Chapter 6: Empowering the Applications with Security (w.r.t. Android)

6.1 Permissions for Application

Android carries a different technique of running apps than traditional, desktop-based systems. On traditional systems, apps run underneath the account using the user who started them, and run with whatever permissions were granted compared to that user account. There won't be per-app separation mechanisms. Furthermore, all apps running there under same user account contain the identical higher level of access system APIs and other services provided by the root environment; both a document viewer and also a VoIP application have similar amount of use of the networking system merely because run beneath same UID automatically. Whenever a user has full root having access to a system, any app that user started will run with full having access to the system, and many types of the information about it, automatically.

This is the primary assumption on this traditional security model: all apps running on behalf of the individual ought to be granted an identical privileges and permissions. Inside Android model, each app runs as its own user account. It's the issue of separating the apps and ensuring they will access only her or his data, not data owned by other apps, automatically. Android then goes further and applies a thorough permissions system to services which might be provided to be used of installed apps. In order make use of services offered by other code with an Android device that may be sensitive or dangerous, like accessing an individual’s personal data or opening a web site connection, an app must first request and change into granted permission on the device’s user. Android uses an install-time permission request model, where an app specifies which of the permissions it entails rolling around in their manifest. Ahead of the app is usually installed, the customer must assess the listing of very damaging stuff the app is requesting if the programmer is allowed to do and approve them before installation is allowed to continue. This contains the effect of informing an individual what things the app would be able to do if installed and allows the
individual to come up with an informed decision about whether those permissions add together before granting them.

This model provides two primary benefits over other ones. First, it allows an individual to learn, before an app is installed, the countless dangerous issues that the app could do. Apps must specify which of people dangerous actions they might waste their manifest or tries to complete them will fail, considering that the apps will lack permission to complete them. A user downloading an activity that runs entirely locally notice the sport is requesting permission to access SMS messages, make phones calls, and discover full Entry to the world wide web, which isn't going to seem sensible whatsoever, tending to not install the action. Such a knowledge isn't on the available to an individual installing the identical game using a traditional computer's desktop system; it doesn’t must enumerate the harmful actions it could possibly perform and tell anyone upfront.

Second, this permissions model allows containment associated with an attack on the legitimate app. Apps inevitably contain coding problems and in some cases, errors in apps allow skilled attackers to consider inside the running app and cause their particular arbitrary code to operate within the same context as the compromised app (with just one UID combined with same permissions).

Inside a traditional system, whenever a web browser’s scripting engine was compromised by an attacker, the attacker’s code would run while using the full privileges in the browser, which are corresponding to everyone in the privileges about the user that started the browser. With an Android device, a compromised browser would run the attacker’s code when using the full privileges in the compromised app and would therefore get on the permissions that the app has declared and been granted. Nevertheless the browser might possibly may not have requested permission to the touch the user’s personal data (or it would have…consider some of the pros and cons of getting a browser being able to access all of the personal data, and what does a risk analysis of letting it to take action reveal?) and then the attacker wouldn't possess the capacity to make use of this type of compromise to extract that data.

The top question that naturally arises, once one knows this permissions model, is, “Do users really see the permissions requested with the new app making an informed decision about installing or otherwise not installing an app?” Basically, will users just approve any permissions set on an app to only use (and even just try) or would they really actually value exactly what is in the list they consent to? This can be a valid
question, considering that the permissions model is effective given that this actually happens. Fundamentally, there isn't way of force an individual to seriously know what they're agreeing to, which is the reason description for permissions tend to be kept short and easily understandable. It truly is concerning somebody, using her own judgment, to scrutinize the permissions list for the new app. Many individuals will approve everything to play a sport they must try to many folks will think for some time about granting an app any dangerous permissions. Developers use a key role with this process, too. If users will usually be assigned details on dangerous permissions for any single app they install, they're going to fatigue of reviewing them and keep approving them. If, however, the amount of permissions requested by apps is kept low, users could be more serious to scrutinize permissions that are presented in their apps. By minimizing the total and type of permissions the app requests, the programmer promotes the overall security within the.
It can be interesting to recollect, and bear in mind, that sometimes the motivation behind a permission request probably usually are not obvious. For example, in Android 1.5 and before, all applications had using of secondary storage (just like an sdcard). So, when the application targets version 1.5 from the platform or before, that application will forever request the WRITE_EXTERNAL_STORAGE permission, create application doesn't need any intention of ever accessing that storage. To ensure that is why the Android permissions model was created the way it is. It's a modern approach, seen most prominently in Android today, to handle the certainty which the fundamental tenant still driving desktop security models today—that all apps running with respect to somebody ought to be granted the identical privileges and permissions—has stopped being a valid technique to design a security model. By implementing this app-specific, permissions based model, Android supplies both the primary features about allowing a user to understand every dangerous action an app might perform and mitigating the final results associated with an app being compromised by an adversary. Considering that the design principles within the model are understood, let’s proceed to the way it is implemented in Android. Also, remember Android stands out as the only current mobile platform using this type of permission system built-in and meant for applications. The chance to limit software’s behaviors and capabilities generally is a significant and powerful capability that other major platforms tend not to offer. It truly is both a wonderful strength while
using Android platform in addition to a great opportunity for user as being a developer, to generate better applications.

Some current malware that target Android devices tries to trick the user into installing them and granting a considerable set of permissions. A frequent way of attackers is usually to complete a malicious application could be seen as an innocent this anyone wants to install. Another common way is usually to reissue the most beneficial application through an “added on” module, being sure that somebody downloading a legit application from a non-trusted source would acquire that application by malware riding along...with all the identical permissions granted for the application.

6.1.1 Permission for Aapplications’ in Android

The Android permission model will depend on the central construct of the permission. Permission is something which is granted to apps and essential to APIs (or certain functions within code) in order to run. An easy example would be the Android INTERNET permission; a variety of networking APIs desire a calling app to have this permission in order to execute the requested action of allowing network communications. Opening an outgoing network connection can be something that will not all apps should can do, and an activity that could make user to incur additional charges whenever they spend on the total amount of web data their device sends or receives plus, this is often the normal way an app would send sensitive data over device to be exploited. So, opening a network connection is restricted due to the INTERNET permission.

Android apps request permissions when installed. As a developer, that the programmer is responsible for determining which permissions the app requires and specifying all these permissions within the AndroidManifest.xml file. One of these is, if the app planned to communicate within the network, it could require an entry like this from the manifest:
INTERNET may be a permission created and utilized through the conventional Android libraries. The app, by specifying it, says that it’s going to make use of the networking libraries at a particular point on time. The qualified name (the programmer starts out with android.permission) is when the Android system sees that the Android system permission would be the one that the programmer is saying the app uses. A couple of many system permissions within Android, plus they can be extremely specified as static String members about the android.Manifest.

For that reason, and also the decision to get the lines just shown inside app’s manifest, Android permissions could be called Manifest permissions. As the Android system specifies plenty of system permissions that happen to be forced to reach certain portions in the default APIs, there’s nothing to counteract an app while using custom permissions, that app defines an innovative permission so runs achievable permission, offers an API (or methods) that requirement a calling process to possess that permission, or both. The permissions system is usually a powerful construct that should provide and extended by app developers away from core Android developers.

We’ve got seen that app should specify which permissions it entails rolling around in its manifest. The question then remains: how are these permissions granted? Which is all handled if the app is installed? At that time, its AndroidManifest.xml file is read along with the list of all permissions it requires to run is parsed. The installer then generates a directory of such permissions (not all of them, at the same time) and prompts the individual to test and approve them. It really is deemed an all-or-nothing process; either the consumer will come as much as take their email list of permissions that app claims it as well the app will install, or user can make to reject their email list in addition to the app’s install will fail. Once an app is installed, no further interaction
while using user will demand destination for a make sure they know which a clear permission has exercised and even state that this user still wish to enable the app to execute an action requiring a permission check. This design choice, to make certain that each app’s permissions with the entire user only at install time, is a key design decision for Android’s permission model.

Now's enjoyable to take into consideration where permissions are enforced. If you're writing an app that only calls other APIs and services, the programmer should pick which permissions are necessary through the APIs and services the app calls, and specify them in the manifest.

If, however, that the programmer is including methods that other apps will call, or other designs of inter-process communication endpoints, that you're sitting on the other hand available to be a provider of services; you're one who has got to grant permission. Thus, the programmer must include code within the app that checks to be certain that whichever app is calling the programmer affords the correct permissions. This is a somewhat more technical matter, so let’s explore more the permission consumer app model first. Using Restricted System APIs and also the Buyer experience When an app is strictly someone of data and offer a mechanism for other apps to have interaction straight from it, one example is providing callable methods or accessible Content providers, developer interaction while using the Android permission system is actually limited to ensuring the permissions required for the APIs in addition to system mechanisms that it app consumes are indexed by the AndroidManifest.xml file. The default reports on Android permissions are sorted into four main categories:

### 6.1.2 Normal

Permissions in this category cannot impart real trouble to any individual. As an example, the permission to enhance the system’s background wallpaper is classified as Normal category. Normal permissions are granted using a requesting app automatically upon installation along with the user has no to explicitly confirm them. However, the user affords the choice to check these permissions upon installation and definitely will review them whenever the programmer wants.
6.1.3 Dangerous

Permissions with this particular category can impart problems for an individual, by accessing private data for instance contacts, or making outgoing network connections. Dangerous permissions are explicitly shown to the user before an app is installed and the user must choose to grant the permissions or not, determining whether the installation continues or fails, respectively.

6.1.4 Signature

This group of permissions is automatically granted into a requesting app hopefully app is signed due to same digital certificate as the app that declared/created the permission. Comes to default/system permissions, since this code is signed with the device manufacturer, requesting apps not produced by the producer will forever be denied a Signature-level permission. This sort of permission is intended first off app developer to data between a lot of their apps however, not apps written by other developers, without burdening a person. Anyone just isn't prompted upon installing an app that requests Signature permissions: should the app be signed in the same certificate considering that the declaring app, the permission are going to be automatically granted; or maybe, the app being installed will never be granted the permission.

6.1.5 Signature or System

Permissions on this particular category continue with the same rule as Signature permissions, except actually automatically granted for that Android system image combined with requesting apps signed while using the very same certificate. This sort of permission can be employed by device manufacturers allowing applications manufactured by different vendors to work seamlessly together within that manufacturer’s Android builds. The effective use of this kind of permission is often discouraged, and developers writing apps for distribution through traditional mechanisms including the Android Market shouldn't have any need to ever create permissions using this type of type.
With all the varieties of permissions available, developers need to include within permission on any API that suits the factors specified above. In the event the action is sufficiently risky so any random app mustn't be performing it, it needs to require an appropriate permission. For example, one of many Normal category permissions is SET_WALLPAPER. Changing the unit’s background wallpaper isn't going to lead to any sensitive information being sent off the device or uses the user incurring any extra charges for network or other message access. However, the consumer is going to be annoyed or inconvenienced by such action should the app changes the wallpaper when it should not be. Which means capability to do so is bound using a permissions check. Opening an outgoing network connection, as we’ve seen, is at quite a different security level, so the INTERNET permission is in the Dangerous category. Along similar lines, the ability to read or send SMS messages is also protected by Dangerous permissions. In fact, there are separate READ and WRITE permissions for SMS messages—remember the Principle of Least Privilege and request only the permissions needed by the app.

Since developers put permission checks on sensitive APIs together with other strategies to reaching other apps one example is subscribing or querying Content providers, it's imperative which the programmer pick which these protected accesses the app requires selling point of, and declare the permissions inside the manifest.

However, this is simply not sure to happen, and the way an unsuccessful permissions check should expose that information towards calling app just isn't directly specified or enforced. Therefore, the strain rests user as the developer through the calling app, to find out which permissions are necessary.

Unfortunately, this may not be always a straightforward task. The Android APIs, and also other interaction mechanisms, are documented online, and plenty of in the methods that requirement a permission to successfully execute document that fact. However, don't assume all do. Indeed, as each API is answerable to building a permission checks, every API developer is likely for his very own documentation, the API documentation is simply not 100% which includes regard to permissions required. Indeed, Internet message boards and other forms of developer communications may also be loaded with queries about which permissions certain APIs require etcetera. Now, this may not be to say you're entirely the self. The API documentation, even though it does contain oversights and omissions, is a good kick off point, and plenty of developers have possibly worked through the identical
permissions-related conditions the programmer could be facing and are also prepared to share similarly info. However, will still be the burden to view which permissions the app requires in order to include those permissions inside the manifest. Also, understand that permissions might be needed for other interactions besides API method calls. As an illustration, Content providers may require specific permissions for query, insert, update, and delete operations, and Intents may need certain permissions to be either called or created.

So, there could be work involved to be certain the app will never fail since the programmer also has never are the permissions it takes in its manifest file. However, the opposite problem also exists and is also even worse. Probable disappointment to add in more permission with the manifest than the programmer actually need. This issue would go to one's heart of the Android permission model. The model exists making sure that apps run only with the permissions they really need to do their job (Principle of Least Privilege). In case the app includes permissions in their manifest who's will not actually need, that the programmer is violating that model. That might have two major repercussions, each of which parallels both the primary advantages of the permission model already discussed.

First, the permissions model exists to allow for users to own accurate info on which sensitive activities an app may perform, to make sure they could make the best decision on if they need to let the app to set up and run. The inclusion of permissions inside the manifest is an explicit statement towards user the app uses these permissions and the user is basing their decision in the claim. Probably the most significant complaints about this aspect might be more user acceptance driven than security driven.

A lot more permissions the app requests, the additionally likely a person especially a security conscious one would be to reject it at installation time. Should the programmer add some INTERNET permission inside the local-only game, for instance, an individual may legitimately wonder why such an app would request that permission and refuse to handle the installation, a perfectly sensible action the other on the key behaviors the model hopes to encourage. When the programmer presumably want as numerous users as the programmer possibly can to setup the app, it is best to require just the permissions the programmer will need without more. Moreover, minimizing the permissions the app requests may help with an increasingly secure Android culture, where users make time for it to examine all permission
requests. So it is to everybody’s benefits of use as few in the dangerous permissions as is possible.

Second, the permissions model provides some mitigation against exploitation of vulnerabilities in the app. Suppose the programmer release the app so find the programmer included an error as part of the network processing which allows an assailant to deliver some packets in the app after which it gain control over it, forcing the app to execute any instructions the attacker wants it to. This can be a frightening although much too common scenario, however the permissions model can help to eliminate the number of damage that attacker can perform. Suppose the attacker really wants to download all the contact information to a target other users with. In case the programmer wouldn't specify the READ_CONTACTS permission for ones app, the attacker cannot work with a compromise of this app to do so, since the process it is running under does not have permission gain access to the contacts list. This can be the real attractiveness of the permissions system and why the Principle of Least Privilege can be so important; mitigating the consequences of any successful attack marvelous thing.

Another element of the development lifecycle that could be affected by the permissions model has to do with automatic updates. In modern versions of Android, apps might be established to automatically update with the Android Market, enabling the programmer to get the most recent version of the app for the users quickly and efficiently. However, automatic updating works only if the update will not include any additional permission requests. One example is, should the app wouldn't require Access to the internet (the online world permission) and the programmer then release an update that does use that permission, the app will not automatically update and users will just go to the message that update can be acquired instead. Only whenever they are assigned the revolutionary permissions and consent to grant them will the modern version be installed. This functionality exists in order to avoid app developers from specifying some permission, which users accept grant, after which it grabbing more minus the user being aware of it, which will violate the Android permissions model.

Should the programmer not provide services for other apps to make use of, be that by API calls, Content Provider interactions, Activity-invoking Intents, or so on, the involvement using the Android permissions system is pretty straightforward and is
also on a the group of lines showed before with the AndroidManifest.xml file. let us now discuss programs that create these permissions.

6.1.6 Conventional Permissions

The programmer has got to produce a custom permission for anyone who is writing an app that can provide some service to others apps and the programmer also want to restrict which apps can jump on. Step one to carrying this out is always to actually declare the permission in the app which is to be enforcing it. This is accomplished inside the AndroidManifest.xml file:

```xml
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.example.testapps.test1">

    ...

    <permission android:name="com.example.testapps.test1.perm.READ_INCOMING_EMAIL"
                android:label="Read incoming email"
                android:description="Allows the app to access the email retrieved from the email server using the test1 app. Any app the programmer grant this permission to will be able to read all email processed by the test1 app."
                android:protectionLevel="dangerous"
                android:permissionGroup="android.permission-group.PERSONAL_INFO"
    />

    ...

</manifest>
```

This block of XML defines a different permission—READ_INCOMING_EMAIL—from the com.example.testapps.test1 package. This new permission continues to be given a name to be self-explanatory. With this example, the test1 app can be a basic app that implements an e-mail client, separate from the Android client built in for the standard Android distribution. The label and outline attributes of the permission
declaration provide both a quick and sweet introduction to the permission after which it an increasingly detailed description, like the ramifications of granting the permission to a new app. Each-sentence description follows a convention suggested from the Android documentation: the first sentence describes the permission plus the second highlights what an app granted the permission could well be permitted to do. From my description, it must be obvious to the user when she grants this permission to another app, all email accessed using the test1 app might be come across that app.

