通过parseServiceInfo解析服务信息：

##### parseServiceInfo

protected ServiceInfo<V> parseServiceInfo(ResolveInfo service)

throws XmlPullParserException, IOException {

android.content.pm.ServiceInfo si = service.serviceInfo;

ComponentName componentName = new ComponentName(si.packageName, si.name);

PackageManager pm = mContext.getPackageManager();

XmlResourceParser parser = null;

parser = si.loadXmlMetaData(pm, mMetaDataName);

AttributeSet attrs = Xml.asAttributeSet(parser);

V v = parseServiceAttributes(pm.getResourcesForApplication(si.applicationInfo),

si.packageName, attrs);

if (v == null) {

return null;

}

final android.content.pm.ServiceInfo serviceInfo = service.serviceInfo;

return new ServiceInfo<V>(v, serviceInfo, componentName);

解析MetaData信息，接着调用子类parseServiceAttributes来解析MetaData中的resource信息。

### parseServiceAttributes

public AuthenticatorDescription parseServiceAttributes(Resources res,

String packageName, AttributeSet attrs) {

TypedArray sa = res.obtainAttributes(attrs,

com.android.internal.R.styleable.AccountAuthenticator);

try {

final String accountType =

sa.getString(com.android.internal.R.styleable.AccountAuthenticator\_accountType);

final int labelId = sa.getResourceId(

com.android.internal.R.styleable.AccountAuthenticator\_label, 0);

final int iconId = sa.getResourceId(

com.android.internal.R.styleable.AccountAuthenticator\_icon, 0);

final int smallIconId = sa.getResourceId(

com.android.internal.R.styleable.AccountAuthenticator\_smallIcon, 0);

final int prefId = sa.getResourceId(

com.android.internal.R.styleable.AccountAuthenticator\_accountPreferences, 0);

final boolean customTokens = sa.getBoolean(

com.android.internal.R.styleable.AccountAuthenticator\_customTokens, false);

if (TextUtils.isEmpty(accountType)) {

return null;

}

return new AuthenticatorDescription(accountType, packageName, labelId, iconId,

smallIconId, prefId, customTokens);

} finally {

sa.recycle();

}

## MetaData

MetaData的resource一般是xml文件，下面举例SimContact模块中的AAS（SimAuthenticateService），看下它的配置文件和MetaDta的resource：



上面xml文件中的accountType标签用于指定账户类型，icon、smallIcon、label等用于界面显示。最终将sim\_authenticator文件解析封装到AuthenticatorDescription对象中返回。

上面xml文件中的accountType标签用于指定账户类型，icon、smallIcon、label等用于界面显示。最终将sim\_authenticator文件解析封装到AuthenticatorDescription对象中返回。

# 添加账户

一般默认添加账户都是在**设置**里面添加，应用程序也可以自己添加。添加账户入口在AccountManager类的addAccount：

## addAccount

public AccountManagerFuture<Bundle> addAccount(final String accountType,

final String authTokenType, final String[] requiredFeatures,

final Bundle addAccountOptions,

final Activity activity, AccountManagerCallback<Bundle> callback, Handler handler) {

return new AmsTask(activity, handler, callback) {

@Override

public void doWork() throws RemoteException {

mService.addAccount(mResponse, accountType, authTokenType,

requiredFeatures, activity != null, optionsIn);

}

}.start();

}

这里返回AmsTask对象，这是什么鬼？

private abstract class AmsTask extends FutureTask<Bundle> implements AccountManagerFuture<Bundle> {

final IAccountManagerResponse mResponse;

final Handler mHandler;

final AccountManagerCallback<Bundle> mCallback;

final Activity mActivity;

public AmsTask(Activity activity, Handler handler, AccountManagerCallback<Bundle> callback) {

super(new Callable<Bundle>() {

@Override

public Bundle call() throws Exception {

throw new IllegalStateException("this should never be called");

}

});

mHandler = handler;

mCallback = callback;

mActivity = activity;

mResponse = new Response();

}

public final AccountManagerFuture<Bundle> start() {

try {

doWork();

} catch (RemoteException e) {

setException(e);

}

return this;

}

public abstract void doWork() throws RemoteException;

private class Response extends IAccountManagerResponse.Stub {

@Override

public void onResult(Bundle bundle) {

Intent intent = bundle.getParcelable(KEY\_INTENT);

mActivity.startActivity(intent);

}

@Override

public void onError(int code, String message) {

}

}

它继承FutureTask实现AccountMana Future，关于FutureTask介绍可查看链接 FutureTask 深度解析 ，构造函数里面还创建了Response对象，它继承IAccountManagerResponse.Stub，根据经验它是服务的实现者，用于响应AccountManagerService的回调。接着调用start-》dowork（）进入到AccountManagerService中的addAccount：

