
ESAME DI PROGRAMMAZIONE DI SISTEMI EMBEDDED

9 CFU

***Tema:* Designing For Privacy & Security**

Avanzi Giacomo, 1223262

Lazzarin Jacopo, 1220993

Segalin Elena, 1225243

INTRODUZIONE

Negli ultimi decenni il progresso scientifico e tecnologico ha raggiunto un grande sviluppo, favorendo una rapida diffusione delle nuove tecnologie nella vita quotidiana delle persone, tra le quali spiccano i mezzi di comunicazione come smartphone e tablet.

Questi dispositivi mobili ci accompagnano ovunque durante le giornate e contengono numerose informazioni sensibili riguardanti il proprietario, a partire dai semplici dati personali, fino ad arrivare alle nuove frontiere dei dati, ad esempio relative alla geolocalizzazione, alle credenziali elettroniche, ai dati biometrici, etc. Pertanto, se da un lato si è assistito all'avanzamento delle tecnologie digitali, dall'altro si è sviluppata una maggiore consapevolezza del diritto alla riservatezza e alla protezione dei dati conservati nei dispositivi.

Il termine riservatezza in generale comprende la sfera privata dell' individuo e il diritto alla protezione dei dati personali che lo riguardano, cioè qualunque informazione associata a un numero di identificazione personale.

Il diritto alla riservatezza quindi tutela quella che è la propria intimità, perciò è da intendersi in un senso molto ampio.

Nel linguaggio giuridico si usa l'espressione "protezione dei dati personali", mentre nel linguaggio corrente ha preso piede l'uso del termine "privacy". Questi, rispetto alla riservatezza, fanno riferimento maggiormente alla tutela della sfera privata e delle libertà connesse. Il termine privacy non è più sinonimo del diritto a non essere disturbati, ma è evoluto coinvolgendo il trattamento dei dati personali e le libertà correlate.

Il progresso tecnologico ha messo in luce come l'espressione "protezione dei dati personali" sia fortemente legata alle tecniche di tutela per la sicurezza e protezione dei sistemi e dei dati personali, e in questo l'informatica gioca un ruolo fondamentale.

In ambito informatico il termine riservatezza indica una delle tre principali caratteristiche della sicurezza, insieme alla disponibilità e all'integrità dei dati.

Questi sono gli aspetti su cui prestare attenzione per garantire una adeguata gestione della security.

Ognuno di questi aspetti è implementato per uno scopo preciso:

- Riservatezza: impedire l'accesso e la fruizione di dati a un utente non autorizzato;
- Disponibilità dei dati: garantire che una risorsa sia sempre disponibile e funzionante;
- Integrità dei dati: garantire che i dati non subiscano modifiche o cancellazioni indesiderate a causa di errori, malfunzionamenti, danni dei sistemi tecnologici o di azioni volontarie.

In generale ci sono varie azioni che gli utenti possono attuare per proteggere i propri dati, in ambito privato e lavorativo, su i dispositivi come pc, tablet e smartphone.

È necessario quindi prestare attenzione e seguire dei semplici accorgimenti per proteggere il proprio patrimonio informativo:

- sapere quali sono i dati da proteggere e dove sono;
- formare i dipendenti / utenti;
- sapere chi ha accesso ai dati;
- valutare i rischi;
- installare software di protezione dei dati ed eseguire scansioni frequenti;
- eseguire regolarmente backup dei dati più sensibili.

Bisogna ammettere però che, sfortunatamente, i dispositivi mobili stanno diventando molto più vulnerabili, rispetto a qualche anno fa, agli attacchi e alla violazione dei dati a causa dell'accesso wireless a internet sempre disponibile.

L'autenticazione e l'autorizzazione tramite dispositivi mobili diventano comode, ma aumentano il rischio, in quanto non sono presenti vincoli di protezione, che invece sono presenti ad esempio all'interno di una rete aziendale.

Le attuali funzionalità, quali giroscopi, accelerometri, GPS, microfoni e telecamere, cambiano il metodo di autenticazione degli utenti e il modo in cui le autorizzazioni vengono fornite localmente al dispositivo, alle applicazioni e ai servizi sulla rete. Una diretta conseguenza è l'aumento del numero di endpoint che devono essere protetti da minacce.

