PassQueen
a.k.a dumpsqueen.com

Android Security Essentials

Android AND-802

Version Demo

Total Demo Questions: 7

Total Premium Questions: 73

Buy Premium PDF

https://passqueen.com

support@passqueen.com

QUESTION NO: 1

What is the purpose of applying `FLAG_SECURE` to an Android activity window?

A. It prevents screenshots and non-secure display of the window.

B. It encrypts all files written by the activity.

C. It grants the activity access to secure device settings.

D. It prevents the activity from being stopped in the background.

ANSWER: A

Explanation:

It prevents screenshots and non-secure displays of the activity window. `FLAG_SECURE` tells Android to treat the window content as secure. The system prevents the window from appearing in screenshots and prevents it from being displayed on non-secure displays, such as certain casting or remote-display paths.

This flag is useful for screens that show sensitive information, including payment details, authentication material, or private documents. It is a defense-in-depth control and does not prevent a user from photographing the screen with another device. See the WindowManager.LayoutParams.FLAG_SECURE API reference.

QUESTION NO: 2

Which of the following are appropriate uses of a signature-level custom permission? (Select three)

A. Restricting a sensitive service to companion apps signed with the same certificate.

B. Protecting a provider shared by a suite of apps from one organization.

C. Limiting access to a privileged internal activity.

D. Allowing every third-party app to use a public feature.

E. Granting access to the device camera at runtime.

ANSWER: A B C

Explanation:

Restricting a sensitive service to companion apps signed with the same certificate is correct because signature permissions are granted only to applications signed with the same certificate as the permission-defining application. **Protecting a provider containing data shared by a suite of apps from one organization** is correct for the same reason. **Limiting access to a privileged internal activity** is also correct when the activity must be exported for trusted companion applications but should not be available to unrelated apps.

A signature permission is not suitable when arbitrary third-party applications must access the feature, because those apps will not share the signing certificate. It also does not grant runtime access to camera hardware; the CAMERA permission is a platform-defined dangerous permission. Custom permissions are declared in the manifest with an appropriate protection level and then applied to the components they protect. See the Android <permission> manifest element documentation and Android permissions overview.

QUESTION NO: 3

You must use permissions to prevent unauthorized access to your application data.

A. False

B. True

ANSWER: A

Explanation:

False is correct because Android does not require an app to declare or enforce permissions merely to protect its own non-exported application data. Android's application sandbox assigns each installed app a distinct Linux user ID and isolates its private files, databases, preferences, and internal storage from other apps by default. Consequently, an app can prevent other applications from accessing data simply by keeping its components non-exported and storing sensitive information in app-private storage. Permissions become important when an app deliberately exposes a component, provider, service, receiver, or other resource to external apps and needs to control which callers may access that exposed interface. The security objective is therefore achieved through the platform's layered application sandbox and appropriate component/storage configuration; permissions are an additional access-control mechanism where cross-app access is intended. Android's security overview describes the per-app sandbox model, while its storage guidance explains that internal storage is private to the application. See Android Application Sandbox and App-specific storage documentation.

QUESTION NO: 4

In the following image of code, what is the purpose of using MODE_PRIVATE in the method getPreferences( )?

```
class MainActivity : AppCompatActivity() {

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)
        save.setOnClickListener({
            savePreferences("save", enter_data.text.toString())
            show.isEnabled = true
        })
        show.setOnClickListener({
            load_data.setText(loadPreferences())
        })
        show.isEnabled = false

        /*
        * If there is no data currently stored then the
        * show button will be shown as disabled when
        * application is launched.
        */

        if (loadPreferences().isNotEmpty())
            show.isEnabled = true
    }

    private fun savePreferences(key: String, value: String) {

        val sharedPreferences = getPreferences(MODE_PRIVATE)
        val editor = sharedPreferences.edit()
        editor.putString(key, value)
editor.commit()
    }

    private fun loadPreferences(): String {
        val sharedPreferences = getPreferences(MODE_PRIVATE)
        val savedData = sharedPreferences.getString("save", "")
        return savedData
    }
}
```

A. This allows you to store data and make it private to this application. Also, No other applications can access your saved data.