## AccountManagerService。addAccount

public void addAccount(final IAccountManagerResponse response, final String accountType,

final String authTokenType, final String[] requiredFeatures,

final boolean expectActivityLaunch, final Bundle optionsIn) {

UserAccounts accounts = getUserAccounts(usrId);

logRecordWithUid(

accounts, DebugDbHelper.ACTION\_CALLED\_ACCOUNT\_ADD, TABLE\_ACCOUNTS, uid);

new Session(accounts, response, accountType, expectActivityLaunch,

true /\* stripAuthTokenFromResult \*/, null /\* accountName \*/,

false /\* authDetailsRequired \*/, true /\* updateLastAuthenticationTime \*/) {

@Override

public void run() throws RemoteException {

mAuthenticator.addAccount(this, mAccountType, authTokenType, requiredFeatures,

options);

}

@Override

protected String toDebugString(long now) {

return super.toDebugString(now) + ", addAccount"

+ ", accountType " + accountType

+ ", requiredFeatures "

+ (requiredFeatures != null

? TextUtils.join(",", requiredFeatures)

: null);

}

}.bind();

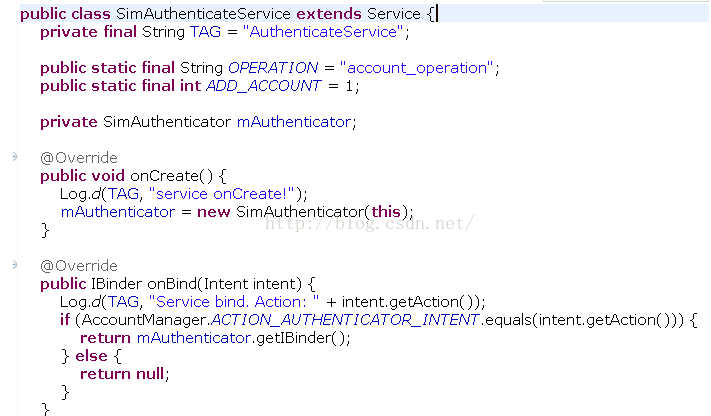
这里创建了一个Session对象，这又是什么鬼？

### Session

Sd

1. private abstract class Session extends IAccountAuthenticatorResponse.Stub
2. implements IBinder.DeathRecipient, ServiceConnection {
3. IAccountManagerResponse mResponse;
4. IAccountAuthenticator mAuthenticator = null;
5. public Session(UserAccounts accounts, IAccountManagerResponse response, String accountType,
6. boolean expectActivityLaunch, boolean stripAuthTokenFromResult, String accountName,
7. boolean authDetailsRequired, boolean updateLastAuthenticatedTime) {
8. super();
9. mAccounts = accounts;
10. mResponse = response;
11. mAccountType = accountType;
12. mAccountName = accountName;
13. synchronized (mSessions) {
14. mSessions.put(toString(), this);
15. }
16. }
17. void bind() {
18. if (!bindToAuthenticator(mAccountType)) {
19. }
20. }
21. public void onServiceConnected(ComponentName name, IBinder service) {
22. mAuthenticator = IAccountAuthenticator.Stub.asInterface(service);
23. run();
24. }
25. public abstract void run() throws RemoteException;
26. public void onResult(Bundle result) {
27. }
28. private boolean bindToAuthenticator(String authenticatorType) {
29. final AccountAuthenticatorCache.ServiceInfo<AuthenticatorDescription> authenticatorInfo;
30. authenticatorInfo = mAuthenticatorCache.getServiceInfo(
31. AuthenticatorDescription.newKey(authenticatorType), mAccounts.userId);
32. Intent intent = new Intent();
33. intent.setAction(AccountManager.ACTION\_AUTHENTICATOR\_INTENT);
34. intent.setComponent(authenticatorInfo.componentName);
35. if (Log.isLoggable(TAG, Log.VERBOSE)) {
36. Log.v(TAG, "performing bindService to " + authenticatorInfo.componentName);
37. }
38. if (!mContext.bindServiceAsUser(intent, this, Context.BIND\_AUTO\_CREATE,
39. UserHandle.of(mAccounts.userId))) {
40. }
41. return true;
42. }

它继承IAccountAuthenticatorResponse.Stub，是服务的实现者，跟谁通信？后面再看，又实现ServiceConnection，感觉在绑定服务的时候见过。接着调用bind()-》bindToAuthenticator，这里通过mAuthenticatorCache获取服务信息，mAuthenticatorCache在前面认识AccountManagerService介绍过。获取到ServiceInfo信息就开始绑定服务。这里以SimAuthenticateService为例，bindServiceAsUser调用后就会调用它的onBind方法：



