4.

- In Android, each application (apk) must be signed with a certificate that is generated using the developer's private key.
- This certificate identifies the developer of the application, and has the purpose to distinguish between application authors.
- It is not mandatory for this certificate to be signed by a certificate authority; for Android applications, you can use self-signed certificates.
- The system applications are the ones that are signed with the same key as the AOSP, called platform key.
- An application can be updated only if the certificate matches with the one of the installed application.

6.

- Android gives each application, at install time, a unique user ID that is available in that moment.
- The data of an application can be accessed only by that specific application because of the Linux access rights of those files.
- Two applications can have the same UID, called shared UID, if you specify the sharedUserId attribute of the <manifest> tag in the manifest file of each package. This way, the two apps are treated as the same application, with the same UID and file permissions. The two apps must be signed with the same private key.
- You can also share the files of an application by using the flags MODE_WORLD_READABLE or MODE_WORLD_WRITABLE when creating a new file. This will allow other applications to read or write your files.

8.

- A central design point of the Android security architecture is that no application, by default, has permission to perform any operation that would adversely impact other applications, the operating system, or the user.
- In Android, a permission is a string that signifies the ability to perform a certain operation.
- Android comes with a set of predefined permissions (called built-in). These are documented online on the platform API reference and are defined in the android package.
- In addition, both system and user applications can define additional permissions, called custom.
- A list of all permissions known by the system in a certain moment can be obtained by using the command pm list permission in the shell.
- The name of a permission includes the package that defines it plus the string .permission, plus the actual name.

- Here we have 2 examples, the first one is a built-in permission defined in the android package and the second one is a custom permission defined by the launcher application.

9.

- Applications will be able to request permissions by specifying them in the manifest file.
- Here you have an example: the tag is called <uses-permission>. We will see later that custom permissions are also declared in the manifest file.
- Permissions are handled by the PackageManager service, which maintains a centralized repository of information about the installed packages, stored in /data/system/packages.xml.
- In order to obtain programatically more information about an installed package, we can use the method getPackageInfo() from android.content.pm.PackageManager. This method returns an instance of PackageInfo, which encapsulates all information from packages.xml file about that package.
- Until Android version 5.1, permissions are granted at installation time, and cannot be modified or revoked afterwards.
- From Android version 6.0, the applications request permissions from the user at run-time. So the user can revoke permissions at any time.

10.

A permission may be enforced at a number of places during your program's operation:

- When making a call into the system, to prevent an application from executing certain methods
- When starting an activity, to prevent applications from launching activities of other applications.
- Binding to a service or starting a service.
- Both sending and receiving broadcasts, to control who can receive your broadcast or who can send a broadcast to you.
- When accessing and operating on a content provider.

11.

- The protection level of a permission indicates the potential risk but also the method used by the system to decide whether to grant that permission or not.
- In Android, we have 4 types of protection levels: normal, dangerous, signature, and signatureOrSystem.
- The normal protection level defines a permission with a low risk for the system and for other applications. Permissions with this protection level are granted automatically to applications, without requiring the user’s approval. For example: ACCESS_NETWORK_STATE, GET_ACCOUNTS.
- Permissions with the dangerous protection level give applications access to the user’s data or some control over the device. In this case, Android will display information about

the permissions requested by the application at install time and the user will be able to accept the permissions or cancel the installation (in Android 5.1) and at runtime in Android 6.0. Examples: CAMERA, READ_SMS.

12.

- Permissions with the signature protection level are granted to applications only if they are signed with the same key as the application that declared that permission. This is the highest level of protection. This is why built-in permissions are used only by system applications (the ones that are signed with the platform key). Examples: NET_ADMIN, ACCESS_ALL_EXTERNAL_STORAGE.
- Permissions with the protection level signatureOrSystem are a compromise: they are granted to applications that are part of the system image or signed with the same key as the application that declared that permission. This allows vendors to have their own preinstalled applications without using the platform key. They are installed in /system/priv-app/ in Android 4.4.

13.

- All dangerous Android system permissions belong to permission groups. Practically any permission can belong to a permission group, but only dangerous permission groups affect user experience. Let's see why.
- On Android 5.1 or less, the system asks the user to grant permissions at install time. But it asks for the entire permission group, not for for individual permissions.
- On Android 6.0, if an app requests a dangerous permission listed in its manifest, and the app does not currently have any permissions in the permission group, the system shows a dialog box to the user describing the permission group that the app wants access to.
- If an app requests a dangerous permission in its manifest, and the app already has another dangerous permission in the same permission group, the system immediately grants the permission without any interaction with the user.
- Examples of dangerous permission groups are: Calendar, Camera, Contacts, Location, Phone, SMS, Sensors, Storage, Microphone.

14.