In questo report si parlerà del sistema operativo Android. Si analizzeranno alcune delle principali problematiche e vulnerabilità legate alla sicurezza.

Si procederà poi con una spiegazione delle buone norme per la privacy, quegli aspetti che la documentazione di Android mette in evidenza e su cui un programmatore dovrebbe prestare attenzione per uno sviluppo sicuro e di tutela nei confronti dell'utente.

Infine verranno spiegate, da un punto di vista tecnico e implementativo, alcune funzionalità che Android mette a disposizione per ridurre i problemi legati alla sicurezza e per aiutare il programmatore nella progettazione, semplificando la gestione di situazioni complicate.

ANDROID: PROBLEMI DI SICUREZZA

Android ha già guadagnato da anni un vantaggio significativo in termini di mercato, grazie alle numerose funzionalità che offre, una tra queste, in particolare, essere una piattaforma mobile open-source che lo rende gratuito agli utenti e personalizzabile in molti aspetti.

Ma se da un lato questo è stato il motivo di maggior fortuna, d'altro canto la natura così aperta di Android e la larga diffusione l'ha resa una piattaforma attraente e redditizia da attaccare. Ci sono exploits e strumenti che possono essere usati sui sistemi operativi di un vasto numero di dispositivi, in questo modo gli attaccanti possono sfruttare in massa delle falle di sicurezza e riutilizzare vettori d'attacco.

Google ha preso delle misure nello sviluppo del kernel di Android per garantire certi livelli di sicurezza: il sistema operativo sfrutta il meccanismo delle sandbox, isolando i diversi processi applicativi e riducendo di fatto il concetto di infezione.

Diverse sono le problematiche di sicurezza presenti all'interno di Android, che si traducono in vulnerabilità sfruttabili da malintenzionati per trafugare dati sensibili dell'utente.

Permessi delle applicazioni

Spesso gli utenti si vedono apparire dei pop up sullo schermo, che possono risultare noiosi, dannosi per l'esperienza di navigazione, a volte incomprensibili e senza spiegazione, i quali generano molto spesso un desiderio di sorvolare sopra e accettare tutto con superficialità. Tuttavia questi permessi vengono richiesti comunemente dalle app per accedere a funzionalità chiave, come la lettura/scrittura degli messaggi, l'accesso alla localizzazione, l'uso del microfono e fotocamere, l'accesso all'IMEI, etc. Allo stesso tempo i dati sensibili sbloccati da questi permessi pericolosi, potrebbero essere raccolti da app malevole che mirano a rubare informazioni personali, quali credenziali, contatti, e altro ancora.

Trasmissione di informazioni

Ciascuna applicazione viene eseguita in un processo separato, che possiede la propria istanza della DalvikVM, il sistema di default (cfr Linux) assegna a ciascuna app un ID univoco e permette l'accesso ai propri file solo con questo stesso codice. Le comunicazioni di dati tra processi di app virtualizzate sono gestite da un protocollo di scambio di messaggi, sotto forma di oggetti detti Intent. In particolare gli Intent impliciti, Intent con un'azione senza un destinatario fissato, possono lanciare componenti diversi che risolvono una stessa azione, di fatto Android lascia scegliere all'utente quale lanciare. Usando delle app malware, un attaccante può registrare la propria Activity con l'azione contenuta nell'Intent nel manifest e impostando una priorità molto alta del filtro per l'Intent. In questo modo nel box con tutte le applicazioni tra cui scegliere, l'app malevola risulta al primo posto. Se l'utente lancia proprio tale applicazione, i dati sensibili trasportati dall'Intent possono essere acceduti impropriamente, con il rischio ulteriore che tali informazioni possano essere falsificate e inoltrate (spoofing).

Play Store

Nell'utilizzo di uno smartphone è naturale installare applicazioni aggiuntive rispetto a quelle di sistema: app di messaggistica, social network, streaming audio, etc. Android mette a disposizione per questo scopo un unico grande market, che assicura una maggiore sicurezza delle app scaricate, grazie a Google Play Protect che controlla l'eventuale presenza di comportamenti dannosi nelle app e nei dispositivi. Esso verifica sul dispositivo la

presenza di app malevole, talvolta indicate con il termine malware, che violano le norme di Google: nascondono informazioni importanti, le presentano in modo fuorviante, accedono a permessi non coerenti, etc.