这里的SimAuthenticator继承AbstractAccountAuthenticator，返回它的getIBinder()，其实是AbstractAccountAuthenticator中的内部类Transport对象的binder对象：

private Transport mTransport = new Transport();

/\*\*

\* @return the IBinder for the AccountAuthenticator

\*/

public final IBinder getIBinder() {

return mTransport.asBinder();

}

private class Transport extends IAccountAuthenticator.Stub {

@Override

public void addAccount(IAccountAuthenticatorResponse response, String accountType,

可见Transport又是服务的实现者，猜测在AccountManagerService会调用。

回到上面AccountManagerService中的添加服务流程中，刚才走到绑定服务，绑定成功会调用onServiceConnected函数，该函数中获取服务代理对象：

mAuthenticator = IAccountAuthenticator.Stub.asInterface(service);

接着run：

public void run() throws RemoteException {

mAuthenticator.addAccount(this, mAccountType, authTokenType, requiredFeatures,

options);

}

mAuthenticator为本地代理对象，调用addAccount会根据binder机制进入服务的实现者，也就是AbstractAccountAuthenticator中的内部类Transport：

final Bundle result = AbstractAccountAuthenticator.this.addAccount(

new AccountAuthenticatorResponse(response)

if (result != null) {

response.onResult(result);

}

这里调用AbstractAccountAuthenticator的子类，也就是上面举例的SimAuthenticator，注意上面传递的IAccountAuthenticatorResponse 的binder对象response，它的真正实现者在AccountManagerService内部类Session（private abstract class Session extends IAccountAuthenticatorResponse.Stub），也就是说这里调用response.onResult(result);又回到了Session中：

public void onResult(Bundle result) {

IAccountManagerResponse response;

response = mResponse;

if (response != null) {

response.onResult(result);

}

}

这里的response又是个binder对象。额，已经晕了。仔细回忆，它是在AccountManager的addAccount传递进来的，它是AmsTask的内部类Response：

private class Response extends IAccountManagerResponse.Stub {

@Override

public void onResult(Bundle bundle) {

Intent intent = bundle.getParcelable(KEY\_INTENT);

if (intent != null && mActivity != null) {

// since the user provided an Activity we will silently start intents

// that we see

mActivity.startActivity(intent);

整个流程差不多介绍完了

最后看下SimAuthenticator是如何添加服务的，它调用AccountManager的addAccountExplicitl

## addAccountExplicitl

public boolean addAccountExplicitly(Account account, String password, Bundle userdata) {

if (account == null) throw new IllegalArgumentException("account is null");

try {

return mService.addAccountExplicitly(account, password, userdata);

} catch (RemoteException e) {

throw e.rethrowFromSystemServer();

}

}

直接进入AccountManagerService的addAccountExplicitly：

public boolean addAccountExplicitly(Account account, String password, Bundle extras) {

UserAccounts accounts = getUserAccounts(userId);

return addAccountInternal(accounts, account, password, extras, callingUid);

## addAccountInternal

private boolean addAccountInternal(UserAccounts accounts, Account account, String password,

Bundle extras, int callingUid) {

final SQLiteDatabase db = accounts.openHelper.getWritableDatabaseUserIsUnlocked();

db.beginTransaction();

ContentValues values = new ContentValues();

values.put(ACCOUNTS\_NAME, account.name);

values.put(ACCOUNTS\_TYPE, account.type);

values.put(ACCOUNTS\_PASSWORD, password);

long accountId = db.insert(CE\_TABLE\_ACCOUNTS, ACCOUNTS\_NAME, values);

values = new ContentValues();

values.put(ACCOUNTS\_ID, accountId);

values.put(ACCOUNTS\_NAME, account.name);

values.put(ACCOUNTS\_TYPE, account.type);

values.put(ACCOUNTS\_LAST\_AUTHENTICATE\_TIME\_EPOCH\_MILLIS,

System.currentTimeMillis());

if (db.insert(TABLE\_ACCOUNTS, ACCOUNTS\_NAME, values) < 0)

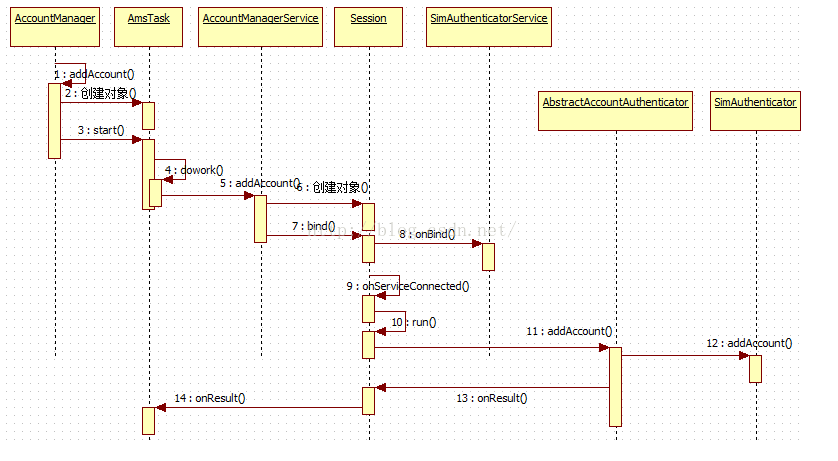
db.endTransaction();