The protectionLevel attribute is determined to Dangerous, which can be the correct type of permission good options that any of us have experienced before. Finally, the permissionGroup attribute is set. This can be a label to sign up for the permission which allows that Android package installer to arrange the list of requested permissions if they are presented to the user upon app install. However, the new permission is of type PERSONAL_INFO, therefore it will be provided towards the user right alongside other permissions that permit having access to personal data if the list is presented on the user. PERSONAL_INFO is defined in the android. Permission-group package that offers many choices to pick from. Also the programmer can want to define a new permission Group if the programmer so choose. But considering that the goal is always to organize the permission list inclined to the consumer the best as the programmer can, it's essential practice to train on a predefined permission Group, provided that one does, in truth, sign up for the new permission.

If the programmer is defining permissions for the app, just be sure the programmers create a model that applies the Principle of Least Privilege. Look at the permission that people have just declared. This permission is termed READ_INCOMING_EMAIL, so one can assume that the only actions the app allows to the process granted this permission will involve reading messages. If it is planned to allow other apps to post messages or take them these devices, one may create separate permissions for anyone actions, each that has a descriptive label and description. Other apps looking to generate use in the email services currently have to request these permissions separately, helping them request the permissions they truly need. This is actually the proper putting on least privilege concepts and ends up with a lot better model than one that uses a detailed-encompassing permission for everyone use of email like creating just one permission to provide full control to any or all of the email services offered in the app.
Now that a new permission has been declared, apps can request it using the same uses permission tag inside the AndroidManifest.xml file when earlier system permissions were used:

```xml
<manifest xmlns:android="http://schemas.android.com/apk/res/android"

package="com.example.testapps.someotherapp">

...

<uses-permission

android:name="com.example.testapps.test1.perm.READ_INCOMING_EMAIL" />

...

</manifest>
```

This configuration will result in the some other app to request the READ_INCOMING_EMAIL permission if it's installed. Even as this permission has been declared to be Dangerous, the install process will prompt the user to confirm she wants to grant this permission to some other app.

Now that we now have declared a custom permission, and configured another app to request it, we should modify the test1 app to be certain calling apps are already granted the permission. The commonest technique of doing it is in a very method call. For instance, if the app carries a method we decide needs to be callable only by an app using the READ_INCOMING_EMAIL permission, we will achieve this the following:

```java
int canProcess =

checkCallingPermission("com.example.testapps.test1.perm.READ_INCOMING_EMAIL");

if (canProcess != PERMISSION_GRANTED)

throw new SecurityException();
```

checkCallingPermission() is successful only if another process has called the method using some selection of Android inter process communication (IPC), and only if that process was granted the permission you’re checking. Note, then, how the call will
forever return PERMISSION_DENIED should the programmer refers to it as from the method in the process.

checkCallingPermission() continues to be call you’re more than likely relating to the app considering that the primary reason choosing checking permissions is the programmer is exposing some methods allowing other apps to give them a call so the programmer want to make sure that the calling app has the correct permissions. Evaluate the above example. The app is checking for the READ_INCOMING_EMAIL permission, therefore it provides that capacity to whatever app is calling it. In order to do so, the app must already be in a position to do this. With this scenario, the programmer don’t worry whether the own app are able to do what we coded it to do; the programmer just want to make sure that whatever app called the programmer contains the READ_INCOMING_EMAIL permission before the programmer decide to proceed.

Another call can be used both by the personal process and another calling process, to test whether or not it has certain permission:

```java
int canProcess =

    checkCallingOrSelfPermission("com.example.testapps.test1.perm.READ_INCOMING_EMAIL");

    if (canProcess != PERMISSION_GRANTED)

        throw new SecurityException();
```

Here, checkCallingOrSelfPermission() returns PERMISSION_GRANTED to any caller using the specified permission, when it’s the self or another calling process. As discussed previously, this type of check may be dangerous want . process without a certain permission could call the method, which runs inside the process and therefore does have that permission, and gain access to a permission which it is not granted. This is known as permission leaking and it is a dangerous vulnerability in Android apps that deals with permission checking. Recommendation here is that the programmer should never refer to this as method.

Additionally it is possible to check whether a particular package incorporates a specific permission:
int canProcess = checkPermission(
"com.example.testapps.test1.perm.READ_INCOMING_EMAIL",
"com.example.testapps.someotherapp");

if (canProcess != PERMISSION_GRANTED)
throw new SecurityException();

Or check whether a particular process which has a specific UID incorporates a specific permission:

int canProcess = checkPermission(
"com.example.testapps.test1.perm.READ_INCOMING_EMAIL", 3445, 312);

// PID = 3445, UID = 312

if (canProcess != PERMISSION_GRANTED)
throw new SecurityException();

As one can tell, there are numerous other ways to evaluate whether a specific permission may be granted. For practical considerations, the programmer is going to should check permissions in the event the programmer offer services along with other apps and even restrict which apps can take advantage of them. In the case of an API, where another app calls the app’s methods, you’ll probably want to make use of the checkCallingPermission() method above to ensure that the calling app has the specified permission. This will likely prevent an app that does not have that permission from calling the app, which does, and gaining that permission. Be cautious about permission leakage and be sure the programmer read the correct way.

The programmatic way of checking for permissions that any of us have just discussed is quite general and powerful, but it is not the most well-liked way of enforcing permissions. The obvious way to require specific permissions is by using declarative tags within the manifest entries for the application’s components. This helps to ensure that the right permission checks are put on all strategies of the component and will make sure the programmer tend not to miss a or two, which may be common with all the programmatic approach.
That, in a nutshell, is how permissions work with Android. If the programmer need to use custom permissions, the programmer first need to generate one. Then, the programmer is able to programmatically determine whether another app that is certainly calling the code continues to be granted that permission.

### 6.2 Security and Permissions of Android Components

Android apps consist of more than one component. Each component can be secured inside a slightly different way, so before discussing on how to secure them, a simple overview of the sorts of components should be helpful.

The component that most Android developers discover first is the Activity. An action is analogous to a single screen displayed on the device to some user which is composed of the user actually sees. When building an app, the programmer creates classes that derive on the Activity base class and typically contain a number of Views, which might be objects that users can interact with somehow, including reading text displayed within the View, clicking a control button, etc. In proper application development terminology, the set of Activities is the application’s presentation layer.

An email finder service can be an Android component that is ideal for back grounding. These components can run whenever the app isn't visible (that's, none of their Activities are increasingly being displayed on the user and they are not active). Services typically either accomplish some computations or update the info sources on the app. E.g., should the programmer build an email client app, a site would run in private to open up connections to the configured mail servers regularly, check out new mail, download it, and store new messages from the database. Services can also be used to update Activities directly, to supply updates for the user directly using a Notification, or really for virtually any other kind of task that should exist in the backdrop.

A Content Provider is a component designed to share data across apps. The programmer is able to think of an Content Provider as being the public interface to the databases, allowing other apps to connect and only run queries (retrieve data) or issue updates (store data). A Content Provider typically is used like a front end for any database constructed with the Android standard SQLite database system. As Content providers are typically utilized to share data across apps, properly securing them so
that appropriate apps can access specific data is critical; we'll soon discover how to accomplish this.

Content providers are accessed using URIs with this form:

```
content://access_title/path/id
```

The authority_name is specified when one declares a Content Provider in `AndroidManifest.xml`, and points for the Content Provider itself that can handle the reference (this is usually the complete, all lowercase name in the implementing class). The path can be numerous segments, from zero on up, and is also as used by necessary, Provider to get the data in question. Inside a basic implementation, the path would simply function as name of the table the details are in. The id refers to a particular item, for instance an email message stored because of the Content Provider. Thus, a URI in a Content Provider as used by a message client app to store messages may appear to be: `content://com.my.example.test.mailprovider/messages/inbox/167`.

This URI-based technique of addressing data within Content providers is essential to properly securing the information contained within. A Broadcast Receiver is a kind of component that listens for system messages called Intents. Intent is usually thought of as a request for a certain thing. Apps can produce Intents and either send them straight away to a selected component (usually an Activity or a Service) or broadcast them system-wide to everyone apps that are running. A Broadcast Receiver is really a component that can receive these system wide broadcasts and work them; it can tend to listen for all those broadcast Intents or setup filters to ensure that it receives only Intents for the specific actions it likes the programmer (and would, presumably, get it done upon). Associated with pension transfer broadcast systems, multiple broadcast receivers can receive, and act upon, an individual Intent.

That, in summary, could be the collection of components that one could create to create an Android app. Some apps could be very simple, composed only of some Activities. Some apps require background processing and can need a Service component, or components. Apps that store and gaze after data may also use content providers to create those databases accessible to other apps. Some apps might also want to implement a Broadcast Receiver, should they wish to perform certain actions responding to requests from other apps.

For every component, care really should be taken to ensure that the component can be used which is intended, no more. The discussion ahead addresses the strategies components could be locked down and secured.
6.2.1 Use of Intents in Inter-component Signaling

Intents will be the primary means for apps to transmit messages along with other apps (which enable it to be employed to send messages with other components in the same app). An app creates an Intent object that signifies whose wants some course of action place. This Intent object can specify a unique Activity or Service that the programmer wants to start up, or it might specify a specific piece of data. Inside later case, the Intent can specify which components should perform an action on it part of data, or perhaps ask that someone should. This means that Android allows for Intents that have specific recipients the experience or Service it wishes to have interaction with in addition to Intents which have been broadcast through the entire system to any components that may be listening. The discussion ahead shows how this can be controlled, along with the relevant application security topics that arise.

One of the most common uses of Intents, and where most developers first encounter them, has one Activity start another Activity. To do so, create an Intent with all the name on the Activity the programmer needs to start then pass that Intent on the calling Context to process. One example is, with the current Activity to start out in the mainActivity, the next code may be used:

```java
Intent actIntent=new Intent(myContext,com.mytest.tapps.example.mainActivity.class);
myContext.startActivity(actIntent);
```

This code creates an Intent that specifies which Activity the programmer wishes to start with providing its full class name. This is what’s called an explicit Intent. Even as just discussed, it is possible to create an Intent it doesn't specify which class should perform the action; this is what's called an implicit Intent. Therefore, the Intent is made which has a specific action being performed and most almost daily, the piece of data that is to be put to work. E.g., if the programmer wishes to display a graphic file, so the programmer wouldn't wish to specify a specific app to do so, you'd probably create and invoke the Intent in this way:
Uri actURI = Uri.parse(pathToSpecificImageFile);

Intent actIntent = new Intent(Intent.ACTION_VIEW, actURI);

myContext.startActivity(actIntent);

After the programmer issues this implicit Intent, the Android system will determine at run time which app ought to be invoked to handle the action specified by the Intent: the viewing of the image file located for the specific URI. The question then becomes: how does Android know which components needs to be helpful to essential certain action specified in the Intent? The answer then is using Intent filters. Each Activity, Service, and Broadcast Receiver can specify several of the filters, as both versions specifies a form of action that they process. As an example, if we had an Activity with the ability to display GIF image files and wanted to expose that ability to other apps, we could set up an Intent filter on the Activity like this:

<intent-filter>

<action android:name="android.intent.action.VIEW"/>

<category android:name="android.intent.category.DEFAULT"/>

<data android:mimeType="image/gif"/>

</intent-filter>

Note that with regards to an explicit Intent, in which the element of obtain the Intent is specified featuring a full class name, there is absolutely no checking of this component’s Intent filters; an explicit Intent is obviously transported to its specified component. Thus far we’ve seen both explicit and implicit Intents. We can easily also employ the Intent mechanism to broadcast messages to any or all apps that are listening for the kids. This broadcast Intent mechanism can be used with the Android system itself to announce certain events that multiple apps might want to be aware of and reply to by incorporating action. By way of example, Android broadcasts Intent every time a call is received on the device. This broadcast capability allows various components to talk about information really decoupled manner; when something which is of potential interest to multiple apps occurs, user can broadcast a statement regarding it for any person that is certainly listening. Obviously, that is a powerful capability, where there are plenty of circumstances that the programmer may desire to
control which components (of other apps within the device) can listen to the broadcasts, letting them be observed only by those apps that should develop the rights to know about events. Let’s review broadcast Intents briefly.

The programmer could recall that people may also use something with in the email client app example running inside background and constantly download new messages from configured mail servers. In the event that the message is downloaded, we need to let the other portions the app finds out about this so the user can be notified. Indeed, it may be a good idea to let multiple components be familiar with this event, making sure that each may take their particular appropriate action in reaction for the new message’s arrival. As the client’s graphical user interface, if active, may prepare to come up with the message, another app may make a copy of it, and many others. In this instance, we are going to use a broadcast Intent to express to anyone that is listening concerning this event.

First, we should instead declare a new action string, to define the action which is being announced in the broadcast Intent. In such cases, we're going to refer to it MESSAGE_RECEIVED. To perform so, we merely define a brand new String, like so:

```java
public static final String MESSAGE_RECEIVED = "com.example.testapps.test1.action.MESSAGE_RECEIVED";
```

Then, to truly create the Intent and broadcast it through the Android system to all listening components, we issue:

```java
Intent bdctIntent = new Intent(MESSAGE_RECEIVED);
myContext.sendBroadcast(bdctIntent);
```

After we run this code, the device will broadcast the Intent to any components that has intent-filters that allow these to receive it. To configure a component to receive these broadcast Intents, we will include a simple intent-filter to the component in the manifest:

```xml
<intent-filter>
    <action android:name="com.example.testapps.test1.action.MESSAGE_RECEIVED"/>
</intent-filter>
```

We don't include any information inside the Intent, such as the URI in the new
message. We just broadcast the data which a message has been received; it can be around the constituents that obtain the broadcast to do this for this notification. So as to receive broadcasts, an app should implement a Broadcast Receiver component, a sort manufactured to take delivery of broadcast Intents and perform some action in reaction.

In light of the discussion above regarding Intents, since the primary communication path between components, both those within one app and people belonging to separate apps, we can easily begin mastering securing different components that make up Android apps. In the course of this discussion, one should always remember the Principle of Least Privilege, each component should be able to do just what it would need to liquidate order to handle its function, and nothing else.

6.2.2 Components - Public and Private

At the most fundamental higher level of access control lays the Android idea of exporting. Each component, consequently an Activity, a Content Provider, or possibly a Broadcast Receiver might be public or private. When a component is public, aspects of other apps can talk with it (start the Service, start the game, etc.). If the component is private, the only components that can get connected to it are in the same app or another app that runs with similar UID.

The exported attribute in each component’s declaration inside AndroidManifest.xml file determines whether or not the component is public or private. When exported is true, the component is public and for that reason exported with apps, otherwise the component is private not exported with other apps.

The default behavior depends upon perhaps the component may just be used externally. With regards to Activities, Services, and Broadcast Receivers, the default will depend on how the component is configured pertaining to Intents. Even as we have witnessed, an element can specify an Intent filter which allows it for Intents from other apps to undertake tasks.

As such, in case a component specifies an Intent filter, it is assumed the programmer wants the aspect of be accessed by other apps and it is, automatically, exported and so public. If, however, no Intent filter is specified for just a component, the only way to send an Intent with it is always to fully specify the component’s class name.
Therefore, it is assumed user does not want to create this component publicly accessible, and so the default exported is false and the component is private. E.g., an email finder service is commonly public automobile Intent filter is per its declaration and private if not. However, we should make a Service public but not specify an Intent filter, to ensure only components that know its full class name could make utilization of it so we may possibly like to restrict which components can get on using permissions, which we will discuss later. In this case, we have to specify the exported attribute as true, overriding the default:

```xml
<manifest xmlns:android="http://schemas.android.com/apk/res/android"

    package="com.example.testapps.test1">

    ...

    <service android:name=".MailListenerService"

        android:enabled="true"

        android:exported="true">

        <intent-filter>

        </intent-filter>

    </service>

    ...

    </manifest>
```

The default exported value for Content providers does work. Because the primary purpose of the Content Provider is to share information between apps, it is assumed why these must be public and accessible by other apps.

### 6.2.3 Restricting Access to Components

Observe that any component that is public that is exported is set to true, could be accessed by any component in a app about the device. This global access is oftentimes necessary: for example, the principle Activity with an app is generally public and unrestricted in order that the app could be started from anywhere. However, many
times, components should be capable of specify which components in other apps can access them. The discussion will now go with the different types of components and discuss how the programmer can restrict entry to them, so the Principle of Least Privilege can be applied on top of a multi-component app.

6.2.4 Secure the Activities

As discussed earlier, Activities are the presentation layer for the app. Their security and permissions are pretty straightforward and consists of that can start the Activity. To call for a certain permission to start a pursuit, it is advisable to add the permission attribute on the specific Activity’s entry in AndroidManifest.xml. E.g., to declare a pursuit that needs the commencement_ACTIVITY1 permission, the entry from the manifest would look like this:

```xml
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
	package="com.example.testapps.test1">

...

<activity android:name=".Activity1"

android.permission="com.example.testapps.test1.permission.START_ACTIVITY1">

<intent-filter>

...

</intent-filter>

</activity>

...

</manifest>
```

Appears to be app desired to start this Activity, it'd achieve this by creating an Intent specifying the will to start out the Activity after which call either Context.startActivity()or Activity.startActivityForResult(). In the event the caller has
the specified permission (START_ACTIVITY1 however), those two calls will succeed, creating the Activity being started. Without that permission, both will fail and throw a Security Exception.