- Access to regular files, devices and sockets is handled by the Linux kernel, based on the UID and GID.
- Some permissions are associated to certain supplementary GIDs, which are verified when low-level operations are performed.
- The mappings for the built-in permissions can be found in the /etc/permission/platform.xml file.
- For example, the INTERNET permission is mapped to the GID called inet. An application cannot create network sockets without having the INTERNET permission. When the application wants to create a socket, the kernel will verify whether the application is part of the inet group.

15.

- At framework level, we have two types of permission enforcement: static and dynamic.
- What does static enforcement mean? The system keeps information about the permissions associated with every component and verifies if the caller process has enough permissions before allowing access.
- Static enforcement is done by the runtime environment.
- The advantage is that it separates the security decisions from the business logic. However, this method is less flexible than the dynamic enforcement.
- Components can also verify if the caller has the necessary permissions, even if they are not declared in the manifest. So the dynamic enforcement is done by every component instead of the runtime environment.
- The advantage is that we have a more granular control over permissions, and the disadvantage is that we make additional operations in the components.

16.

- Let's see how dynamic enforcement is performed.
- Android provides a series of methods for verifying permissions in the class android.content.Context.
- Here we have two examples:
- checkPermission(String permission, int pid, int uid):
    - Will return PERMISSION_GRANTED if the process with that uid has the necessary permission (the string) and PERMISSION_DENIED otherwise.
    - If the caller is root or system, the permission is granted automatically.
    - If the permission is declared by the caller application, the permission is granted without further verifications.
    - If the destination component is private, then access is denied.
    - Otherwise, it will call PackageManager in order to find out whether the caller has the necessary permission or not.
- Another method is enforcePermission(String permission, int pid, int uid, String message), which performs similar operations with the previous one, but will generate a security exception with the specified message if the permission is not granted.

17.

- Static enforcement is performed when an application tries to interact with a component declared by other application, through an intent.
- The target component must specify the needed permission using the android:permission attribute and the caller must specify the permission through the <uses-permission> tag, in the manifest file.
- The verification is done by the ActivityManager, which resolves the intent and verifies whether the destination component has the associated permission or not. It delegates permission verification to the PackageManager.
- If the caller has the necessary permission, ActivityManager will start the destination component. Otherwise, it will throw a SecurityException.

18.

- Permission verification for activities is done when an intent is given to startActivity() or startActivityForResult() that is resolved into an activity that declares a permission.
- Permission verification for services is done when an intent is given to startService(), stopService() or bindService() that is resolved into a service that declares a permission.
- If the caller does not have that permission, a SecurityException will be thrown.

19.

- Content provider permissions protect the entire component or a certain exported URI.
- There are different permissions for reading and writing.
- The permission for reading controls who is able to call ContentResolver.query() on a certain provider or URI.
- The permission for writing controls who is able to call ContentResolver.insert(), update() or delete() on a certain provider or URI.
- These verifications take place synchronously when one of these methods is called.

20.

- When a broadcast is sent, the receivers may need a certain permission in order to receive the broadcast message. The sender can specify the permission when calling Context.sendBroadcast(Intent intent, String receiverPermission). If the receiver does not have that permission, it will not get the message, but an exception will not be thrown.
- In addition, it may be necessary for the sender to have a certain permission in order to send a broadcast message. This permission is also verified when the delivery is made, but will not throw an exception if the permission is missing.
- In conclusion, we may have 2 permission verifications when broadcast messages are delivered: one for the sender and one for the receiver.

21.

- Custom permissions are the ones declared by third party applications.
- After they are declared, they can be used in application components, in order to be statically enforced by the system, or the application itself can verify if the caller has the necessary permission.
- A custom permission is declared in the manifest file.
- Here you have an example. First, a permission tree is created, then a permission group, and finally the permission itself that belongs to that group.
- For the permission, we set the protection level, which in this case is signature. This means that it can be granted only to applications signed with the same key as the application that declares the permission.
- The protection level is mandatory, while the permission group is optional. It is recommended to use an existing permission group, because it simplifies the UI shown to the user.

23.

- Java Cryptography Architecture (JCA) provides an extensible framework for cryptographic providers and a set of APIs for accessing cryptographic primitives.
- Applications that use JCA only need to ask for a certain algorithm, without specifying a certain provider. The framework will find the provider that implements the algorithm.
- Cryptographic Service Provider (CSP) is a package that includes the implementation of a set of services and cryptographic algorithms. Each provider will notify JCA about the implemented services and algorithms.
- JCA will handle a register of providers and the algorithms implemented by them. In this register, providers will be ordered by preference. If two providers implement the same algorithm, the one with the highest preference (the lowest number) will be selected.
- Service Provider Interface (SPI) is a common interface that must be taken into consideration by all providers that implement a certain algorithm.
- More exactly, it is an abstract class that is implemented by the providers that include a certain algorithm.

24.