Nonostante questo filtraggio imponente eseguito alla fonte, può capitare che alcune applicazioni, in particolare quelle appena pubblicate, celino funzionalità dannose, che non lo sembrano in prima battuta ai controlli iniziali, ma poi con controlli più accurati emergono. Quindi è molto importante assicurarsi in fase di installazione sull'origine dell'applicazione, soprattutto sullo sviluppatore e sui permessi richiesti in base alle funzionalità offerte.

Applicazioni di terze parti

A causa della natura così aperta della piattaforma Android verso gli utenti e gli sviluppatori, il Play Store non è l'unica fonte disponibile da cui installare applicazioni: è sufficiente possedere un qualsiasi archivio d'installazione apk di qualche app e non è più necessario passare per lo store ufficiale. In questo modo viene bypassata una buona parte dei meccanismi di controllo della sicurezza messi a disposizione da Google, perciò bisogna essere consapevoli di questa maggiore esposizione e consentire nelle impostazioni l'installazione di app da fonti sconosciute.

Questo passo, se da un lato garantisce una maggiore libertà nell'usare applicazioni non disponibili nello store ufficiale, ipoteticamente anche appena sviluppate dal singolo utente, dall'altro aumenta il rischio d'infezione. Infatti insieme all'installazione di un programma, o anche al suo posto, le app hanno accesso ai dispositivi degli utenti e possono rubare dati personali.

Rooting e Malware

Il rooting è un processo attraverso cui gli utenti di un dispositivo Android possono ottenere i permessi di root sul sistema, superando così i limiti imposti dagli sviluppatori e dai tecnici hardware sul dispositivo, infattibile per un normale utente Android. Questo è possibile poiché Android è basato su un kernel Linux (Unix-like), quindi il rooting consente l'accesso ai permessi di superutente.

Tramite questa procedura l'utente ha il pieno controllo del dispositivo, può modificare le impostazioni di sistema, installare app che richiedono i permessi di amministratore, accedere ai file di sistema e alterare il sistema operativo. Tuttavia ci sono anche numerosi svantaggi per i dispositivi rootati: in primis l'aumento del rischio di hacking o di incorrere in virus, ma anche il rischio di corrompere irrimediabilmente il firmware o causare problemi hardware. In aggiunta i permessi di root sono un exploit comune usato dalle applicazioni malevoli per ottenere l'accesso completo al sistema.

Ad esempio DroidKungFu è un malware che colpì le versioni di Android 2.2, esso sfruttando due diversi exploits di root, "udev" e "RageAgainstTheCage", rompeva la sicurezza di Android e aveva pieno accesso al dispositivo. Infatti, una volta eseguito, decriptava gli exploits e iniziava a raccogliere dati sensibili come l'IMEI, la versione di Android, il modello del telefono, e molti altri, comunicandoli poi a un server remoto attraverso un HTTP POST, e in tutto questo l'utente non aveva la consapevolezza di nulla.

Accesso allo storage

Qualsiasi dispositivo Android può essere collegato a un PC, in questo modo, una volta concesso il permesso, viene fornito l'accesso in lettura e scrittura ai dati contenuti nella memoria interna e esterna, come una scheda SD. Anche se tutti i dispositivi di archiviazione esterna, essendo comunque rimovibili, possono essere acceduti direttamente.

Questi metodi di collegamento possono fungere da vettore d'attacco, ovvero usati per diffondere malware, trasferire di nascosto contenuti dannosi da/verso il pc o raccogliere informazioni private. In questo modo si possono leggere i dati in uso dalle applicazioni, come quelli che sono manipolati correntemente o quelli che sono stati manipolati di recente. Le informazioni sensibili trovate tra questi dati possono essere sfruttate dai malware in vario modo: ad esempio assumendo l'identità di una persona grazie al reperimento delle credenziali del proprio account.

Per proteggersi da questo fenomeno, Android fornisce strumenti di crittografia per codificare i dati in un formato indecifrabile per gli utenti senza le credenziali del dispositivo. Si tratta di un processo irreversibile, la crittazione dei dati può essere rimossa solo con un reset di fabbrica del dispositivo.