6.2.5 Secure the Services

Services are the background processing components in Android. The programmer is able to restrict who are able to get connected to a site by using permissions; this may affect any attempts to interact with all the service by either creating it or binding for it. As Services typically perform a various processing that consists of updating databases, providing notifications of the external event, or performing a few other job for major benefit of a component which will communicate with anyone, it is very important to be certain that there're accessible only by appropriate consumers. To have to have a permission to produce or bind to a service, simply add the permission attribute towards specific Service’s entry in AndroidManifest.xml. For example, to declare an email finder service pots the BIND_TO_MAIL_LISTENER permission, the entry in the manifest would appear like this:
Be aware that in this particular example, the service being produced does not have an Intent filter, so that it could be invoked only by an Intent that fully specifies the course name. Also, a design is not implied where only the personal apps would hook up with this particular repair and for that reason would be aware of individual's name in the Service. Much more service doesn't need an Intent filter, we also had to specify the exported attribute as true, as this defaults to false for components that will not specify Intent filters.

If an app attempted to interact with a reverse phone lookup—starting it by calling Context.startService(), stopping it by calling Context.stopService(), or binding going without running shoes by calling Context.bindService()—the call would succeed if the caller has got the specific permission (BIND_TO_MAIL_LISTENER). Should the caller won't have this permission, the email would fail using a thrown Security Exception.
This capability controls who is able to talk with an email finder service only in a very coarse level the programmer can control what can start, stop, and bind to some Service). The programmer cannot restrict access permissions on the more fine-grain level by using mechanism, for example providing permission checks on specific callable strategies of a bound service.

One can call checkCallingPermission() to view regardless of if the calling app has a specific permission in the access point of any methods the programmer would imagine require permission checks past the permission configuration set for the overall Service. Of course, the programmer can create a real check at any point in the method too; this before performing a sensitive operation is the key.

### 6.2.6 Secure the Content Providers

Content providers are the standard opportunity for apps to produce their data accessible to other apps. Because the components exist to share data among different consumers and suppliers, they raise the requirement for a far more complex security model. Unlike Activities and Services, Content providers can specify two different required permissions: one for reading and another for writing. This will give apps to be configured using the Principle of Least Privilege and recognizes how common designs are where some apps should be able to read certain data, other apps can write that data, but still others mustn't be allowed to access the information in any respect.

One common misconception about these permissions is the fact that obtaining the write permission automatically implies the read permission. The logic is updating data (writing) is really a more powerful permission than simply reading it and anyone who can write right into a database should likewise have the ability to read from using it. This is the fallacy, and also the Android design, separating read permissions, acknowledges that. As an example, consider an email client app. In the app, one can implement a Content Provider to store email downloaded at a mail server. An email finder service may periodically connect to that mail server, download new messages, and put them into the database by contacting that Content Provider. The Service clearly needs the permission to create for the Content Provider, but can it require the permission to read as a result? No. Actually, it shouldn't have that permission, since it
is unnecessary for it for the position it performs. Remember the Principle of Least Privilege: grant the permissions necessary to complete the same task no more.

For example, to declare a Content Provider that will require separate permissions to read or write data, the entry from the manifest would look like this:

```xml
<manifest xmlns:android="http://schemas.android.com/apk/res/android"

package="com.example.testapps.test1">

...

<provider android:name="com.example.testapps.test1.MailProvider"

android.authorities="com.example.testapps.test1.mailprovider"

android:readPermission="com.example.testapps.test1.permission=DB_READ"

android:writePermission="com.example.testapps.test1.permission=DB_WRITE">

</provider>

...

</manifest>
```

With Content providers, permissions are first verified after the programmer connects with the provider. If the programmer achieves this, you'll be able to hook up with it when the programmer has either the read permission or the write permission. When the programmer has neither, the link request will fail using a thrown Security Exception. Permissions are then checked once the programmer tries and gets connected to this link Provider. Featuring its query() method requires the read permission, with all the its insert(), update(), or delete()methods necessitates the write permission, as per the manifest declaration (DB_READand DB_WRITEin this example). Again, realize that keeping the write permission doesn't automatically provide the programmer with the read permission; the two are separate and enforced independently. Should the programmer call one of these brilliant methods without worrying about corresponding permission, the call won't succeed along with a Security Exception would be thrown instead.
The chance to separate read and write permissions allows us to better control which apps can communicate with the Content providers and just how they will accomplish that. However, when permissions are enforced using these methods, they apply at each of the data in just a given provider, which may sometimes be a significant amount of. Let’s look at the email client app again. Only GUI components of the app tend to allowed accessing the root Content Provider that holds each of the email; and certainly do not want other arbitrary apps in order to achieve this, as email is sensitive information. However, how about the situation of attachments to email? Somebody who is image or sound file attached with a message, the e-mail client app has to be able to invoke the proper media handler to properly present that attachment on the user, and this would require the media handling app to own read permissions in the Content Provider that this attachment is kept in. That is clearly not a good idea, even as we could be granting these apps the flexibility to read every one of the stored email.

The reply to this problem is always to utilize URI permissions. The Android designers have recognized that developers ought to grant permissions to particular portions of a Content Provider’s database for a while of the energy. The situation of allowing a media app entry to an attachment from the Content Provider in the event the user wishes to display that attachment is an illustrative illustration of this indeed. With this mechanism, any component that has permission to access a URI within a Content Provider can temporarily grant that usage of another app/component. With regards to the email client app, the person interface app which has permission towards entire Content Provider with email and attachment content could temporarily grant permission towards the URI which contains an attachment towards the image viewer that may be being asked to show off the picture, even though the image viewer doesn't have a permissions to access the Content Provider by itself.

To operate the URI permission system, a Content Provider has to be configured to allow this inside the Android manifest. Because system is needed to give permissions beyond what is normally configured, the mechanism is turned off automatically. The programmer will discover two solutions to enable URI permissions: one that covers any URI contained within the Content Provider and another that covers merely a subset of URIs. As a way to enable temporary URI permissions globally with a Content Provider, set the grant UriPermissions attribute within the <provider>declaration to true, like this:
Realize that this configuration attribute won't actually grant those permissions simply by including this approach; it merely allows an element which includes the access already to temporarily grant those permissions later to a different app. This global configuration option allows the Content Provider to grant permissions to any accessible URI within it. It really is entirely possible for the Content Provider to become interested only in granting temporary permissions with a subset of URIs. In this email client app example, we envisioned by using mechanism to temporarily grant media-handling apps entry to attachments in the email database. It's wise to store attachments in the different path from the database, separate from messages. By way of example, the URI to a mail message may look like this:

```
content://com.test.tapps.example.mailprovider/messages/inbox/167
```

Along with the URI with an attachment may look like this:

```
content://com.test.tapps.example.mailprovider/attachments/97
```

In cases like this, we may want to temporarily grant permissions to entries inside the /attachments/path, but never grant permissions for the /messages/path. In this case, globally enabling URI permissions would violate the Principle of Least Privilege. So that the programmer can deal with this, we can easily omit the grantUriPermissions
attribute and instead include separate <grantUriPermission> elements inside <provider> element. Each <grantUriPermission> element specifies a subset from the Content Provider that it can grant URI permissions. For each such element, the programmer includes one among three possible attributes: path, path Pattern, or path Prefix. To specify a directory whose files could be accessed through URI permissions, utilize the path attribute within the <grant-uri-permission> element. One example is, to allow URI permissions for being granted for the /attachments/path, the appropriate configuration would appear to be this:

```xml
<manifest xmlns:android="http://schemas.android.com/apk/res/android"

    package="com.example.testapps.test1">

    ...

<provider android:name="com.example.testapps.test1.MailProvider"

    android.authorities="com.example.testapps.test1.mailprovider"

    android.readPermission="com.example.testapps.test1.permission.DB_READ"

    android.writePermission="com.example.testapps.test1.permission.DB_WRITE">

    <grant-uri-permission android:path="/attachments/"/>

</provider>

...

</manifest>
```

The path attribute is just not recursive; when there is any subdirectory under /attachments/, no access is granted to files in that subdirectory. To specify a path that is a prefix (meaning that most URIs that fit in that path may be granted URI permissions), use the path Prefix attribute inside the <grant-uri-permission> element. As an example, if we needed to be capable to grant URI permissions on messages in the database any path that began with /messages/, we will configure this content Provider such as this:
Finally, the path Pattern attribute can be used to supply the whole path just like the path attribute does, but supports wildcards. Wildcards on this kind of specification follow the format of utilizing an asterisk (*) to fit zero or higher occurrences with the character that immediately precedes it. So to match zero or higher on the letter A, one would utilize the sequence A*. It's also possible to use a period (.) to enhance any single character. To complement zero or maybe more characters (any characters), you'd utilize both-character .*sequence. E.g., in the event the programmer seriously considered able to grant permissions to any URIs that contain the string public in them, write:
So we now identify that Content providers might be configured so apps that currently have use of them can temporarily grant those permissions to apps. In the event the Content Provider has been configured in this manner, the app needs to actually grant those permissions. This is easier than the programmer think, as it consists solely of setting the appropriate flag inside the Intent manufactured to call the newest app. To temporarily grant read permissions on the app being called, set the FLAG_GRANT_READ_URI_PERMISSION flag. To temporarily grant write permissions towards the app being called, set the FLAG_GRANT_WRITE_URI_PERMISSION flag. In the case of the own email client app, an Intent would be created that specifies the URI of the attachment that any of us wish to open. Next the FLAG_GRANT_READ_URI_PERMISSION flag will be attack the Intent. When this Intent is passed to Context.startActivity(), the image viewer Activity could well be started, targeting the URI inside the Content Provider. As the appropriate flag was set, the look viewer app can have permission to access the specific URI inside the Content Provider and all of would work as intended. For example:

```xml
<manifest xmlns:android="http://schemas.android.com/apk/res/android"

package="com.example.testapps.test1">

...

<provider android:name="com.example.testapps.test1.MailProvider"
android.authorities="com.example.testapps.test1.mailprovider"
android.readPermission="com.example.testapps.test1.permission.DB_READ"
android.writePermission="com.example.testapps.test1.permission.DB_WRITE">

<grant-uri-permission android:pathPattern=".*public.*"/>

</provider>

...

</manifest>
```
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);

An alternative way of specifying the flags within the Intent should be to explicitly grant a certain app permissions over a Content Provider URI. Realize that this isn't the preferred approach to pass permissions; the flags specified in the Intent will be the standard approach. But to accomplish the other approach in a program, the programmer can use the Context.grantUriPermission() method, passing within the package name this agreement the programmer needs to temporarily grant the permissions, the URI under consideration, and also the Intent flags specifying which permission(s) the programmer intends to grant. E.g., to grant another app read permissions gain access to a message within the email client's database:

```java
uri = "content://com.example.testapps.test1.mailprovider/attachments/42";
Context.grantUriPermission("com.example.testapps.test2", uri,
Intent.FLAG_GRANT_READ_URI_PERMISSION);"
possible to programmatically verify whether a specific app (if the programmer realize its process and UID) may be granted use of a certain Content Provider URI. Remember that this mechanism only checks to see whether the specified process has been granted permission using one of many two mechanisms described within this section and doesn't consider broader, general access. Therefore the programmer needs to count on this process only to check temporary URI permissions. To check on to discover whether another app gets the specific permissions, call checkUriPermission() and pass within the URI, the PID the programmer would like to check, the UID the programmer wishes to check, along with the flags the programmer wishes to verify. For instance:

```java
int canProcess = checkUriPermission(uri, 377, 122,
Intent.FLAG_GRANT_READ_URI_PERMISSION);

if (canProcess != PERMISSION_DENIED)
...
else

    throw new SecurityException();
```

This code will verify if the process running with PID 377 and UID 122 may be granted read entry to the URI specified. If the programmer is handling an IPC for another app, use a different procedure for checking whether that calling app continues to be granted the specified permissions. Realize that this method works as long as that the programmer is processing an IPC from another app; in case the programmer is not, it will always return PERMISSION_DENIED:
Understanding that’s the term on Content providers. These ingredients require a longer discussion compared to the other components because they are the most tricky to secure. However, if the programmer understands the final concepts, it’s definitely not a lot of work left. Content providers use standard Android permissions permitting apps read and/or write access to them. Many situations require more fine-grained access, so URI permissions are for sale for apps to temporarily grant permissions to a particular content in a Content Provider when necessary. As soon as the programmer discovers how the URI permissions work, it's possible to grant specific apps the precise permissions they have to specific content at the particular times they want them. This can be a Principle of Least Privilege at its finest.

### 6.2.7 Secure the Broadcast Intents

Messages may be broadcast out to any app that is listening for him or her using Broadcast Receivers. As discussed earlier the email client app example and how the Service that is constantly checking for brand new mail may decide to send a broadcast Intent whenever a new message have been received, to ensure that multiple components might want to do something about this. In this case, one possibly needs to limit the components that will receive a really broadcast, as we are afraid to visit announcing towards the whole world that an email message just can be found in.

The sender of broadcasts can choose to apply an Android permission to each broadcast it sends that broadcast will be delivered only to those Broadcast Receivers that both have an Intent filter that allows them to receive it and the specified permissions that indicate they are authorized to do so. When it comes to the Service...
example, we could restrict which Broadcast Receivers may receive the broadcasts by sending the broadcast merely to people that have a MSG_NOTIFY_RECEIVE permission that we create due to this purpose:

```
Intent bdctIntent = new Intent(MESSAGE_RECEIVED);
myContext.sendBroadcast(bdctIntent,
"com.example.testapps.test1.permission.MSG_NOTIFY_RECEIVE");
```

Observe that most of the time, when a permission check fails, a Security Exception is thrown. If we lock down broadcasts in doing this, no Security Exception will be thrown if a Broadcast Receiver specifies that they should receive these broadcasts nonetheless they don't have the specified permissions. Indeed, since this code attempts to send the desired broadcast Intent to any Broadcast Receiver which has a matching Intent filter, most of these receivers could have the specified permission plus some may not; no feedback is returned for the component sending the broadcast Intent concerning which succeeded and which failed.

This mechanism enables the sender of a broadcast to specify which receivers are permitted to receive it. It is usually possible to do overturn: to configure a Broadcast Receiver to accept incoming broadcast Intents only from senders that retain the specified permissions. To do this, simply specify a permission attribute within the <receiver>element in AndroidManifest.xml. For example:
This declares a Broadcast Receiver that listens for MESSAGE_RECEIVED broadcast Intents and accepts them only from senders which have been granted the MSG_NOTIFY_SEND permission. If the MESSAGE_RECEIVED broadcast Intent arrives from a sender without that permission, it will not be transported to this Broadcast Receiver. It is usually possible to register a Broadcast Receiver programmatically, as opposed to from the AndroidManifest.xml file, by calling registerReceiver(). In this instance, you'll be able to still apply a permission restriction, only allowing sender's achievable permission for the programmer to the registering Broadcast Receiver. For example:

```java
IntentFilter intentFilter = new IntentFilter(MESSAGE_RECEIVED);