- JCA Engines provides one of the following services:
    - Cryptographic operations (encrypt, decrypt, sign, verify signature, hashing)
    - Generate and convert cryptographic materials (keys or algorithm parameters)
    - Handle and store cryptographic objects (keys, digital signatures)
- Engine classes separate the client code from the algorithm implementation, therefore, they cannot be instantiated directly.
- However, they provide a static factory method called getInstance(). Through this method, we will request the implementation of a service indirectly.
- We have here 3 method signatures of getInstance(). The first one is the most used, it specifies just the name of the algorithm and JCA will identify the provider with the highest priority that implements the algorithm.
- In the second format, it also requests a certain provider, by specifying the name of the provider in string format. The third gives as argument an instance of the requested provider.
- All may throw NoSuchAlgorithmException, and the second one can also throw NoSuchProviderException in case it does not find that provider.

25.

- The class MessageDigest is used for obtaining a simple hash function.
- Here we have an example. First of all, an instance of the class MessageDigest is obtained by using the factory method getInstance() and the algorithm SHA-256.
- Then we use the method digest() and give it an array of bytes, in order to obtain the hash.

- There are two methods to give the data: if we have large data, we may give it piece by piece through the update() method. In the end, we call digest() without any argument. If we have small data of fixed dimension, we can call digest() directly with the data as argument.

26.

- Class Signature provides an interface for the digital signature algorithms based on asymmetric encryption.
- The name of the algorithm is usually <digest>with<encryption>, where digest is the name of a hashing algorithm and encryption is the name of an asymmetric encryption algorithm.
- Here we have an example of how to create a digital signature. First of all, we obtain an instance of Signature through the getInstance() method and the name of the algorithm SHA256withRSA.
- Then, we set the private key that will be used for signing through the method initSign(). Then we give the data through the method update() and finally we obtain the signature through the method sign().
- Then, we have an example of digital signature verification. First, we obtain the instance, then we configure the public key that is used for verifying the signature through the method initVerify(). Then we give the data through the update() method. And the signature is validated using the verify() method.

27.

- The Cipher class provides a common interface for encrypting and decrypting.
- Here we have an example for encryption. First of all, we obtain a Cipher instance that uses the AES encryption algorithm, the CBC operation mode and PKCS#5 padding.
- Then we generate a random Initialization Vector and put it in a IvParameterSpec object.
- We initialize the Cipher using the method init() with the flag ENCYPT_MODE, an encryption key and an IV.
- Then we give pieces of data through the update() method which returns intermediate results (also returns null if the data is too short for a block).
- We obtain the last block using the method doFinal().
- The whole ciphertext is obtained by concatenating the intermediate results with the final block.

28.

- For decryption, we obtain the Cipher instance, initialize the Cipher using the init() method, the DECRYPT_MODE flag, the key and the IV.
- Then we give pieces of the ciphertext using the method update() and call doFinal() in order to obtain the last piece of data.
- The final plaintext is obtained by concatenating the intermediate results with the last piece of data.

29.

- The Mac class provides a common interface for Message Authentication Code algorithms.
- Here we have an example. First of all, we obtain a Mac instance by using the method getInstance() in which we specify the algorithm HMAC that uses SHA256 as hashing method.
- Then, we initialize it with the method init() and the secret key.
- In the end, we call doFinal() to obtain the value of the MAC based on the data.
- We could have also given pieces of data through the update() method and then call doFinal().

30.

- The KeyGenerator class is used for generating symmetric keys (that are used in symmetrical encryption and MAC algorithms).
- It is better than SecureRandom because it performs additional verifications for weak keys and can set parity bytes when necessary (for DES). In addition, it can use cryptographic hardware when it is available.
- Here we have 2 examples. In the first example we obtain an instance of KeyGenerator for the algorithm HmacSha256 and then we generate the key using the method generateKey. It will give us directly a key on 256 bits for that algorithm.
- In the second example, we obtain an instance of KeyGenerator for the AES algorithm. We give the key dimension (256, because AES works with 3 dimensions of keys). And finally, we generate the key.

31.

- The KeyPairGenerator class is used for generating public and private key pairs (for asymmetric algorithms).
- Here we have an example. We obtain the KeyPairGenerator instance for RSA algorithm. Then we give it the key length. Then we obtain a KeyPair object through the method generateKeyPair(). Finally, we obtain the keys through the methods getPrivate() and getPublic()

32.

- Let's see some providers that can be used in Android.
- Harmony's Crypto Provider is a JCA provider with a small set of functionalities, that is included in the Java runtime library. It includes just 4 algorithms, for SecureRandom, KeyFactory, MessageDigest and Signature.
- Android's Bouncy Castle Provider is a JCA provider with a very large set of algorithms and services, that is part of the Bouncy Castle Crypto API. We have many algorithms for each class: Cipher, KeyGenerator, Mac, MessageDigest, SecretKeyFactory, Signature, CerticateFactory, etc.

- AndroidOpenSSL Provider is implemented in native code for performance reasons. It has many functionalities, covering most of the algorithms and services offered by Bouncy Castle. Practically, it uses JNI in order to access the OpenSSL library. This provider is the default one, having the biggest priority, 1.