ANDROID: LINEE GUIDA PER LA PRIVACY

La piattaforma Android è in costante evoluzione nel tempo, vengono rilasciate continuamente nuove funzionalità per gli sviluppatori affinché agli utenti sia garantita un'esperienza ai massimi livelli. Allo stesso tempo, la privacy degli utenti è un principio imprescindibile e prioritario per Android.

Nella documentazione viene sottolineato di fare particolare attenzione ai seguenti aspetti:

- permessi delle applicazioni
- uso dei servizi di localizzazione
- gestione sicura dei dati
- uso degli identificatori

Permessi delle applicazioni

L'applicazione deve acquisire la fiducia dell'utente essendo trasparente durante la propria esecuzione e fornendogli il pieno controllo sull'esperienza d'uso.

In primis l'applicazione deve gestire in maniera adeguata i permessi richiesti: va richiesto il minor numero di permessi possibile, limitandosi a quelli strettamente necessari al funzionamento dell'app. Infatti le richieste dei permessi sono uno strumento per proteggere l'utente e le relative informazioni sensibili conservate nel dispositivo, ma non solo, proteggono anche il sistema perché manipolare una risorsa sensibile potrebbe avere impatto sull'intero sistema. Tali richieste non vanno fatte tutte contemporaneamente durante la fase di startup, ma è buona norma diluirle, progettando un'interfaccia utente in cui specifiche azioni richiedano specifici permessi, in modo tale che le richieste siano rimandate al momento in cui sono necessarie e che l'utente capisca a cosa serve il particolare permesso. Questo è stato reso possibile da Android 6.0 (API level 23), in cui le applicazioni possono richiedere i permessi all'utente a runtime, mentre prima erano chiesti dal sistema in fase di installazione, senza alcun modo di rifiutarli o revocarli. Addirittura in Android 11.0 (API level 30) sono stati introdotti i permessi one-time e la revoca dei permessi per le app non usate da tempo.

Nel caso di versioni successive dell'applicazione, vanno rivisti i permessi richiesti in precedenza per capire se sono ancora necessari, eventualmente rimuovendoli nel caso diventino superflui. Si deve tenere in considerazione che le nuove versioni di Android spesso introducono nuove tecniche per accedere ai dati, le quali garantiscono il rispetto della privacy, senza richiedere permessi all'utente.

Un altro passo importante da compiere durante lo sviluppo dell'app è la spiegazione del motivo per cui una certa funzionalità richiede un particolare permesso, specialmente quando c'è un mismatch tra ciò che si sta richiedendo e lo scopo dell'applicazione. Numerosi studi mostrano che gli utenti sono più predisposti a dare l'autorizzazione quando sono consapevoli della ragione per cui l'app necessita di un permesso.

Inoltre va gestito anche il caso in cui l'utente neghi o revochi un permesso all'applicazione: in tal caso l'app può disabilitare tutte le funzioni che necessitano di quel particolare

permesso. L'utente o il sistema possono negare molte volte tale permesso, per questo Android ignora le successive richieste del permesso della stessa app.

Infine, come l'utente sceglie le app che richiedono meno permessi per la stessa funzionalità, anche gli sviluppatori devono usare librerie o SDK di terze parti che limitano i permessi superflui. Infatti tali permessi sono pericolosi per la fiducia data all'applicazione perchè, seppur non siano richiesti dall'app ma dalla libreria, l'utente crede provengano da essa.

Uso dei servizi di localizzazione

Nel caso in cui l'app raccolga informazioni riguardo la posizione, è ragionevole, quando si chiede il permesso, prima spiegare all'utente come l'app utilizza queste informazioni e per quali funzionalità sono necessarie. In questo modo l'utente capisce le ragioni ed è in grado di fare una decisione ragionata sull'autorizzazione da concedere.

Ovviamente, visto che i dati sulla localizzazione sono strettamente personali, vanno richiesti esclusivamente dalle applicazioni fornitrici di servizi in cui conoscere la posizione è indispensabile, chi può farne a meno non deve richiederli.