UIMailBroadcastReceiver rcv = new UIMailBroadcastReceiver();

myContext.registerReceiver(rcv, intentFilter,

"com.example.testapps.test1.permission.MSG_NOTIFY_SEND", null);
```
As the programmer can see, broadcasts may be secured in a choice of direction. Senders of broadcasts can configure them making sure that only receivers with specific permissions are allowed to receive them. Receivers of broadcasts might be configured to simply accept them only from senders with specific permissions. By using a mix of both of these mechanisms, you'll be able to create a broadcast architecture where each component that needs to be notified about certain events is notified and those which should stop notified about options are not, every component accepts event notifications only from authorized senders. Putting It All Together: Securing Communications in a Multi Tier App

The Android permission system allows the different components that comprise an app to line restrictions on the other components can access them. Even though basic apps are composed of a few Activities, larger plus more sophisticated apps are typically composed of a mixture of Activities, Services, Content providers, and Broadcast Receivers. In addition, Android encourages the application of decoupled components through the Intent system; where multiple components owned by different apps can work together in order to meet a desired function. This method of app design, thinking when it comes to components as opposed to apps, could lead to several advantages, but not also end in many apps receiving usage of data and information they shouldn't have. Consequently, a proper security architecture should be section of any involved app design. Like a developer, the programmer has to consider what type of data and the type of service the components are providing and who really should be granted having access to them. Upon having determined that, the actual application of a permissions scheme to implement its very straightforward. For Activities, the programmer should carefully consider who are able to invoke/start the Activity. For the initial Activity which will be displayed to the user once the app starts, no permissions are needed. For other items, apply permissions so the Activities in question may be started only by others that contain the correct permissions. For Services, the programmer needs to consider who will be able to start them, stop them, and bind for many years. Because Services run in the shadows, it's likely they're providing some service that other aspects of the app will access. If they're doing anything with sensitive data or performing any functions that will’t be available to any app on the globe, permissions really should be applied making sure that only apps that should have access for their functions are permitted. In addition, if the Service supports binding and allows apps for making calls to methods for functions it could
possibly provide, the programmer may want to add further programmatic checks within some of those methods, if the programmer will find different amounts of sensitivity. E.g., inside email client app, certain permission could be necessary to reach a method that checks when the latest email message was received, while a separate permission may be required to update the mail server settings. In this case, restricting different methods with various permissions is clearly necessary.

For Content providers, consideration has to be consumed in designing their permissions. Read and write permissions can be set on Content providers and any components which need access to the provider really should be granted only the permissions that they truly should perform their jobs. Content providers also include the opportunity to grant URI permissions—whereby a certified component may temporarily grant permissions to another component that does not have any rights itself—to specific components of data inside Content Provider. This enables an exceptionally fine-grain permission model being set up and enforced, but requires careful planning. The grant-uri-permission configurations need to be placed in the manifest. The programmer has to get these configurations right so that permissions is not temporarily granted to data which will not be accessed by components that won't carry the permissions specified to the Content Provider.

It is strongly recommended that the programmer employs individual grant-uri-permission entries as opposed to the overall grant-uri-permissions attribute for the entire Content Provider, if the data inside the provider might be segregated into sections that could be safely shared and that can't. Finally, once permissions are temporarily granted, they need to be revoked when the necessary actions are carried out, to ensure permission is granted only back then period it can be needed for.

Broadcasts certainly are a key way of inter component communications. Mechanisms exist so the senders of broadcasts can require any receivers that are looking for to get these phones have certain permissions, and this needs to be enforced should the broadcast is at all sensitive. The contrary mechanism, allowing Broadcast Receivers to accept broadcasts only from those senders with specific permissions, allows receivers to accept messages only from those senders that they'll trust since they have been granted the permission.

When utilizing broadcasts to speak details about anything sensitive, both mechanisms ought to be deployed, authorizing both receivers and senders, and barring unauthorized aspects of the system altogether. Individually, each component type
allows the programmer to specify whom the component should be allowed to speak to. Put together, these mechanisms allow an Android app composed of multiple pieces to completely control how those pieces interact, both along with other aspects of the same app and those which are section of different apps. Once the architecture associated with an app has been designed, applying the principles we've got just discussed should be a fairly straightforward task and may result in communications that correctly employ the Principle of Least Privilege as well as a robust, rugged, safer app.

6.3 Stored Data Protection

Now let us turn our attention to how to protect data that the application stores around the Android device it is run on. This could include basic, non-sensitive data such as high scores on the game, or very sensitive data such as user’s banking account login details, when the app require usage of such data. Some of the built-in protections implemented by the app due to the basic architecture of Android should prevent other apps from accessing (or, potentially even worse, modifying) this stored data. But basic access control, including that furnished by the operating system, is normally not enough to shield from the risk of compromise. Remember, if the risk, constructed from vulnerability, threat, and consequences, is high, the programmer will need to mitigate that risk down to a sufficiently reduced level). This is when cryptography comes into play; if results are stored within a scrambled or encrypted form, an attacker will get on the job without recovering the conventional, unencrypted form.

6.3.1 Dangers and Weaknesses for Stored Data

Protections needed for stored data or data resting, count on the mechanisms of the running Android system. They assume that the system is up and running as it should which all accesses to stored data will be through these controlled channels. Unfortunately, this is just not necessarily the truth.
6.3.2 Weaknesses of Stored Data

All data which is stored on the typical Android device gets kept in some number of mass memory device. Most devices include some storage internal on the device (included on the primary plug-in from the case) along with a typically much larger level of storage on a removable storage device. No special protections are allowed to data compiled by an app onto this storage, and accessing from the computer is fairly straightforward (more so while using storage device, which could simply be removed of the device and inserted into a pc, but gaining access to the memory internal with a device isn't a lot more difficult).

If the app stored some sensitive data, say banking accounts numbers or passwords for social websites sites, that data might be easily read by an opponent who happened to swipe someone’s device, or even just its memory card. Offline entry to the storage while on an Android device is straightforward to obtain and affords an assailant full having access to whatever the app stores.

Furthermore, any data stored on the memory card seriously isn't given any isolation because of the underlying Linux kernel which enables it to be freely accessed by any application while still from the device. This becomes an important concern whenever the programmer understands that not all developers understand this and store data on the media card without hesitation. It is usually possible to obtain root-level entry to Android devices.

As root could be the super user identity, the programmer isn't root access are able to do anything they need with a system. Regarding Android, Google has created simple to use for somebody in possession of an entry to a tool to realize root usage of it. This allows complete and unrestricted entry to the memory and storage while device is running. The isolation provided to files, databases, along with resources simply does not exist when other apps can run with all the root identity.

The threat caused by root usage of a tool the application is running on goes past those of an individual performing this process themselves. Some in the heightened malware targeting Android devices employ kernel-level vulnerabilities for getting root themselves. This will give the malicious code to run as root, in a position to bypass
the Android security model. This is the prime illustration of where reliance on that model to protect sensitive data may not suffice.

6.3.3 Reducing Exposures of Stored Data

Thinking back to the previous risk discussion, one has to find the threats to the data stored by the apps along with the consequences if this were to be successfully attacked. For instance, suppose the programmer were to create a casino game that stored information such as high scores, player progress, along with such details. The risk of someone having a vulnerable ability within the code to learn to read such details are probably pretty low, since there are no real consequences because info is not sensitive. However, the risk of somebody using this kind of vulnerability to modify such data, perhaps giving themselves a new high score or advancing past part of the game illicitly, might be a the upper chances, particularly when the user can find rewards offered for excellent play. What elements of the danger equation alter in this case?

Consider another example of the social media aggregator, here, the chance of someone utilizing a vulnerability to read stored data including account passwords is probably pretty high, while risk someone modifying such data or deleting it is probably pretty low. Will be worst thing that someone could do if they can affect the stored account information an app purposes of the Twitter access? The answer could make the programmer think about this as a higher risk than the programmer may first think!

Remember, the steps you're taking to defend the info the app stores about the device must be appropriate given the threat. Should the app stores no sensitive data, along with the consequences of their data being either confronted with others, modified, or deleted is minimal, then the programmer definitely does not need to stress about items like applying proper permissions in the data store or encrypting the computer data. If, however, the data that the programmer is storing is incredibly sensitive, and compromise of their data would produce serious consequences, you'll need to take more steps to defend it.
6.4 Protection Mechanisms

Protection of stored data, or data resting, is better accomplished having a series of protection mechanisms that reinforce one another. By way of example, if proper permissions are set within the data store the app is utilizing, other apps aren't going to be capable to read or modify that data on the normally running Android system. Sensitive data should likewise be encrypted, to ensure if an attacker can compromise the protections offered by the Android system and find access to the results, they can not a single thing with it. Deploying different layers this way is a kind of information security practice known as Defense in Depth (DiD).

The steps delivered to protect stored data have to be appropriate based about the risk of compromise. Along similar lines, before determining the best way to protect the data stored from the app, consider carefully what data the programmer actually ought to store. Do the programmers really need to store bank account details? Would the programmer like to store user account credentials? This goes returning to the Principle of Least Privilege: apps should store the data was required to manage to get their job done effectively and nothing more.

It is an extremely important point. Carefully consider what data the programmer needs to store to effectively accomplish its purpose. Store that data and zilch else. The less data the programmer store, the less the programmer should protect.

Think about the data that the programmer try have to store? Modern cryptography has provided us with many tools and techniques that can be put on protect stored data. In order to understand how the programmer can look after data the apps store, one should instead understand the fundamentals of cryptography.

6.4.1 Cryptography Starter: Encryption

When people hear the saying cryptography, they have a tendency to consider encryption. Encryption, basically, refers to the process of having a message and transforming it in a scrambled version of itself. The theory goes that may be it safe to achieve the cipher text compromised, for attacker won't be in a position to deduce the plaintext from exactly the scrambled form.
6.4.2 Symmetric Encryption

To be a basic example, consider the Caesar Cipher, named after Julius Caesar, who is said to have invented it. If the programmer wishes to transmit a communication on the army, but they are fearful of the message being captured and look by the enemies on the way, the programmer could have each letter inside the message and instead of writing that letter, write the letter that's x letters far from it. As an example, let's supposed that Caesar and the generals thought that he would use an x value of 6. When Caesar planned to send some text out, rather than writing an A, although write the letter that's six (as x=6) letters away from A, that's G. Instead of writing a B, he would write an H. Etc. Once the generals received this message, they will subtract six from each letter and recover the true message. Anyone intercepting the message could be not able to recover its true meaning, when they wouldn't normally determine what x is.

On this example, the actual message will be the plaintext. The shifted version with the message is the cipher text. The algorithm accustomed to encrypt the message is actually a shift cipher, as each character is now use another value, which is x away on the true value. The importance of x is known as the key. Look this encryption system, one can possibly change from the plaintext for the cipher text (encryption) if an individual knows the algorithm used as well as the key.

The programmer can rewind through the cipher text for the plaintext (decryption) if an individual knows the algorithm along with the key. The Caesar Cipher is a classic and very basic illustration showing a symmetric key encryption algorithm, for the reason that same key is employed for both encryption and decryption.

The Caesar Cipher is quite illustrative to how these symmetric key algorithms work. Even if this system can be quite old and simple to sneak nowadays, modern systems work along similar principles, the place where a plaintext message is sent in an encryption algorithm along with a key, as well as the cipher text equates another end.

Essentially the most famous symmetric algorithm known today is probably the Data Encryption Standard (DES), which has been utilized for decades to shield information. It's got largely been replaced with the Advanced Encryption Standard (AES), which is much newer and stronger. Today, AES will be the standard for symmetric key encryption.
Symmetric encryption algorithms, AES and others, have similar principles. First, the algorithm is generally public and not a secret. The key, alternatively, is very secret and need to be protected. Anyone can recover an original message if they know the important thing used to encrypt some text. Second, the effectiveness of the symmetric encryption is often in line with the entire key used; the longer the important thing, the stronger the protection proposed by the encryption. Since 2011, the present standard key size is 256 bits (AES in combination with this size secret's called AES-256).
Third, symmetric encryption algorithms are typically very fast; modern computers can encrypt and decrypt with them in a short time.
These algorithms could be regarded as a box the place the programmer put in some text, locked by way of a key, and an encrypted message is released in the event the key's helpful to unlock it. The process is reversible; the same key accustomed to encrypt the material can decrypt the encrypted version. The longer the real key the programmer utilize, the harder it can be to get better the main message through the encrypted one by brute force. Assuming the programmer employ a fantastic size key and a good algorithm, he'll be able to store the encrypted message around the worse enemies all that's necessary and they're going to not be able to recoup the original message. Just protect that key!
Symmetric encryption is a wonderful thing. However, there is one primary flaw by using it. Think to the Caesar Cipher. If Caesar needed to use his algorithm to communicate securely in reference to his generals, although have to be sure that his generals in the field had identical knowledge of x that he did. Otherwise, they might struggle to decrypt his messages. Caesar cannot just send the true secret out by using a message because his messages are being compromised, hence the main need for this scheme. This can be a primary flaw with symmetric encryption; it utilizes all communicating parties getting the same key before any messages are exchanged.

6.4.3 Asymmetric Key Encryption

Symmetric key encryption carries that name because the identical key's used by both encryption and decryption. Asymmetric key encryption uses two different keys: one for encryption then one for decryption. The keys have a special mathematical relationship which allows messages encrypted with one critical for be decrypted from
the other. As useful use such keys revolves around their use with one another, they are considered to be key pairs.

Within a typical asymmetric cryptography application, each communicating party carries a key pair. It makes one key recognized by everyone in the world; this is what’s called everyone key. It props up other key, the private key, to itself and don’t divulges it. If a person would like to send some text to someone, they could protect the message by applying an asymmetrical encryption algorithm going without running shoes, while using the recipient’s public key. In the event the intended person receives what it's all about, the guy can then use his private step to decrypt the message remember, one key can decrypt messages encrypted which consists of partner key, so the private key can decrypt messages encrypted which consists of public key. The 2 parties would not have to exchange keys using some protected method before using encryption.

Asymmetric encryption algorithms are less plentiful than symmetric ones. Essentially the most common is called RSA, with each letter standing for the first letter in the last name of the three inventors. The algorithm itself is publicly known but not a secret. Each user’s public key is also public knowledge and anyone can be aware of it, but each user’s private keys have to be only accessible by that user. The effectiveness of the asymmetric encryption, like symmetric encryption, is often using the entire key used.

However, the important thing lengths are not comparable between asymmetric and symmetric algorithms; RSA keys usually are considered these days, to be least 2048 bits long to offer protection just like that surrounding AES-256. Also, asymmetric algorithms tend to be very slow, especially when compared with symmetric ones, therefore the data being encrypted or decrypted ought to be minimized.

Summing up, asymmetric encryption involves complementary encryption and decryption processes. Encryption requires the message and one person in the important thing pair. The process is reversible, but decryption necessitates the other member of the key pair. Assuming the programmer uses a superb key size plus a good algorithm, it is possible to encrypt some text using someone’s public key and only see the face would be able to make out the print, no prior key sharing required.

Asymmetric key encryption has other uses, too. Rather than encrypting a communication with other people’s public key (that allows only her to see it, as only she gets the private key), let’s say the programmer encrypted the message with the
own personal private key? What benefit would that contain? Well, say the programmer is writing an email and wants to prove that he is the owner of it. The programmer might release the two messages along with the encrypted kind of it concurrently. Anyone can grab the public key (as it's public) and decrypt the encrypted form. If the published message equates, the programmer will need to function as the author of the message. This scheme provides the basics for the purpose is actually a digital signature, but that needs a little more explanation first.

6.4.4 Cryptography starter: Hashing

Encryption, whether it is symmetric or asymmetric, is a good thing. However, it isn't the end all-be-all of cryptography. Another essential concept that the programmer can know about is a hashing algorithm. Should the programmer has studied data structures before, maybe the programmer is informed about hash tables and other alike organizational strategies. The true secret to presenting hash tables efficiently would be the hashing function, and that is put on a bit of data and creates a result currently in use to view which of x hash buckets the results is to be stored. The idea is that if the programmer will find x buckets, a fantastic hash function will produce each result of 0 to x in equal frequency. A cryptographic hash is very similar, but this trait is taken further.

A cryptographic hash function takes a message of any length and creates a fixed-size hash termed as digest. The digest appears to be a random sequence of bits. For example, SHA-256 can be a popular hashing function that produces 256-bit digests. The programmer can put a message of any size, from a few bytes to multiple megabytes, in to a SHA-256 function and the resulting digest value will always be 256 bits. The digest functions as a digital fingerprint on the message. Cryptographic hash computation is additionally very quick, roughly exactly the same speed as symmetric cryptography.

All cryptographic hash functions have four main properties:

- If two digests are wide and varied, the two messages that produced options are different.
- If two digests are similar, it can be highly likely that these two messages is the same.
• In case a digest is computed over a message and also a single little bit of the material is modified, the digest produced are going to be contrasting.
• It can be computationally infeasible to get a message that may spark a specific digest.

Hash functions work with large inputs and produce small outputs. Therefore, there are a numerous messages that could produce a similar digest. When two different messages produce the same digest, this is known as being a hash collision. An excellent hash algorithm will minimize how many collisions. Notice also that hash functions are certainly not reversible as encryption functions are. They are classified as one-way functions, as possible go one of the ways, however, user cannot rewind.

Precisely what could we utilize a cryptographic hash algorithm for? Let’s take a look at two examples. One common one is to store credentials. Every time a user supplies a password, something must be in a position to see whether the password is correct. One way this is done is usually to encrypt the users’ passwords and decrypt the password of a specific user if they seek to log in. Then, we can easily compare the password they furnished with the decrypted value to see should they match. However, this actually is a very bad idea. To encrypt then decrypt user passwords, we'd must store and use an integral in this instance, a symmetric key. If that key were to be compromised, all with the passwords for the users may very well be retrieved—an incredibly bad outcome. Instead, passwords or other credentials should be hashed and also the digest stored. When a user tries to authenticate, we hash the credential they supplied and compare it to the stored value; whenever they match, the consumer needs to have supplied the best credential. Remember our previous discussion about whether we need to store sensitive data: hashes pass possible not to store actual passwords.

