Overview

- More transparency into users' private data access and causes of process exits
- New APIs introduced in Android 11

Data Access Auditing APIs

- A callback allowing apps to **back trace the use of data protected by runtime permission** to the code that triggered the usage
- AppOpsManager provides a callback
    1. This callback will be invoked every time when the code uses private data. (Location Updates)

```
override fun onCreate(savedInstanceState: Bundle?) {
    val appOpsCallback = object : AppOpsManager.OnOpNotedCallback() {
        private fun logPrivateDataAccess(opCode: String, trace: String) {
            Log.i(MY_APP_TAG, "Private data accessed. " +
                    "Operation: $opCode\nStack Trace:\n$trace")
        }

        override fun onNoted(syncNotedAppOp: SyncNotedAppOp) {
            logPrivateDataAccess(
                    syncNotedAppOp.op, Throwable().stackTrace.toString())
        }

        override fun onSelfNoted(syncNotedAppOp: SyncNotedAppOp) {
            logPrivateDataAccess(
                    syncNotedAppOp.op, Throwable().stackTrace.toString())
        }

        override fun onAsyncNoted(asyncNotedAppOp: AsyncNotedAppOp) {
            logPrivateDataAccess(asyncNotedAppOp.op, asyncNotedAppOp.message)
        }
    }

    val appOpsManager =
            getSystemService(AppOpsManager::class.java) as AppOpsManager
    appOpsManager.setOnOpNotedCallback(mainExecutor, appOpsCallback)
}
```

*onAsyncNoted() : called if the data access doesn't happen during your app's API call, rather when your app registers a listener and the data access happens each time the listener's callback is invoked.

*onSelfNoted(): very rare case when app app passes its own UID into noteOp()
*noteOp(): returns int, and parameters are op(String), uid(Int), packageName(String), attributionTag(String), message(String)

- In a multipurpose app, like social media, Users can create a new Context object that allows them to attribute a subset of their app's code to one or more features.

- Using this tag helps users trace what each part of the code accesses to.

```
class SharePhotoLocationActivity : AppCompatActivity() {
    lateinit var attributionContext: Context

    override fun onCreate(savedInstanceState: Bundle?) {
        attributionContext = createAttributionContext("sharePhotos")
    }

    fun getLocation() {
        val locationManager = attributionContext.getSystemService(
                LocationManager::class.java) as LocationManager
        // Use "locationManager" to access device location information.
    }
}
```

- Therefore, every permission usage would be traced to the features associated with the context.

Process Exit Reasons

- Difficult tracking down the cause of termination due to the various reasons (ANR, Crash, or forcing to stop the app)
- To tackle this tough situation, there have been several solutions to diagnose and report the reason, such as Crashlytics
- New **ActivityManager** in Android 11, **to report historical information** related to an app process's termination
    - ApplicationExitInfo.getTraceInputStream()
        - Returns **an InputStream to the stack trace dump of the app** prior to the termination.
        - More helpful on newer OS, because of the complexity due to the privacy and security considerations
        - Don't forget to close that stream after use to avoid leaks
    - ActivityManger.setProcessStateSummary()
        - To store custom state information
        - Useful way to save arbitrary process data to debug.
        - Takes ByteArray as a parameter, and the size is limited.
        - In a case that the current state is important (like a game), can save the current state when a crash happens.
    - ActivityManager.getProcessStateSummary()
        - A method to retrieve the stored state information

Resources

1. https://developer.android.com/preview/privacy/data-access-auditing
2. https://developer.android.com/preview/features#app-process-exit-reasons
3. https://github.com/android/permissions-samples