}

sendAccountsChangedBroadcast(accounts.userId);

这里做了添加数据库的操作和发送账户修改广播。

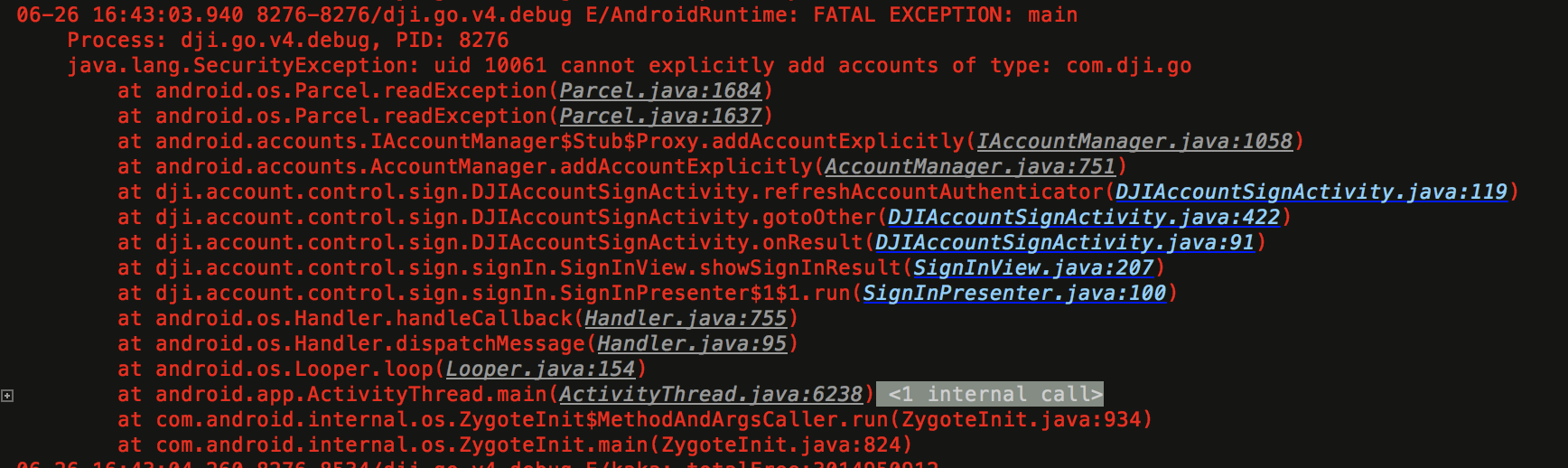
## 最后贴一张添加服务的流程图：



# REF

[基于N源码的AccountManagerService简单认识和账户添加流程分析](https://blog.csdn.net/yq6073025/article/details/52808094)

# 账号共享



uid

callingUid=10061

account.type = com.dji.go

String msg = String.format(

"uid %s cannot explicitly add accounts of type: %s",

callingUid,

account.type);

isAccountManagedByCaller(account.type, callingUid, userId)

isAccountManagedByCaller

return getTypesManagedByCaller(callingUid, userId).contains(accountType);

private List<String> getTypesManagedByCaller(int callingUid, int userId) {

return getTypesForCaller(callingUid, userId, false);

}

**private List<String> getTypesForCaller(**

**int callingUid, int userId, boolean isOtherwisePermitted) {**

**List<String> managedAccountTypes = new ArrayList<>();**

**long identityToken = Binder.clearCallingIdentity();**

**Collection<RegisteredServicesCache.ServiceInfo<AuthenticatorDescription>> serviceInfos;**

**try {**

**serviceInfos = mAuthenticatorCache.getAllServices(userId);**

**} finally {**

**Binder.restoreCallingIdentity(identityToken);**

**}**

**for (RegisteredServicesCache.ServiceInfo<AuthenticatorDescription> serviceInfo :**

**serviceInfos) {**

**final int sigChk = mPackageManager.checkSignatures(serviceInfo.uid, callingUid);**

**if (isOtherwisePermitted || sigChk == PackageManager.SIGNATURE\_MATCH) {**

**managedAccountTypes.add(serviceInfo.type.type);**

**}**

**}**

**return managedAccountTypes;**

**}**