Storing hashed passwords is more technical than hashing the password and storing that value. Modern implementations use functions that employ multiple rounds of hashing as well as the utilization of an additional protection mechanism termed as a salt. Another common utilization of cryptographic hash algorithms is usually to verify data integrity. In case user desires to verify a bit of data hasn't been altered, the programmer possibly can compute the hash of that data if it's stored after which implement it again when it's accessed. In the event the hashes match, the information hasn't been altered. This application is why hashes are now and again called a cryptographic checksum.
Finally, let’s revisit the example we discussed above. We said that if a user wished to prove he wrote a message, he could encrypt the message together with his own private key and release that cipher text with the original message. Anyone could then use that user’s public answer to decrypt the cipher text and if it matched the initial message, we may realize that anybody did, in fact, write the message. However, remember the properties of asymmetric encryption; it could be very slow to complete along with the message involved can be quite long. To ameliorate this, apply hashing. As opposed to encrypting the full message in reference to his private key, the author could hash what it’s all about and encrypt just the hash. Because digests are incredibly small, the encryption really should be in no time, as will the decryption required to verify the author’s claims. This scheme of releasing an encrypted hash of any message to prove which the programmer wrote it is actually a digital signature.

**6.4.5 Computational Impossibilities**

We have now used the word computationally infeasible a lot. Because of this the quantity of computing power needed to perform this kind of task is so great to produce this effort unlikely. As computing power increases, issues that were infeasible decades ago become feasible. This is why cryptographic algorithms will almost always be being updated, and key sizes tend to range in price up as time progresses. DES was the key algorithm before AES plus it stood strong for quite a while. However, DES was limited to a 56-bit key. As computers got faster, a brute-force attack from this small key size became possible. Here, computers would simply try every possible key (256 of which) until they found the one which would properly decrypt the message. AES, however, will use different key sizes and 256-bit keys usually are standard, these days. As computers keep advancing, this really is prone to increase in the near future.

Cryptographic hash algorithms have similar traits. MD5 was the hash function that saw widespread use ahead of the SHA group of hashes came around, and yes it creates a 128-bit digest. The SHA family can produce hashes of sizes and, again, a 256-bit digest is recognized as strong right now, despite the fact that 512-bit hashes are fast becoming commonplace.
6.4.6 Algorithm Selection

When scouting for the algorithms and key sizes to make use of, the programmer ought to use well-known, well studied industry standard options. For symmetric encryption, AES with 256-bit keys could be the current standard. For asymmetric encryption, RSA with 2048-bit keys is the current standard. For hashing, SHA-256 will be the current standard. Its predecessor, SHA-1, remains in widespread use although it shouldn’t be, and SHA-512 has grown a lot more common. After the programmer selects which algorithms and key length to utilize, find and use a rightly-known and well-tested implementation. Do not write the personal implementation of cryptographic algorithms, the Android system gives a perfectly capable implementation of major cryptographic algorithms, which has been extensively tested and validated, so just use those.

Keeping them right is very tricky and small, subtle flaws can produce output that looks enjoy it is working but offers protection that is nowhere close to strong as that made available from a suitable implementation. Among this can be the CSS encryption scheme deployed on DVDs; flaws for the reason that custom algorithm reduce its effectiveness considerably.

It is very important remember that different capabilities, including different cryptographic implementations, are included with Android every time. Different versions of Android will include different support for a number of cryptographic algorithms, and a few versions may remove support for outdated algorithms because they become obsolete. This is the one other answer why the programmer has to test the application on multiple Android versions and multiple Android devices.

6.4.7 Cipher Operation Modes, Initialization Vectors

Once we have discussed, encryption algorithms focus on blocks of web data (AES, for example, uses 128-bit blocks). One strategy to encrypting a tremendous level of data would be to simply encrypt each block independently. We have an issue with this method, however, as all blocks that contained identical data would encrypt towards the same return and an attacker that may understand the cipher text knows that two blocks contained the same plaintext data. This might be bad, because it allows the
attacker to learn something around the plaintext. To face this issue, encryption algorithms are generally performed in sequences, or modes, that produce each block influenced by the prior block. One example is, in Cipher Block Chaining (CBC) mode, each plaintext block is XORed with the previous cipher text block after which encrypted. This makes certain that plain text blocks sticking with the same contents encrypt to different cipher text blocks. Note that we need something to XOR the very first block with to produce this consistent. This value is generally known as an initialization vector (IV). It ought to be random, but it really doesn't have to become secret, as being the key does.

Along similar lines to IVs, applications of hashing algorithms typically involve use of the salt. This is the value which is with the data being hashed, usually just concatenated from it, to generate cracking a hash more difficult. Pre-computed dictionary attacks against hashing algorithms include attackers using big tables, sometimes called rainbow tables that list what common or likely passwords hash to, to allow them to get a hash value they are fully aware and may see whether it absolutely was computed by hashing a common password quickly. Including a random value, the salt, in the hash computation thwarts these types of attacks as the creation of such tables for data plus specific salt values can be computationally infeasible to compute and merely require an excessive amount space to store. As opposed to computing a table where data1 hashes to hash1, you'd need a table with data1+salt1 hashes to hash1, data1+salt2 hashes to hash2, and so on, for everyone possible hash values and salt values. Obviously, the longer the salt, more effective it truly is; recently (2011), sizes approximately 128 bits of salt are routine.

Exactly like with IVs, salt values should be random nonetheless they do not need to be placed secret. Indeed, this approach is used for password storage in countless systems: the salt value is stored along with the digest with the password for every user; the system looks in the salt, combines it while using submitted password, and hashes that in order to while using the stored password digest. Note that the use of separate salt values for each and every password makes attacks difficult, for attacker will need to brute-force crack each password hash individually if everyone uses independent salt values.
6.4.8 Public Keys

With regards to asymmetric cryptography, we've got asserted users publish their public key to ensure that anyone is able to see how it is. The programmer will find there's problem here: how does he be sure what a user’s public key in fact is? Imagine he publishes his public key after which an opponent reads it and publishes it out as his or her own? There are two general strategies to remedy this problem.

The first is the net of trust, as used by the favorite encryption scheme known as Pretty Good Privacy (PGP). Here, the user asks other users who already know the one requesting to digitally sign his public key together with their identity, usually email addresses. The concept is that the programmer may not know the user, however the programmer may have a friend of the user who signed the user’s key, the programmer trusts their verification so it truly does fit in with me. Or, maybe the programmer doesn’t know the user or anyone that knows the user, however, the programmer knows someone else who knows the user. The programmer will see the spot that the “web” section of the web of trust arises from. One other means of carrying this out is definitely digital certificates. Digital certificates involve the very idea of the trusted third party (TTP). Using such a scheme, we all agree that some other party is going to vouch for everyone. That party verifies us and attests that the public keys really remain in us. If the programmer would like to verify that the key the user has given is his actually, the verifier possibly can ask the third party and in addition they will reveal what my public secrets are. This kind of system is normally implemented using digital certificates, which is simply a standard means of including a public key, identity information, as well as the digital signature with the TTP inside a standard package. After the programmer obtains searching results for certificate, it offers the user’s identity and public key which is digitally signed by a TTP that the programmer simply trusts. So the programmer can be assured that everyone secret is mine. This kind of implementation is known as a public key infrastructure (PKI) and the entities that issue certificates are known as certificate authorities (CAs).

One common thought processes about public keys, PKIs, and certificate authorities is to consider them as ID cards or, inside the typical U.S. means of doing things nowadays, driver’s licenses. A lot of people sign checks and legal documents every day. To prove that this signature they've got produced is, the truth is, theirs, driver’s
licenses such as a copy from the person’s signature. They exist, among a number of other reasons, to be the document attesting to the fact that the person listed on the card gets the signature which is also present within the card. In this instance, the individual on card is the identity (the person) along with the signature will be the public key. The programmer will find there's whole separate argument about whether a driver’s license is an effective or appropriate method this, but that’s not much of this discussion right now. Regarding a PKI, an electronic digital certificate may be the license, the signature around the license is the public key, and also the DMV (Department of Motor Vehicles) would be the CA that issued the certificate. In the event the DMV verifies that you're user, they verify that by issuing a license. The programmer may notice a license, you've got some trust that this DMV verified the person for auction on it and the signature that appears on there is one of the person listed. Case such as a digital certificate; the CA has verified how the person for auction on it, plus the public key included on it, match. By doing this the programmer no longer need to verify himself. The programmer needs to simply trust the Certificate Authority.

### 6.4.9 Key Derivation

Essentially, cryptography offers us a vast set of tools which might be applied to solve data protection problems. On the list of primary things that could be jumping out at user is the issue of keys and key management. Symmetric encryption can help mitigate a lot of risk a part of storing data with an Android device, but where does the important thing are derived from? Where should I store it? Or must I even need to store it? When the programmer has decided to see, key management is usually the most difficult part of deploying cryptography on the app.

### 6.4.10 Motivation

Let’s look at an old example, the social media aggregator app. Have a look at store sensitive data, the credentials the user’s social websites accounts. We don’t want someone unauthorized to be able to read that data, so we care about confidentiality. In this instance, we could use encryption to mitigate potential risk of compromise with
the stored data. Even as there is only one entity (our app) ever accessing the info, a symmetric encryption algorithm is the suitable choice. When the app stores a whole new or updated credential, it will first encrypt the password, and then store the encrypted values. Once the app needs to access those credentials to visit on the site the programmer wants, it'll simply decrypt them when this occurs. The credentials will exist within the device only in encrypted form. The programmer might find himself questioning the storing of a password, even as we just said several sections ago we need to store the hash of any password and do not the actual value. In that case, however, we only ever wanted to verify how the password an individual gave us later was the precise value. Therefore, really do not should store the password because we could compare the digests to know if the supplied password is valid. In this case, however, we are going to send the actual password into an alternative party (the web 2.0 network). They cannot accept hash digests, but only real passwords, so we should store the particular password so we can decrypt it and send it on the social network server.

This is remarkably an easy task to code up. However, in order to do so, we need to supply a key for the encryption API calls. If we generated a random key and stored it combined with encrypted data, this will nullify the safety which is available from encrypting the data. And this isn't the answer.

6.4.11 Reverse Engineering

Storing an encryption key and also the encrypted data renders the safety offered with the encryption almost moot. It is never a good option and only slightly increases the project an assailant will have to insert to recover the info. The real reason for this really is that Android applications are usually reverse engineered. Applications are packaged into an APK file, which contains compiled code and resources. The programmer will discover multiple tools available that can take an APK file and recover all the contents, easily showing anyone who has an interest how a software works, the hardcoded values that are present, database connection strings, and the like. It's never a smart idea to make use of security by obscurity, that the programmer assumes that the attacker cannot figure out how the application works like a defense mechanism...an attacker can always work out how the job works and from any of the
values contained within. As a consequence, the risk analysis should assume that the attacker has full knowledge of this all information whilst the application ought to be created to resist assault from attackers with this knowledge. The programmer is able to provide some protection to the belongings in the APK, along with the design that may be revealed by reverse engineering it, by obfuscating it. This refers to an automated procedure that takes the application and turns it right into a functionally equivalent version that is certainly extremely tough to understand. This makes it making sure that an opponent that is attempting to reverse engineer the programmer could recover the code; nevertheless it would be very confusing and even more a hardship on them to understand, hiding an original design. However, it's going to still run much like the original well-designed version, making this not only a panacea against this threat.

Also, remember reverse engineering isn't specific to Android. Anything can be reverse engineered, given the required time and effort. While this process is a lot easier in environments like Android and Java, and it's also more challenging in environments including compiled C or C++ code that is a threat against all applications. The programmer can never prevent reverse engineering, however the programmer causes it to be much more difficult and even more costly, using techniques like obfuscation.

The reply to this challenge should be to derive the real key when it is needed. The app will need to prompt an individual to get a credential, normally a password and then use that value in order to create the real key. Basically, you'll want to have a variable-length input and are available up with a limited-length output, the key. Furthermore, it should be very unlikely that two different passwords would produce the identical key. Sound like anything to you? Yes, this can be the very definition of a hash algorithm. Older types of key derivation typically did this, and modern techniques depend on similar principles. In cases like this, we have to utilize a key derivation function (KDF) designed for password based encryption (PBE). There are a number of standard solutions to accomplish this, composed of multiple iterations of hashing, performed within the password combined with salt. The resulting value is then used as a key for subsequent encryption computations. Every one of this complexity, the inclusion of salt including a many iterations, serves to create cracking the password far more difficult than it will be when a single hash computation was adopted. Android has a good PBE algorithm making use of SHA-256 and AES, standardized
and strong algorithms, to compute a symmetric key coming from a password through hashing, encryption computations, and multiple rounds. To create a key from the String for example the passwords, add something like this to the code:

    String password = "...";

    String PBE_ALGORITHM = "PBEWithSHA256And256BitAES-CBC-BC";

    int NUM_OF_ITERATIONS = 5000;

    int KEY_SIZE = 256;

    byte[] salt = "xyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxyxy
String PBE_ALGORITHM = "PBEWithSHA256And256BitAES-CBC-BC";

int NUM_OF_ITERATIONS = 1000;

int KEY_SIZE = 256;

This part of the code just sets up what algorithms and parameters we can use. In this instance, we will utilize built-in Password Based Encryption key derivation function (PBE), using SHA-256 since the hashing algorithm and AES because the encryption algorithm. The CBC denotation inside algorithm specification points too were using AES in Cipher Block Chaining mode, a standard method of using AES when working with data that is constructed from multiple 128-bit blocks. We will use 5,000 iterations on the hashing function during the process (NUM_OF_ITERATIONS), because this is mostly accepted as a secure number and can make current brute-force attacks computationally infeasible, and definitely will get a 256-bit key (KEY_SIZE). They're standard parameters and can result in a strong key.

byte[] salt = "bgdfhchfgsdhs".getBytes();

Here, we setup the salt which will be utilized in the real key derivation computation. While you’ll recall, the salt is concatenated while using the value being hashed to generate cracking the password harder to try and do. In cases like this, the worth of the salt is usually anything and does not require to get kept secret, so we are going to just hardcode a unique value. To be able to always derive exactly the same key from the same password, we must be sure that the salt could be the same when; this is the reason we hardcode it into the program. Remember that we are only repeating this example; an increasingly ideal implementation would compute and store this salt value throughout the first run on the application on the particular device. This might enable each device to have a salt value that's unique to that particular device, the stronger implementation.

PBEKeySpec pbeKeySpec = new PBEKeySpec(password.toCharArray(), salt, NUM_OF_ITERATIONS, KEY_SIZE);

SecretKeyFactory keyFactory = SecretKeyFactory.getInstance(PBE_ALGORITHM);

SecretKey tempKey = keyFactory.generateSecret(pbeKeySpec);

SecretKey secretKey = new SecretKeySpec(tempKey.getEncoded(), "AES");
Here the main element is actually generated. First, we produce a PBEKeySpec object containing every one of the parameters we simply create, for example the password we intend to derive the key from, the salt value, the number of iterations we would like to use within the computation, and also the size of the key we want to derive. Only then do we build a Secret Key Factory that specifies the algorithm we are going to use (AES in Cipher Block Chaining mode). The PBEKeySpec will be passed to the Secret Key Factory to create a secret value, which can be then encoded for use with AES (because we'll be using AES to complete the actual encryption. The resulting 256-bit key, encoded for usage with AES encryption, is described using the secret Key variable. Understanding that’s it. We now have generated a symmetric key from the password. Here is the same process the app is going through whenever the user desires to access sensitive data; they will input their password and you'll derive the symmetric type in this fashion. The actual encryption and decryption in the data are going to be discussed next section, in the complete example.

### 6.4.12 Encryption in Void of User Specified Key

We have now seen the best way to derive an encryption key depending on some user-supplied data. If we must encrypt or decrypt stored data, we should prompt the person to be with his password and run it though a KDF, when we have just seen. Practically, it might make for an incredibly bad user experience if we prompted the person on her password every time we was required to store or read a single little bit of data. So what is frequently done should be to prompt the user when the application begins, derive the key, after which it cache it. While this does alter the risk calculation and reduce the security in the solution, it is a tradeoff which enables sense for any number of scenarios. Of course, there may very well be a situation the spot that the data being stored is of ultra-high sensitivity and that we may wish to prompt an individual to be with her password before every write or read than it; again, it's all regulated a question of applying appropriate safeguards for a given higher level of risk. Just what exactly if the programmer doesn't wish to prompt the person for any password at all, the app needs to do encryption or decryption operations? Unfortunately, there's no straightforward solution in charge of an Android app. The programmer will need secrets to perform encryption, so you'll want either the main element as well you'll be
able to derive the real key from the password, in the examples. Unless the programmer wants an individual to supply that critical piece of information, it is advisable to store it inside app or somewhere else for the device. This practice of storing a hardcoded key within the app isn't a secure technique of handling things, because an assailant can simply look over the app’s code to learn the key and thus compromise the data. Android apps provide little to no protection against reverse engineering, where an assailant may take a compiled application and recover the source code for it.

Unfortunately, Android just doesn't present the programmer with a better solution right now; there isn't a super safe place you'll be able to store that key or password in order that someone with having access to the device cannot have it.

Understand that protections need to be right the given level of risk. For sensitive data like passwords, storing the encryption keys hardcoded into the application is usually a poor choice and will not be done as that results are highly sensitive. If, however, the goal is simply for making things a bit more a hardship on an informal attacker, rather than someone with good levels of motivation to recuperate the info the application is storing, this kind of approach could be appropriate. We have a huge difference between storing data unencrypted over a removable media card and storing it encrypted with all the key embedded inside application that is installed on it itself. In cases like this, we're deploying a countermeasure towards the loss of the media card, as opposed to an assailant that might gain access to the two phones (in which the key exists) and also the media card in which the data exists. Is that this protection effective against a determined attacker? No. Tend not to confuse the self into convinced that storing a hardcoded encryption key as part of the application is ever secure; it amounts to obfuscation and bit more. However, apply appropriate safeguards with the amount of risk the programmer is attempting to address, which is all based about the data the application is storing.

Another option to consider is off-device storage. We can easily solve this concern using tools like encryption, but only then ought we to either prompt the person to deliver credentials to build the key after we require it producing a slightly degraded user experience or store the keys around the device resulting in an incomplete and insecure storage solution. If the app can instead make the best and protected connection time for a server after which ask that server because of its key whenever it
needs to perform an encryption or decryption operation, the important thing storage problem goes away.

6.4.13 Applying Cryptography to Shield against a Threat

Since we understand the basics behind the key cryptographic tools we have now at our disposal, let’s check out the whole data protection situation. Take a look at where we should store sensitive data, and now we should protect it from compromise. We love the confidentiality and integrity in this data. It is not safe whether or not it really is stored when using secondary storage media card or maybe a rooted phone. We should instead encrypt the information. Once the app needs to access the results, we’re going to prompt the person for just a password, derive the symmetric encryption key from that password, and then decrypt the data or encrypt it.

We’ve already seen enough of the code to do this whenever we discussed key derivation functions. Here’s the complete listing that will enable the encryption or decryption of data with different password:

```java
String password = "...;  
String PBE_ALGORITHM = "PBEWithSHA256And256BitAES-CBC-BC";  
String CIPHER_ALGORITHM = "AES/CBC/PKCS5Padding";  
int NUM_OF_ITERATIONS = 5000;  
int KEY_SIZE = 256;  
byte[] salt = "xyxyxyxyxyxyxyxyxyx".getBytes();  
byte[] iv = "1234567890abcdef".getBytes();  
String clearText = "..."; // This can be a value to become encrypted.  
byte[] encryptedText;  
byte[] decryptedText;  
try
```
So that the programmer can grasp that this various components of encrypting and decrypting data work, let’s experience this code and signalize established track record portions:

```java
PBEKeySpec pbeKeySpec = new PBEKeySpec(password.toCharArray(),
    salt, NUM_OF_ITERATIONS, KEY_SIZE);

SecretKeyFactory keyFactory = SecretKeyFactory.getInstance(PBE_ALGORITHM);

SecretKey tempKey = keyFactory.generateSecret(pbeKeySpec);

SecretKey secretKey = new SecretKeySpec(tempKey.getEncoded(), "AES");

IvParameterSpec ivSpec = new IvParameterSpec(iv);

Cipher encCipher = Cipher.getInstance(CIPHER_ALGORITHM);

encCipher.init(Cipher.ENCRYPT_MODE, secretKey, ivSpec);

Cipher decCipher = Cipher.getInstance(CIPHER_ALGORITHM);

decCipher.init(Cipher.DECRYPT_MODE, secretKey, ivSpec);

encryptedText = encCipher.doFinal(clearText.getBytes());

decryptedText = decCipher.doFinal(encryptedText);

String sameAsClearText = new String(decryptedText);
```

catch (Exception e)
{
...
}
```

So that the programmer can grasp that this various components of encrypting and decrypting data work, let’s experience this code and signalize established track record portions:
String PBE_ALGORITHM = "PBEWithSHA256And256BitAES-CBC-BC";

String CIPHER_ALGORITHM = "AES/CBC/PKCS5Padding";

int NUM_OF_ITERATIONS = 1000;

int KEY_SIZE = 256;

This section of the code just creates the algorithms and parameters we're going to use. However, we make use of the same key derivation parameters we previously discussed, with SHA-256 because the hashing algorithm and AES in CBC mode because the encryption algorithm. We are going to use 5,000 iterations on the hashing function during the process (NUM_OF_ITERATIONS) and will end up with a 256-bit key (KEY_SIZE). We also now specify the algorithm to work with to the actual encryption and decryption of data. We are using AES, still in Cipher Block Chaining mode, and employ PKCS#5 padding (a standardized method of padding some text to become encrypted). The padding setting specified how to pad the data being encrypted (or decrypted) when it is not in multiples of 128 bits, as AES operates on blocks of 128 bits at the same time. They are all standard algorithms and parameters, representing a robust encryption scheme (lately 2011, anyway).

byte[] salt = "bgfdfhchfgjsdhs".getBytes();
byte[] iv = "bgfdfhchfgjsdhs".getBytes();

Here, we setup the salt which is to be utilized in the important thing derivation computation along with the Initialization Vector (IV) which is to be utilized in encryption and decryption of data. We have already discussed the role of salt in computations involving hashing and the role of IVs in starting encryption when the programmer uses a mode (like CBC) that chains blocks together. Just like a salt, it is usually anything and should be kept secret, so we'll just hardcode a specific value. So as to always derive the identical key from the same password, we have to make certain that the IV, identical to the salt, is identical each and every time; that is why we hardcode it into the code.

Be aware that with this implementation, were employing a static IV to encrypt the data we're storing. Remember how much of an IV is employed for: it is designed to ensure in case the programmer encrypts exactly the same plaintext using the same key, you’ll end up getting another cipher text should the programmer uses a different IV. The use of the identical IV, and encrypt exactly the same plaintext sticking with
the same key, you’ll get exactly the same cipher text. For stored data, this is simply not a tremendous deal, but if you’re using it to make a communications protocol, you’ll want to use different and random IVs each time the programmer sends an email. Because of this someone observing the encrypted communications cannot know if the programmer signals identical message more than once.

    String clearText = ...;
    byte[] encryptedText;
    byte[] decryptedText;

Here, we just established the values we will encrypt and decrypt. With regards to this example, clearText will hold the String we would like to encrypt before we store. EncryptedText is a byte array which will hold the encrypted worth of the String. Finally, decryptedText will hold the decrypted bytes, once we encrypt the string after which decrypt it.

    PBEKeySpec pbeKeySpec = new PBEKeySpec(password.toCharArray(),
        salt, NUM_OF_ITERATIONS, KEY_SIZE);
    SecretKeyFactory keyFactory = SecretKeyFactory.getInstance(PBE_ALGORITHM);
    SecretKey tempKey = keyFactory.generateSecret(pbeKeySpec);
    SecretKey secretKey = new SecretKeySpec(tempKey.getEncoded(), "AES");

Here, we perform key derivation, going from the user-supplied password to a 256-bit AES key, just as we discussed within the last section.

    IvParameterSpec ivSpec = new IvParameterSpec(iv);

The programmer has to create the Initialization Vector (IV).

    Cipher encCipher = Cipher.getInstance(CIPHER_ALGORITHM);
    encCipher.init(Cipher.ENCRYPT_MODE, secretKey, ivSpec);
    Cipher decCipher = Cipher.getInstance(CIPHER_ALGORITHM);
    decCipher.init(Cipher.DECRYPT_MODE, secretKey, ivSpec);
We utilize the derived key along with the IV, along with our specification of what encryption parameters the programmer should use, to form a Cipher object. We actually create two, one in ENCRYPT_MODE and one in DECRYPT_MODE. Apart from that mode, the Cipher objects are created in exactly the same manner.

```java
encryptedText = encCipher.doFinal(clearText.getBytes());
```

As we hold the Cipher objects, we are able to perform encryption and decryption operations. First, we're going to encrypt. We pass the String we should encrypt (converted to a byte array) to the doFinal() approach to the encrypting Cipher object. The encryption operation is carried out along with the resulting encrypted bytes are returned to us. That’s all there exists for it.

```java
decryptedText = decCipher.doFinal(encryptedText);
```

```java
String sameAsClearText = new String(decryptedText);
```

The programmer can also decrypt data utilizing the same approach. In cases like this, we utilize decrypting Cipher object, passing it the encrypted bytes and having to plaintext bytes. Once we started having a String, we could reconstruct it, and the same AsClearTextString will contain the exact same value because clearTextString. And that’s it. We've got a total solution for protecting stored data, where we require a user-supplied password, derive a symmetric key from this, and encrypt and decrypt data at will, all in a few short lines of code. As we encrypt the data, we can store it inside a file, in the database, or wherever. As long as we maintain symmetric key from compromise, an opponent are not capable of recover the data from its encrypted form.

### 6.5 Secure Server Interactions

When information is being sent away from the device to someplace else, a security-minded developer must consider two primary considerations. With this context, we make reference to the capability of verifying that the entity we're communicating with, either sending data to or receiving data from, is the entity that any of us realize it. This really is essential for many reasons. First, we might want be sure that a pc that we are uploading data from the device is definitely an entity that will get it. Thus, in the example of the social networking aggregator, we have to send the Facebook account just to a Facebook server. Otherwise, we may be exposing confidential data to a party that should not need it, which has to be avoided. Also, we may want to
download data only from the trusted source. If the web 2.0 aggregator won't verify that the server it is lecture really is a member of Facebook, may display an update towards the user that develops from a rogue source—which often could potentially cause the person to take action he should not take. These examples illustrate the purpose, but they are not worst-case scenarios the apps must be capable to verify that the entities they are communicating with are, the truth is, and the entities they're saying to be.

One other consideration is the confidentiality of the data being transmitted within the network. This identifies steps that prevent a third-party from reading the data whilst it will be transmitted. In the matter of the social media aggregator app, we do not want someone uninvolved from the transaction to view the consumer’s social media marketing credentials as is also sent on the device towards the Facebook server. Pointed in the other direction, really do not want financial data sent from the bank server to app running on an Android device being read by another party who happens to have use of a critical section of the cellular data network. SSL/TLS: That is a Standard The gold standard in protecting data on the road over the web is the Secure Sockets Layer (SSL) as well as successor, Transport Layer Security (TLS). This protocol was created to provide both critical services organized from the preceding section: protecting the confidentiality of data as it's transported across a network and allowing the client to authenticate the server it's communicating with, so it knows it is sending data to, and receiving data from, the correct entity. It's also possible to make use of SSL/TLS permitting the server to authenticate the client, in a very reversal of the traditional mechanism.

The present discussion mainly focuses on protecting data that is certainly sent between the client (the Android application) and a server. In particular discussion is device authentication—letting the server verify it really is actually talking to the client it thinks it's and in addition allowing the client to verify it is actually speaking with the server it thinks it's. Authentication of the user is not thing discussion, but is one thing that the programmer should consider when coming up with the application plus the system it'll be an integral part of. Many applications call for a user use a username and password to authenticate the person. While commonplace, this implies the credentials being passed for the server whenever, which even though done this over an encrypted channel, seriously isn't ideal. Better solutions do exist, and so are
supported within Android, that enables the client to build an expression and send that to a server for ongoing authentication.

The Account Manager class in Android allows the job to train on a token to authenticate towards servers in question. Indeed, this is why the Google applications authenticate an individual to Google’s servers as an alternative to pass the username and password within the channel all of the time. That is a much more resilient way of credential management and usage. Designing the job to use Account Manager and an Authenticator can be a strong approach for user authentication and really should be used instead of passing passwords, if at all possible.

### 6.5.1 Authentication

The primary function of SSL/TLS is always to enable the client to verify that it's, actually, talking the entity (the server) it thinks it is. Each time a client browses to a hostname (or an IP address), it has a reasonable expectation how the entity it can be speaking to could be the one which is assigned to that hostname (again, or IP address). However, there is a whole slew of attacks and techniques the programmer can use to make that communication to happen with other rogue entities. The type and varieties of these compromises usually are not vital to an knowledge of this topic, so we will not discuss them here, but recognize that even though the app tries to get connected to a server which has a specific hostname or IP address, the entity that responds back is probably not normally the one the programmer believe it is.

SSL/TLS uses asymmetric key cryptography and optionally a public key infrastructure to supply this server authentication service, even as we will discuss later. One of the primary levels in an SSL/TLS communication necessitates the server sending its digital certification to the client. This certificate is employed to spot the customer and typically includes the server name, the certificate authority that issued the certificate, plus the public key that is among the server. At this stage, complainant can attempt to verify which the certificate presented is correct and valid, or can select to decline to execute this validation. Clients must always perform the validation to produce certain the certificate to be real signed from the CA that this server claims issued it and that the certificate is not a forgery. The programmer will discover, however, lots of client implementations that don't perform this task.
Once the client verifies which the server’s certificate is valid, it will generate a random number, using whatever random number generation routines can be about the client’s platform. It'll encrypt that random number with the entire public key supplied by the server in its digital certificate after which send the encrypted value returning to the server. The server will then use its private key, which only it offers entry to, to decrypt that random number. This step verifies the client is, in reality, communicating with the server whose thinks it truly is, one identified within the digital certificate. Because a certificate authority has issued this certificate specifying the general public key used and the server identity, so we trust the CA, we understand that the public enter in question does, in point of fact, remain in the server listed. If we encrypt a value with this public key, only the entity that has use of the related private key will be able to decrypt it. So if we verify the certificate which the server presents is, in reality, valid so we encrypt something using the contained public key, just the entity listed in the local certificate will be able to decrypt the value, and the same random number is going to be present on the server plus the client, if the server had not been owning the corresponding private key, it would not have the ability to decrypt the worthiness successfully and would, therefore, stop capable to know the value in question. This random number is employed to generate a shared secret key relating to the client as well as the server, so only they will compute the shared secret key the master secret. After this handshake occurs, the customer and the server communicate using symmetric encryption which utilizes the key derived from the master secret, plus the client sees that it has to be communicating with the server it thinks it can be. Also possible with SSL/TLS, but is not nearly as common, is perfect for the server to authenticate the client beyond just the client authenticating the server. SSL/TLS was originally made with ecommerce in mind, where many clients running browsers would connect with a few volumes of servers. However, the server isn't going to care who is lecture it, however the client absolutely cares which server it really is actually talking to. However, SSL/TLS can give to get a server to authenticate the client, in situations where the server should communicate simply with a select group. In this particular SSL/TLS: The Standard case, be sure the programmer certificate that is among the client is sent within the SSL/TLS handshake. This certificate offers the same kind of information as being the server’s certificate, but identifying the customer, along with the communications in the server for the client make utilization of the public key contained within the certificate. So only clients who develop the
private key that corresponds for the public key inside the certificate can successfully decrypt the information from the server (this is the one-to-one parallel with how server authentication is performed). Client-authenticated SSL/TLS is rarely done, since the majority of clients don't have digital certificates with which to spot themselves (indeed, server owners pay a significant cost for the CA that issues their identity digital certificate to make up them for the verification which they perform). However, if a server does want to authenticate the clients that make an effort to contact it, this is a technique that can be used. The need to supply clients using a digital certificate to use for identity purposes, either coming from a commercial certificate authority or coming from a private certificate server that the programmer can stick up himself is sort of problematic, as we will discover inside a later section.

### 6.5.2 Encryption

Once an SSL/TLS handshake is complete, and also the master secret has become computed by both client as well as the server, symmetric encryption keys are derived from this value. The complete method by which this occurs isn't relevant to programmers that use SSL/ TLS, rather than those who design and study it, so we start to use not discuss it here. Once the symmetric encryption keys are derived, each message between the client and the server is encrypted with them, typically by utilizing a stream-based encryption algorithm. SSL/TLS messages also typically add a Message Authentication Code (MAC) that is a construct of the message itself and a secret value created from the master secret; this permits the person receiving the content to ensure its authenticity. Remember that successful decryption with the message from the recipient already implies this; the inclusion on the MAC is optional and derived from a solution make use of SSL/TLS while specifying a null cipher, where encryption couldn't survive performed.

Using null ciphers with SSL/TLS removes the protection which is available from the encryption. This type of configuration should never be used. It may be used during testing of a new application, but more desirable approaches to testing and debugging exist.
6.5.3 Protecting Data

Ever since could how a SSL/TLS protocols work, we can easily glance at the implementation in Android and ways in which we're going to build a secure connection between the application and the server it can be conversing with. For the factors like this discussion, we are going to assume were speaking with a public server, meaning built to be using an SSL/TLS server certificate issued by a poster Certificate Authority, whose root is trusted from the Android system automatically. When Google releases Android builds, they range from the root CA certificates from many different providers, which issue commercial certificates to entities that prove their identity with regards to servers. If we should instead throw open a secure communications path encrypted to shield the confidentiality of the data and authenticated to ensure we are communicating using the server we presume were, the code required is minimal.

6.5.4 SSL/TLS Environment

The common strategy to build an SSL/TLS-protected communications channel to the server is by using the Https URL Connection class. This can be a class based on the common Http URL Connection class, but implements SSL/TLS with the connection with all the standard Android SSL/TLS configuration. The best way to open appreciable link using HTTP over SSL/TLS would be to call openConnection() on the URL object, specifying the HTTPS protocol:

```java
URL url = new URL("https://clientaccess.example.com");

HttpsURLConnection urlConn = (HttpsURLConnection)url.openConnection();
```

Or alternatively, the programmer is able to build a HttpsURLConnection directly:

```java
HttpsURLConnection urlConn = new HttpsURLConnection("https://clientaccess.example.com");
```

Following the HttpsURLConnectionis open, the programmer can use it just like a regular HttpURLConnection:
As we see, opening a SSL/TLS-protected link to a server using a reliable CA-issued certificate can be just as easy as opening up a typical HTTP connection. When the programmer finally will have an SSL/TLS connection, the programmer can examine various data about it, since the HttpsURLConnection class exposes a lot of that information. Rarely would the programmer have to know it, but it is a good idea for testing, and it is good to find out that the programmer could know more about it will the programmer needs to. Let us discuss one or two degrees of information that the programmer could get in the HttpsURLConnection object:

getCipherSuite()
This kind of returns the cipher suite just for this connection. This tells the programmer the effectiveness of the root protection mechanism with this communications path.

getServerCertificates()
This process returns several the many certificates how the server supplied as identifying information. This allows the programmer to definitely view the entire chain of certificates from the basis CA to the certificate for the actual server the programmer might be communicating with.