Si consiglia di accedere ai dati relativi alla posizione mentre l'app è in foreground, cioè visibile all'utente e con cui c'è interazione, cosicchè egli capisca più chiaramente il motivo per cui l'app richiede tali informazioni, limitando il più possibile le richieste, eventualmente modificando la logica di accesso alla posizione, quando l'app è in background e quindi non visibile. Se l'applicazione richiede la localizzazione in background, è fondamentale assicurarsi che sia una funzionalità chiave, che offra dei benefici evidenti all'utente e che sia fatta in modo trasparente per lui. Questo vale, ad esempio, nel caso di implementazione del geofencing, cioè l'uso di geo-fence (lett. geo-recinzione), ovvero un perimetro virtuale associato a una zona del globo terrestre, il quale permette di comunicare, sfruttando tecniche di geomarketing, messaggi pubblicitari particolari a persone che passano in un determinato luogo, oppure consente più in generale forme di sorveglianza e sicurezza. Nei casi in cui l'applicazione acceda alla localizzazione in background, essa può ricevere solo qualche aggiornamento della posizione durante ogni ora: questo è dovuto ai limiti sulla localizzazione in background introdotti da Android 8.0 (API level 26) per preservare la durata della batteria.

Per accedere ai servizi di localizzazione occorre richiedere un permesso, in cui deve essere specificata la categoria e l'accuratezza.

La categoria distingue se la localizzazione viene fatta in foreground o in background: il sistema considera l'app come utilizzatrice della prima modalità se c'è un'activity visibile o un foreground service in attività, appartenenti all'app che chiede la posizione corrente del dispositivo, in tutti gli altri casi l'app viene considerata come utilizzatrice della localizzazione in background.

Da Android 10 (API level 29) per richiedere l'accesso alla localizzazione in background deve essere dichiarato il permesso `ACCESS_BACKGROUND_LOCATION` nel manifest dell'app. Nelle prime versioni di Android si sbloccava tale accesso in automatico, quando l'app riceveva l'accesso alla localizzazione in foreground.

L'accuratezza della localizzazione contempla due livelli: approssimativa o precisa. La prima fornisce una stima della posizione del dispositivo, in un'area di circa 3 chilometri quadrati; deve essere dichiarato nel manifest dell'app il permesso `ACCESS_COARSE_LOCATION` ma non il permesso `ACCESS_FINE_LOCATION`. La localizzazione con un'accuratezza precisa fornisce una stima della posizione del dispositivo nel modo più accurato possibile, con uno scarto entro soli 50 metri, molto spesso questo scarto si riduce a qualche metro; deve essere dichiarato nel manifest dell'app il permesso `ACCESS_FINE_LOCATION`. Se l'utente concede il solo permesso della localizzazione approssimativa, l'applicazione ha accesso solo a questo tipo di localizzazione, indipendentemente da quali permessi sono stati dichiarati nel manifest. Da Android 12 (API level 31) l'utente può richiedere all'applicazione di funzionare con le sole informazioni della localizzazione approssimativa, nonostante l'app abbia specificato il permesso runtime per la localizzazione precisa. L'applicazione dovrebbe continuare a funzionare quando l'utente concede solo l'accesso alla posizione approssimativa, se c'è una determinata funzionalità che richiede necessariamente la posizione precisa, si chiede all'utente di accettare l'ulteriore permesso relativo.

A volte è necessario che l'applicazione mantenga l'accesso alla localizzazione per un task iniziato dall'utente e che deve continuare dopo che è stata lasciata la relativa interfaccia utente: per fare ciò si può iniziare un foreground service prima che l'app finisca in background, nel metodo `onPause()`. Non è buona pratica lanciare un foreground service dal background, piuttosto conviene lanciare una notifica dall'app e poi eseguire il codice per accedere alla posizione quando l'interfaccia utente dell'app diventa visibile.

Android fornisce degli strumenti per evitare di accedere al permesso della localizzazione inutilmente e quindi aumentare la protezione della privacy dell'utente. Ad esempio da Android 8.0 (API level 26) è stato introdotto il Companion Device Manager: una classe che esegue una scansione via Bluetooth o Wi-Fi dei dispositivi vicini, senza sfruttare il permesso `ACCESS_FINE_LOCATION` e impedendo un rilevamento continuo dei dispositivi.