B. This allows you to share your data with other applications which have the same mode.

C. This allows you to run your apps in Android API 27 or higher.

D. This allows you to run your app with minimum use of device battery.

ANSWER: A

Explanation:

`MODE_PRIVATE` opens or creates the activity's default `SharedPreferences` file in private mode. This is the standard mode used when preference values, such as settings, flags, or small pieces of application state, are intended for use only within the application. The preference file is stored in the app's private storage area and is protected by Android's application

sandbox and Linux UID-based permissions. Consequently, unrelated applications cannot directly open the preference file and read or modify the saved values through normal Android storage access.

The mode is supplied to `getPreferences()`, which is a convenience method for obtaining a `SharedPreferences` file associated with the current activity. Android documents `MODE_PRIVATE` as the default file-creation mode, where the created file is private to the calling application. This makes it appropriate for ordinary app preferences that do not need to be exposed outside the app. Developers should still avoid storing highly sensitive secrets in ordinary preferences without additional protections, because private storage is an access-control boundary rather than a substitute for secure secret handling.

See the Android Context.MODE_PRIVATE reference and the SharedPreferences documentation.

QUESTION NO: 5

Which of the following choices represent the flow of states of data within any app? (Select three)

A. Stored data

B. Data under processing

C. Data in transit

D. Printed Data

ANSWER: A B C

Explanation:

Applications handle information in three core states: stored data, data under processing, and data in transit. Stored data is information at rest, such as files, databases, preferences, caches, or credentials retained on a device. Android apps should protect this state through appropriate private storage, encryption where warranted, and careful control of backups and exported components. Data under processing is information in use while the app reads, transforms, displays, or otherwise handles it in memory. This state requires minimizing exposure of sensitive values and avoiding unnecessary retention or logging. Data in transit is information moving between app components, processes, devices, or remote services; it requires secure communication protections such as TLS and properly configured network security controls. Together, these states describe the complete data lifecycle that an app must secure: at rest, in use, and in motion. Android's security guidance discusses protecting app data and secure network communications, including the use of network security configuration. See Android security best practices and Network security configuration.

QUESTION NO: 6

Which of the following choices is considered Android application storage? (Select five)

A. SharedPreferences

B. Android's file system

C. External Storage

D. Cache

E. SQLite database

F. Microsoft Access

G. Text file

ANSWER: A B C D E

Explanation:

Android applications can persist data through several platform-supported storage mechanisms, selected according to the data's structure, sensitivity, size, and expected lifetime. **SharedPreferences** is application storage for small key-value data such as settings and flags; in modern Android development, Jetpack DataStore is commonly recommended as its successor. **Android's file system** represents app-specific internal storage, where an app can create and read files that are private to the app by default. **External Storage** provides locations for app-specific files and, where appropriate, user-visible shared media or documents, subject to Android's storage-access model. **Cache** is app-managed temporary file storage used for data that can be recreated or fetched again; the system may remove cached files when space is needed. **SQLite database** is the platform's traditional relational database storage facility, typically accessed directly through SQLite APIs or through the Room persistence library. Together, these mechanisms cover Android's standard preferences, files, temporary data, external/app-specific files, and structured relational-data use cases. Android's official storage overview describes internal storage, external storage, preferences, and databases at Android data and file storage overview. Guidance for temporary cache files is available at App-specific storage.

QUESTION NO: 7

getExternalFilesDir( ) method is used to get the directory of the external storage.

A. True

B. False

ANSWER: A

Explanation:

`True` is correct because `Context.getExternalFilesDir(String type)` obtains the path to an app-specific directory on shared/external storage where the application can store its files. With `null` supplied as the type, it returns the root of that app-specific external-files area; a type such as `Environment.DIRECTORY_DOCUMENTS` returns the corresponding categorized subdirectory. These locations are intended for files owned by the app, and Android removes them when the app is uninstalled. For modern Android versions, app-specific external storage is especially useful because the app can access its own external-files directory without requesting broad shared-storage access. The method may return `null` when the relevant external storage volume is unavailable, so production code should check the result and storage state before writing. The statement correctly identifies the method's purpose, with the important practical understanding that it refers specifically to the application's directory on external/shared storage rather than the top-level external-storage root. Android documents this API at Context.getExternalFilesDir and describes app-specific external storage in the Android data storage guide.