6.5.5 Server Verification

On the list of features of SSL/TLS is hostname verification. When an SSL/TLS connection is to establish, the SSL/TLS environment compares the hostname from the

```java
try {
    URL url = new URL("https://clientaccess.example.com");
    HttpsURLConnection urlConn = (HttpsURLConnection)url.openConnection();
    urlConn.setDoOutput(true);
    OutputStream output = urlConn.getOutputStream();
    InputStream input = urlConn.getInputStream();
}
```
server the programmer is hoping to plug to, to the certificate that's presented through the server itself. Should the verification pass, the programmer can be assured that the call was established okay. However, if the hostname specified by the URL won't match the hostname in the certificate presented because of the server, the call will fail through an exception. This is intended to ensure that the programmer is connecting to the server the programmer thinks he might be. So connections to clientaccess.example.com require the server to which the programmer is connecting to supply a clientaccess.example.com certificate but not one using a different name.

The hostname verification function is implemented by classes that produce technique Host name Verifier interface. Android has a few concrete classes that implement this interface and perform hostname verification in different ways:

**AllowAllHostnameVerifier**

This hostname verifier basically turns hostname verification off. Providing the certificate which is presented through the server is trusted (i.e. develops from a trusted CA), it's going to be accepted whether or not the hostname from the URL matches the name within the certificate. Many clients skip verification in this fashion, but it is recommended that the programmer simply considers the more time and traffic to do the verification.

**StrictHostnameVerifier**

This hostname verifier matches the verification in the default (non-Android) Java releases. It checks the very first CN seen in the server certificate resistant to the hostname per the URL and also checks the many subject-alt entries inside the certificate for a match. Wildcards are permitted, but only up to one level. By way of example, *.example.com would match server1.example.com, however, not server1.domain1.example.com).

**BrowserCompatHostnameVerifier**

This hostname verifier is like Strict Host name Verifier, but wildcards inside the certificate name will match multiple levels. For example, *.example.com would match server1.domain1.example.com in this method, whereas Strict Host name Verifier would reject it.

If the programmer needs to implement hostname verification for certificates using some kind of verification not contained in one of these providers, simply create the own Hostname Verifier class. All that the programmer should do is implement the Host name Verifier interface. It has only 1 method—verify()—that can within the
URL to verify an SSL Session object, where you’ll be able to have the name about the certificate. Perform whatever verification logic the programmer would like to and return a Boolean value to point regardless of if the certificate should be accepted or otherwise. An entire implementation may resemble this:

```java
HostnameVerifier customHV = new HostnameVerifier()
{
    public boolean verify(String urlHostname, SSLSession connSession)
    {
        String certificateHostname = connSession.getPeerHost();
    }
};
```

When the programmer has a Host name Verifier object, either through the use of one of several provided implementations or by creating the own personal, it is advisable to configure the SSL/TLS environment to make use of it when creating new SSL/TLS connections. To get this done, simply call set DefaultHostnameVerifier(), a static technique of the Https URL Connection class. Any future SSL/TLS connections can make utilization of the specified Host name Verifier:

```java
HostnameVerifier newHV = new StrictHostnameVerifier();
HttpsURLConnection.setDefaultHostnameVerifier(newHV);
```

### 6.5.6 Hostname Verifier

It is important to remember that hostname verification and certificate trust are two different things. To ensure that an SSL/TLS link to be established, both checks must pass. First, the certificate the server presents have to be trusted; in most cases that means 1 of three are actually from a dependable Certificate Authority. Second, the hostname verification must match; because of this the name for the certificate must match the hostname the programmer is attempting to plug to. These are separate issues. With that being said, it truly is entirely possible to use any criteria the
programmer wants to determine in a custom Host name Verifier. By way of example, suppose the programmer desired to reject certificates from a particular CA that's, automatically, trusted in Android. The programmer could potentially create a Trust Manager that would trust every CA that is normally trusted by default aside from the main one in question (we’ll see this approach shortly). But that’s lots of manual effort. The programmer may instead decide to go through the CA that has issued the certificate in the Host name Verifier and tend to reject the verification if the certificate the programmer wants was from the CA the programmer does not need to trust. This is a good example:

```java
HostnameVerifier customHV = new HostnameVerifier()
{
    public boolean verify(String urlHostname, SSLSession connSession)
    {
        boolean isCertOK = true;
        String certificateHostname = connSession.getPeerHost();
        try {
            Certificate[] certs = connSession.getPeerCertificates();
            X509Certificate caCert = (X509Certificate)(certs[certs.length - 1]);
            String caName = caCert.getIssuerX500Principal().getName();
            if (caName.equals("CA_WE_DO_NOT_WANT_TO_TRUST"))
            {
                isCertOK = false;
            }
        }
    }
}
```
As can be seen, that one can be done pretty much any check that one wants with a Host name Verifier to receive or reject. The factors the programmer ultimately chooses, and the way the programmer chooses to enforce them, is entirely the responsibility good risk analysis for the application.

6.5.7 SSL/TLS Connection Errors

Even as have discussed, when attempting to make SSL/TLS connections, two primary errors may come up. First, checking the trust in a server certificate may fail should the certificate just isn't from a trusted CA. Second, verifying the hostname may fail should the hostname inside the connection URL isn't the same as that specified around the server’s certificate. When it comes to server certificate trust, an attempt to connect can lead to an SSL Exception with the exception text indicating that a “not trusted server certificate” was provided. In case such things happen, what if the client application does? The answer then is very simple: don’t talk to that server. At this time, some developers feel that the answer is with an ordinary HTTP connection instead of the protected HTTPS one.
This could completely get rid of the protections for confidentiality and authentication that SSL/TLS provides. Indeed, take into consideration what this type of decision would mean: the programmer is unable to verify the server the programmer might be speaking with would be the one the programmer thinks it really is, and that means the programmer just ignores this and will speak to them anyway? This may not be an extremely secure or safe practice! Indeed, it’s entirely possible that very little could possibly be wrong. The server in question is utilizing an expired certificate, or there have been a mis configuration by those maintaining the server. These cases are possible, but we simply cannot be sure. All we understand is we’re able to not verify the server’s identity.

So, we must not speak with it. It could be difficult to keep server certificates up-to-date and configured correctly. When a certificate expires, browsers will display one when they hook up with it. If a certificate isn't going to exactly match the name entered into a browser e.g. a request to example.com resulting in a certificate for www.example.com coming back, browsers will display an error. The two of these the situation is the explanation for many false alerts men and women see from other browsers.

Exactly the same can be stated of hostname verification. This check makes sure that the certificate being presented matches one the programmer simply asked. Suppose the programmer walked into a conference room and asked for Mr. Smith. Men claiming to become him get up and teach the programmer his identification, which reads Mr. Taylor. Can the programmer trust that man would be the one the programmer is interested in, just because he taken care of immediately the query for Mr. Smith together a valid driver’s license? Course the programmer doesn’t. And the client application must do the same. Hostname verification, like server certificate trust checking, is performed for a reason and is particularly critical to the authentication function of SSL/TLS. But if the application receives a failure in a choice of one of these simple checks, tend not to communicate further with that server, plain and straightforward.

Making sure that certificates presented by the server are trusted (issued by way of trusted CA in the matter of public-facing servers) and performing hostname verification are critical on the authentication function of SSL/TLS. If the programmer cannot verify who the programmer is speaking with, stop speaking with them!
6.5.8 Protecting Data

The prior section showed the best way to protect data on the way to public services, where our application may need to hook up with a wide range of servers and/or these servers may be from the control. In those cases, validation on the SSL/TLS certificates used by those servers is accomplished while using the built-in Android PKI support and relies upon these servers using certificates which might be signed within a chain that commences with a root CA certificate offered with Android. This model successful if we need to speak with servers that so, but provides extensive issues when we are communicating with one specific server (or perhaps a small group of them), all of these they are under the control and therefore are designed to serve the customer application. We're going to now take a look at how the programmer can modify the configuration from the Android SSL/TLS implementation to limit which server certificates will probably be trusted, allow the trust of self-signed certificates, and use client certificates to spot the client application on the server.

6.5.9 Using Specific Certificates for SSL/TLS

The normal SSL/TLS environment accepts all server certificates which have been from a trusted Certificate Authority; along with the Android system contains a big list of certificates for CAs to trust. When results are being exchanged with specific servers, the SSL/TLS environment can be configured allowing only specific certificates to be used. This permits that the programmer lock down the servers that the application can connect. In addition, it permits the programmer to configure the SSL/TLS environment to trust certificates that aren't issued by a trusted CA, such being a self-signed certificate. Once the client will likely be connecting here we are at a server that the programmer controls, that is a valid and effective way of securing the SSL/TLS communications link between them and ask the programmer to purchase certificates from commercial CAs.

To illustrate this security model, we will configure an Android application to trust a self-signed server certificate. Other certificates, including those issued by Certificate Authorities which have been normally trusted by the Android system, are going to be rejected along with the device won't be in a position to establish SSL/TLS sessions
with these. Everything done here works equally well having a CA-issued certificate as opposed to a self-signed one; we'd just export the certificate from the server as an alternative to generating it ourselves, as we are about to do.

To be able to fix a personal client-to-server secure communications path, the first task would be to install a self-signed certificate with the server's web server. There are lots of ways to get a real certificate. One of the ways uses the existing Android tools, specifically the keytool CLI utility. For the example, we are going to first operate the keytool to come up with the certificate, in cases like this for any host named server.example.com:

```
keytool -genkey -dname "cn=server.example.com, ou=Development, o=example.com, c=US" -alias selfsignedcert1 -keypass genericPassword -keystore certsjks -storepass genericPassword -validity 365
```

This command generates certificates to the server.example.com, the complete DNS name for the server the Android application could communicate with. The validity argument specifies that the certificate is going to be valid for 365 days, the programmer start with the modern. The certificate is generated through the keytool utility and trapped in a key store that we call certsjks. This key store influences JKS format, short for Java Key Store. User can think of a key store as being like a safe, with many different bits of important information stored within (in this case, just one piece: the new certificate and it is associated private keys). We assign password strength to the JKS key store this password is needed to do one thing while using key store in order to the certificate itself, this password must export a piece of paper outside the key store or to do anything whatsoever else from it.

We've got a self-signed certificate within the name of server.example.com. Now, we should instead do a pair of things with it. First, we should transfer it (and its particular associated private keys) to the server that will utilize it (server.example.com in this example). The exact technique of this will depend on what sort of server the programmer could have, and is not really part of the Android effort, so we won't discuss it in greater detail here. However, we should also set it up in the Android application so that it recognizes and trusts the certificate when the server presents it. To do this, we must obtain the certificate in a very more usable form, because Android applications do not directly help JKS key stores. Rather, they prefer to use BKS key stores, short for Bouncy Castle Key Store. Bouncy Castle is really a library of cryptographic routines that Android uses of much of its cryptography
implementation, key stores included. So we have to get the certificate out of the JKS key store and right into a BKS key store before we can easily make use of it in the Android application. So that the programmer can create a BKS key store with this certificate, we have to download the Bouncy Castle cryptographic provider. The programmer will discover this on-line when the programmer go to the Legion on the Bouncy Castle website and downloading the modern provider JAR files.

Upon having this file downloaded (say, in the /home/user1/dev/libdirectory), you'll be able to utilize key tool to export the self-signed certificate from the JKS key store and to the BKS key store:

```
keytool -export -alias selfsignedcert1 -keystore certsjks -storepass genericPassword -keypass genericPassword -file cert.cer
```

```
keytool -import -file cert.cer -keypass genericPassword -keystore /home/user1/dev/project1/res/raw/selfsignedcertscks -storetype BKS -storepass genericPassword -providerClass org.bouncycastle.jce.provider.BouncyCastleProvider -providerpath /home/user1/lib/bcprov-jdk16-146.jar -alias selfsignedcert1
```

The first of the two commands exports the certificate out of the JKS keystore by which we created it, right into a raw certificate file, cert.cer. Then, while using second command, we produce a new BKS key store and import the certificate in it. Observe that we should instead supply all of the way to the JAR file we downloaded from Bouncy Castle to complete this operation. Also, note that we assign the password to the BKS keystore, just like we did with our JKS one.

We have the certificate in the BKS key store. Step 2 will be to include that key store inside the Android app, making sure that we can employ it when setting up an SSL/TLS connection to the server. Recognize that we created the BKS key store in an application’s project directory, /home/user1/dev/project1/res/raw/. That is a resource directory. Whatever development environment that the programmer is using should detect which a new raw resource have been copied there and include it with the project, permitting the programmer to take advantage of it.

Now that the key store together with the certificate can be purchased to the Android application, user should configure the approval to open SSL/TLS connections utilizing it. This method might be a complex and requires the usage of multiple objects:
Now let us take a look at how to use these classes together to try and do the goal associated with an SSL/TLS secure session to some server using a self-signed certificate. First, the programmer should create and employ a Key Store object to access the key store that the programmer simply added as being a resource. The process for this is easy to understand:

```java
KeyStore selfsignedKeys = KeyStore.getInstance("BKS");
selfsignedKeys.load(context.getResources().openRawResource(R.raw.selfsignedcertsBks), "genericPassword".toCharArray());
```

This code reads raw resource that may be actually the BKS key store and uses it as being the Key Store class object. Be aware that we have to provide the programmer with the password that is utilized to protect the key store (the main one we set if we created the key store) in order that Android can open it and access the certificate contained inside. Next, we create a Trust Manager Factory object and initialize it by using this Key Store. This configures the Trust Manager Factory to produce Trust Manager objects based on the contents of the Key Store. Do not forget that a Trust Manager is accustomed to decide whether a server’s certificate should be trusted, so by building a Trust Manager around the key store containing the self-signed certificate, we're building the one that will trust the certificate. To create the Trust Manager Factory, we do the next:

```java
TrustManagerFactory trustMgr =
    TrustManagerFactory.getInstance(TrustManagerFactory.getDefaultAlgorithm());
trustMgr.init(selfsignedKeys);
```

This creates a Trust Manager Factory that will produce Trust Manager objects which use the default SSL/TLS algorithms (it is precisely what the getDefaultAlgorithm() method call does) and can utilize the custom key store, that's set in the call to init() that passes in that key store, to make the decision which server certificates to trust. Even as contain the Trust Manager Factory, the programmer can create the SSL Context that we're going to need to generate SSL/TLS connections using the certificate:
SSLContext selfsignedSSLcontext = SSLContext.getInstance("TLS");

selfsignedSSLcontext.init(null, trustMgr.getTrustManagers(), new SecureRandom());

HttpsURLConnection.setDefaultSSLContextFactory(selfsignedSSLcontext.getSocketFactory());

The call to init() takes three parameters: the causes of the client-side keys used, the sources of trust, and also the types of randomness. As we usually are not doing SSL/TLS where the server authenticates the client but only the place that the client authorizes the server, we need not supply the first parameter and merely pass null.

The second parameter is where we give the programmer the Trust Manager Factory’s generated Trust Manager list. For the reason that Trust Manager Factory was configured to apply the key store we created which includes only the self-signed certificate involved, this new SSL Context will accept only the certificate no others, not just the normally accepted CA-signed certificates. Since the goal of this is the private client/server secure communications path that is fine (indeed, this is exactly what we would like; consider the Principle of Least Privilege). Another parameter is how the random numbers required for SSL/TLS are generated; take a look at make use of a new Secure Random instance.

Finally, the static method setDefaultSSLSocketFactory() on the HttpsURLConnection class is called. Using this method changes the SSL Socket Factory used to make new SSL/TLS connections. This line creates an SSL Socket Factory based on the SSL Context and then configures the HttpsURLConnection class to work with it, making sure that every one of the parameters we have now set—the trust of the own self-signed certificate—are utilized. From this level forward, any SSL/TLS connections made from the application use this new configuration and, therefore, are able to be made to the server and nowhere else:

URL serverURL = new URL("https://server.example.com/endpointTest");

HttpsURLConnection serverConn = (HttpsURLConnection)serverURL.openConnection();

We've just discussed how a decision is created in regards to the trusting in the certificate presented through the server. Remember that two checks are created in the fix of an SSL/TLS connection: certificate trust and hostname verification. Even when we configure the SSL/TLS environment for the application to trust a self-signed certificate, the hostname verification checks are still made, to ensure the name around
the certificate must still match the name of the host the user is trying to connect in order for the link with be fix. While we are employing a self-signed certificate with a different name, a custom Host name Verifier may need being configured as the programmer can see previously. This should be avoided if possible, because hostname verification checks are essential to generate sure we're lecture the server that we think we're.

A measure Further: Using Client-Side Authentication SSL/TLS until now, we now have seen how the programmer can modify the SSL/TLS configuration of an application by changing its Trust Managers so it would trust specific certificates. This had two major effects. First, it allowed the Android application—you—to trust a self-signed certificate within the server, something the default SSL/TLS configuration would not do. Second, it forced the Android application to trust a certificate only from the server it becomes discussing with. This enables the customer to learn that it really is talking with the server that it thinks it really is, and thus the communications between them are safe with strong encryption. This can be a right beginning to a secure communications channel.

However, there is certainly still one hole on this solution: the server has not yet authenticated the client, so as the client witnesses that it truly is speaking with the server it would like to, the server doesn't have any assurances that it is discussing with the customer it would like to. In a controlled environment, such as a company application that logs in employees and lets them conduct sensitive operations or read proprietary information, the server may wish to be sure that only authorized products are getting in. Hence the necessity for client-side authentication in SSL/TLS.

We will now implement such a solution. We'll extend the previous demonstration of trusting self-signed server certificates to deliver client certificates towards the server also. Remember that to require clients that connect using SSL/TLS to provide a certificate and to specify which certificates should be accepted requires configuration around the server. This area of the process is beyond the scope with this book, largely considering that the process can vary greatly according to which kind of web server you're going to be communicating with. For the present time, let's assume that you've this build in the server and you've got a client certificate that you'll want to deploy in to the Android application. We will further assume the programmer has that client certificate in a BKS format key store, named client authors BKS and the programmer also still have the self signed certsBKS key store containing the server certificate.
Starting with those two key stores, we are going to create the client for mutually authenticated SSL/TLS, with only specific server certificates trusted. Just as before, we must develop a Key Store object to reach the BKS key stores which can be in the project. First, we’ll setup the Trust Manager Factory that we need to make sure we are talking merely to the servers we want to. Here is the exact same code as with the prior section:

```java
KeyStore selfsignedKeys = KeyStore.getInstance("BKS");
selfsignedKeys.load(context.getResources().openRawResource(R.raw.selfsignedcertsbsks), "genericPassword".toCharArray());

TrustManagerFactory trustMgr =
TrustManagerFactory.getInstance(TrustManagerFactory.getDefaultAlgorithm());
trustMgr.init(selfsignedKeys);
```

Now we're going to perform largely the identical process, but we need to develop a Key Manager rather than Trust Manager. Key Manager Objects represent the own credentials (certificates) which might be supplied to the server for reasons like client-side authentication in SSL/TLS. The procedure is analogous to a Trust Manager, because we develop a Key Manager Factory based within the Key Store object holding the certificates:

```java
KeyStore clientauthKeys = KeyStore.getInstance("BKS");
clientauthKeys.load(context.getResources().openRawResource(R.raw.clientauthcertsbsks), "genericPassword".toCharArray());

KeyManagerFactory keyMgr =
KeyManagerFactory.getInstance(KeyManagerFactory.getDefaultAlgorithm());
keyMgr.init(clientauthKeys, "genericPassword".toCharArray());
```

Similar to after we were creating a Trust Manager Factory, this creates a Key Manager Factory that will produce Key Manager objects designed to use the default SSL/TLS algorithms (that’s what the getDefaultAlgorithm() method call does) and may utilize the custom key store, that is emerge the letter to init() that passes because key store, to determine which client certificates to supply to servers that want client-side authentication. Note that however, we need to give the programmer the key store
password on the init() function, as we will be utilizing the private keys associated with the certificates, something we would not want to do when getting through a Trust Manager.

We now have a Trust Manager Factory to generate Trust Provider objects that may accept certificates a part of the key store when they are furnished by a server we hook up to (the self-signed certificate in this case). We in addition have a Key Manager Factory to generate Key Manager objects that may provide client certificates that any of us will use to identify ourselves to the servers. Basic, the programmer can create the SSL Context that we ought to make SSL/TLS connections.

```java
SSLContext privateSSLcontext = SSLContext.getInstance("TLS");

privateSSLcontext.init(keyMgr.getKeyManagers(), trustMgr.getTrustManagers(), new SecureRandom());
```

Recall how the call to init() takes three parameters: the options for the side keys used, the causes of trust, as well as the reasons for randomness. These times, were doing client-side authentication, so we include the Key Manager objects since the first parameter. Then we include the Trust Manager list as the second parameter. Finally, because third parameter is when the random numbers essential for SSL/TLS are generated; have a look at work with a new Secure Random instance.

And that’s it. We now have an SSL/TLS configuration that restricts which server certificates we will accept (the belongings in the self signed certs key store) and includes client certificates that people may use to distinguish ourselves to people servers. By utilizing these constructs, we are able to fully specify how the programmer wants the SSL/TLS communications to do something for the application. The ability to specify both who we're prepared speak with and also to whom we will identify ourselves allows the advance of private, secure communication tunnels which might be both mutually authenticated and encrypted, suitable for the concept of Android client applications communicating with servers.

```java
URL serverURL = new URL("https://server.example.com/mutualauth_endpointTest");

HttpsURLConnection serverConn = (HttpsURLConnection)serverURL.openConnection();
```
6.5.10 Threats while Data is in Transit

Aside from the confidentiality with the data the app might be transmitting or receiving, the app needs to be competent to defend itself against rogue data submitted to it. As the developer of an mobile app, the programmer is writing code that runs locally over a device, plus the inputs to the next app is extremely tightly controlled. Maybe the programmer accepts input through the user by means of the screen and/or keyboard. The programmer might accept input above the network from other entities that speak with you. Also the programmer can read data stored for the file system that the programmer just previously read. In all of such cases, however, you'll want to account for attackers which may be in a position to mount their dastardly deeds by manipulating those inputs on the app. Web application developers should be extremely interested in proper input validation because malicious input which the programmer neglect to catch may result in dangerous vulnerabilities including SQL injection (or other command injection), cross-site scripting attacks, and exploitation of buffer overflows. Client applications, including Android applications, will not be as vulnerable to these kinds of attacks, but the programmer is not immune either. Being an application developer, Android or iOS, client-side or server-side, you'll need to be aware about the advantages of proper input validation as well as the consequences if the programmer may not properly perform it. This really is truly the main tenant of application security today.

The principal programming flaw practically in most now a day’s applications is really a deficit of proper input validation. Applications trust that the input that is presented to them really they expect. In the end, should the programmer asks a user to deliver a username that is certainly between four and eight characters, ya think that they can provide everything else? Even though the programmer does, could the programmer thinks that rather than meeting that criteria, they will supply 10000 characters instead? The programmer might very well not. But failing to consider a really case may give rise to some classic symptom in application security: the buffer overflow. Now, Java (this also includes Android) applications are less complicated harder to exploit using buffer overflow attacks, however it is still possible. Other vulnerabilities beyond buffer overflows exist as well. This can lead to our primary point with this section: never blindly trust input into the job; validate anything before the programmer decides
to process it. The key tenant of application security is a snap. Don't trust any input into you. Validate it first, and then process it.

Let's consider an example of SQL injection, which can be one specific (even so the most prevalent) form of a class of vulnerabilities generally known as command injection. On this form of vulnerability, input from some other source is used directly in a command, in this case a SQL expression that's passed to some database interpreter. E.g., consider a case where the user offers account information with an application, which must then verify that this combination is valid to allow for access. In such a case, an average SQL statement that has to be issued by the application form towards the database would be:

```java
String loginQuery = "SELECT * FROM useraccounts WHERE userID = " + request.getParameter("userID") + " AND password = " + request.getParameter("password") + " and ";
```

In this example, the programmer will observe that the application has a SQL statement that's designed to see whether or not the submitted user ID and password are valid. However, if whoever is submitting the info were to submit these values:

```java
userID = ' or 1=1 -
password = doesNotMatter
```

The resulting query that could be supplied towards SQL database interpreter could well be:

```sql
SELECT * FROM useraccounts WHERE userID = ' or 1=1 - AND password='doesNotMatter'
```

This SQL statement would then evaluate and return the many data from the user accounts table, since the WHERE condition would be true (a result of the OR 1=1 condition) and also the password checking clause will be commented out. This unintended behavior is created possible as a result of two primary problems. First, our app didn't properly validate the input that had been received before using that input to the SQL statement to become listed in the database interpreter. Second, the issue is enabled because when query strings are constructed in that manner, the database system cannot tell the difference between code and data; the initial apostrophe ('') submitted within the user ID is definitely data, but has interpreted as code for the reason that database cannot tell who's must be interpreted as being a literal data value.
Consider some of the lessons here? First, SQL injection is a style of vulnerability that's made possible when proper input validation isn't performed. Expanding within this, as an alternative to search for each specific vulnerability, developers need to take one very easy concept to heart and contemplate it when writing the portions of these application that cope with processing input, no matter where can it be received from. All input is not to become trusted and need to be validated prior to it being utilized in any way. User, for application developer, has zero control over the proceedings over and above the application. The programmer cannot control the device’s operating system, the programmer doesn't control the wireless base station how the device is communicating with, and also the programmer doesn't control one of the routers that the data about to, or from, the device passes through. For this reason, the programmer must consider any incoming communications to be hostile and maliciously tampered with, so that the programmer must properly validate and/or sanitize them prior to using them, to guard you. Second, database calls in which the entire SQL statement that'll be shipped to the database consists of text concatenated together, mixing commands and data, is dangerous. Indeed, SQL injection attacks cannot be fully mitigated by input validation because sometimes dangerous inputs (such as apostrophe, when it comes to SQL) are often valid inputs. In such cases, it is advisable to separate the command/ control portions of the query through the data that is used to construct it. Luckily, the database programming methodology of Android (and of Java) supplies a means to do this. So, let’s move ahead and discuss those two lessons in greater detail. First, we'll explore input validation so move on to safe database interaction.

### 6.6 Input Validation

As we have observed, all input to the application need to be considered untrusted and must be checked before we are able to apply it. We've got already explored ways of make sure that the data we've got received did, in fact, range from entity we expected it to. We've also seen complementary methods to ensure we have been sending data towards entity we feel we are sending it to. But whether or not the app accepts data from anywhere, or we've nothing but absolute trust in the entity we have been
accepting input from, we must validate that data. There are two primary approaches and we will discuss both of them here.

6.6.1 Reject-Known-Bad

Reject-Known-Bad is a type of input validation in which the app will be for inputs that contain data considered bad, and reject them. For example, when it comes to SQL injection that people discussed before, the exploit input worked since it contained a ' that terminates a string in SQL. Somebody who is application were to see all submitted input and reject any input that included such a character, it would be implementing a Reject Known-Bad input validation strategy. Along similar lines, if the developer were to write down an HTTP form submission routine that processed a comment into a website, he may want to find inputs which include script tags and reject any such inputs. Trying to find known-exploit inputs in this way is often a staple of Reject-Known-Bad input validation.

Such techniques are weak. This really is largely considering that the pair of possibly bad inputs is infinite. As each new attack technique or variation is discovered, their email list of known-bad inputs will need to be updated. This blacklist would quickly be unsustainable, as each developer might need to keep their apps’ input validation routines up-to-date with each new discovery. Additionally, examining each input listed in a credit card application for every single known-bad string would require extensive processing of every input string and this would decelerate this application considerably. This is simply not a sustainable, nor desirable, technique for input validation.

6.6.2 Accept-Known-Good

A better technique for input validation is Accept-Known-Good, where a credit card application actively seeks inputs which it does expect and rejects all the inputs that don't fit that specification. It is sometimes known as positive validation or white listing. Here, each input is checked for a number of criteria. Only received data meets all of these criteria if it is accepted and processed. If it fails any of these criteria, it must be rejected and discarded. Another essential concept in
input validation is sanitization. This refers back to the concept of changing a port into a satisfactory and safe format as an alternative to just choosing to accept or reject it. Just as in the accept-or-reject approach, sanitization can be achieved using either a white list or blacklist approach; identical evaluation of these two approaches as well as the clear superiority with the accept-known-good solution is true of sanitization as much as it does with the accept/reject approach.

This introduces another question. If illegal input keeps coming in, appropriately do the programmer allow the user (or no matter what application is communicating auction web sites the network) resubmit it, then when does one terminate the approval? It is another decision that should be made according to risk analysis. The programmer doesn’t want to let illegal input keep being retried, but the programmer doesn’t wish to severed legitimate users who're just messing up either. The harder sensitive the info being processed with the application, the quicker you'll want to stop accepting input if each attempt is prohibited.

Input validation is vital portion of application security. Applications make use of inputs, data that arrives externally in the application; they are driving actions and direct the approval how to proceed. On the list of primary problems in applications is always that developer’s trust that input. Do not forget that any piece of web data submitted externally the app might be submitted by, or manipulated by, an attacker. You'll want to verify that the data received because of the application is valid and safe to process. We now have established that the second approach is vastly superior.

6.6.3 Avoiding Command Injection

Even as have noticed, input validation is really a critical part of protecting any application from attack. However, once we also discussed recently, it will be important however, not sufficient. Input validation only verifies that inputs are the opinion they should be. There might be situations (indeed, there often are) where valid inputs are dangerous when used in certain ways. To return to the example we used previously, suppose we accept a person’s name as input. To correctly allow certain names, including many Irish names, we have to allow apostrophes such inputs. However, if we use such inputs in order to create SQL statements, were exposing the application to SQL injection attacks. This happens because after we simply combine
the command portions the statements using the data portions using string concatenation, it's impossible for the database to find out that is command and that's code, so data may be misinterpreted as code, leading towards attacks. Luckily, there is a strategy to separate the commands on the data when interacting with the database. Hook alteration to the way we write database statements can prevent SQL injection attacks.

Investigating an illustration exactly like the earlier one, let's consider researching a person's last name in a very database. The unsafe strategy to form this statement looks similar to this:

```java
SQLiteDatabase db = DBHelper.getWritableDatabase();

String userQuery = "SELECT lastName FROM useraccounts WHERE userID = "+ request.getParameter("userID");

SQLiteStatement prepStatement = db.compileStatement(userQuery);

String userLastname = prepStatement.simpleQueryForString();
```

With this example, we first form the SQL query that is to be shipped to the database by concatenating command instructions with data obtained from the network request. This can be unsafe as we are not aware of what are the network request contains and also if we have performed input validation at this point (which we absolutely should have), dangerous characters can be section of the white list. Hence the input could be dangerous for the programmer to a SQL-based database this way. That which the programmer requires is a method to separate the command from the data. Here's the most convenient way to accomplish such a query against the database, where the command is separated through the data:

```java
SQLiteDatabase db = DBHelper.getWritableDatabase();

String userQuery = "SELECT lastName FROM useraccounts WHERE userID = ?";

SQLiteStatement prepStatement = db.compileStatement(userQuery);

prepStatement.bindString(1, request.getParameter("userID"));

String userLastname = prepStatement.simpleQueryForString();
```
Through advantage of the compile Statement capability, we are able to effectively separate commands from data in SQL statements, by using the? Marker. This is what's called a parameterized query, because the query string includes placeholders (question marks) to mark the info along with the values for those components of data are filled in. Once we employ this capability, the database can separate command from data. Get the job done user ID input were to contain an apostrophe, the database knows it's the main data and address it as data, seeking ends up with the database which include that character, rather than treating it as section of the command, which may give rise to a SQL injection attack. Typically, when the application is interacting with a database, the programmer should utilize parameterized queries. A SQL statement must not be formed by concatenating command and data together.

Any interactions with SQL-based databases will most likely always use parameterized queries. The programmer must never form the firm stand out by concatenating command and data together, considering that the database couldn't survive able to differentiate relating to the two.

### 6.6.4 Why Android as a Case

This chapter and major references in this research were made to application security in android and iOS, because iOS was the major smart phone OS that took upon the legacy of Symbian and BlackBerry OS and brought it to the generation where smartphones are not only for the geeks and the rich but for general public too, and android’s growth is phenomenal. In a very short time span, it has succeeded in becoming one of the top mobile platforms out there. Clearly, the combination of open source licensing and aggressive go-to-market and classy interface is bearing fruit for Google’s Android team. Obviously, the huge user uptake generated by Android has not gone unnoticed by handset manufacturers, mobile network operators, silicon manufacturers, and app developers. Products, apps, and devices “for,” “appropriate for,” or “according to” Android appear to be released positively fast.

Beyond its mobile success, however, Android is usually attracting the attention of yet another, unintended crowd: embedded systems developers. While quite a few embedded devices have virtually no human interface, a considerable amount of devices that would traditionally be regarded “embedded” do have user interfaces. For
a goodly number of modern machines, in addition to pure technical functionality, developers creating user-facing devices must handle human-computer interaction (HCI) factors. Therefore, designers must either present user through an experience they already are knowledgeable about or risk alienating users by requiring them to practice a lesser known or entirely new buyer experience. Before Android, individual interface choices available to the developers of such devices were fairly limited and limiting.

Clearly, embedded developers prefer to offer users an interface these are already familiar with. Although that interface might have been window-located in the past and hence a wide range of embedded devices were according to classic window-centric, desktop-like, or desktop based interfaces. Apple’s iOS and Google’s Android have forever democratized the employment of touch-based, like graphical interfaces. This transfer of user paradigms and expectations, along with Android’s open source licensing; have created a groundswell of great interest about Android inside the embedded world.

The Android architecture will depend on little while-tested components, as well as the Linux kernel it uses is a vital isolation presented to applications. Android, however, is also an empty platform which is trivial to get root-level having access to an Android device, less difficult than on another major mobile platform. Usually do not ignore this consideration; ensure the programmer understands fully the environment the applications will likely be running in. In order to store some sensitive data, guarantee that it really is protected at a level appropriate for raise the risk involved.