Gestione sicura dei dati

L'applicazione deve gestire in modo trasparente e sicuro i dati sensibili di cui è in possesso, in primis rendendo consapevole l'utente relativamente a quali sono i dati riservati raccolti, usati e condivisi.

Dal punto di vista tecnologico questa tutela viene realizzata prestando attenzione a diversi aspetti nella manipolazione da parte delle applicazioni delle informazioni.

Uno dei punti principali riguarda il salvataggio dei dati nella memoria. I dati sensibili dell'utente dovrebbero essere memorizzati in una zona ad-hoc della memoria: l'app-specific internal storage. Questa area corrisponde ad alcune directory, con una capacità ridotta, che si trovano nell'internal storage e in cui l'app può conservare i propri file sensibili, a cui le altre applicazioni non possono accedere. Non serve alcun permesso di lettura e scrittura, l'accesso è garantito in automatico.

Se l'internal storage non dovesse fornire abbastanza spazio, può essere preso in considerazione l'uso dell'app-specific external storage. Il sistema fornisce anche in questa zona di memoria, su un dispositivo esterno, delle directory dove l'app può mantenere dei file, da cui attingere per servire le richieste dell'utente alla relativa app.

Da Android 4.4 (API level 19) non serve alcun permesso per accedere a tale zona dello storage. Tuttavia non c'è garanzia che il dispositivo di archiviazione esterna sia accessibile, visto che potrebbe essere stata rimossa, perciò, se i file archiviati nell'app-specific external storage sono indispensabili per l'avvio dell'applicazione, vanno spostati nella zona riservata all'app nell'internal storage. Fino ai dispositivi con la versione Android 9 (API level 28) o inferiore, un'applicazione poteva accedere agli app-specific file sull'external storage appartenenti alle altre applicazioni, specificando solamente un appropriato permesso per lo storage: `READ_EXTERNAL_STORAGE` per la sola lettura, `WRITE_EXTERNAL_STORAGE` anche per la scrittura. Per proteggere di più i file riservati degli utenti e per maggiore sicurezza, da Android 10 (API level 29) di default le applicazioni hanno un accesso limitato all'external storage, vedono solo il proprio scoped storage, non possono più accedere alle aree app-specific appartenenti alle altre app. Infine, da Android 11 (API level 30) il permesso `WRITE_EXTERNAL_STORAGE` non ha più alcun effetto sull'accesso allo storage da parte dell'app, in questo modo la privacy dell'utente aumenta poiché le app possono accedere solamente alle aree del file system del dispositivo che effettivamente usano. Viene anche introdotto il permesso `MANAGE_EXTERNAL_STORAGE` per fornire l'accesso ai file nella memoria esterna che non si trovano nell'app-specific directory o nel MediaStore, che si occupa di gestire i media (immagini, video, audio, etc).

Oltre al salvataggio accurato, si deve fare attenzione anche alla comunicazione di dati sensibili tra le diverse applicazioni. Per rendere più sicuro questo processo conviene usare permessi di sola lettura o sola scrittura e fornire ai client un accesso singolo (one-shot) ai dati usando il flag `FLAG_GRANT_READ_URI_PERMISSION` in lettura e il flag `FLAG_GRANT_WRITE_URI_PERMISSION` in scrittura negli Intent di richiesta.

Per una maggiore trasparenza verso l'utente, è buona pratica che l'applicazione in foreground segnali in tempo reale, attraverso qualche simbolo, quando sta utilizzando i sensori di sistema, ad esempio il microfono, la fotocamera, la localizzazione, etc. Da Android 9 (API level 28), per aumentare la privacy dell'utente, sono stati introdotti numerosi cambiamenti, tra cui la limitazione dell'accesso da parte delle app in background ai sensori del dispositivo. Infatti le applicazioni attive in background devono rispettare numerose restrizioni: non è possibile accedere al microfono o alla fotocamera, i sensori che si sono registrati per ricevere periodicamente aggiornamenti, come l'accelerometro e il giroscopio, non ricevono gli eventi e nemmeno i sensori che si aggiornano quando c'è un cambiamento o con modalità one-shot. Tuttavia, prima di Android 11 (API level 30), era possibile accedere a sensori avviando un foreground service dall'app in background, ma poi vengono introdotte limitazioni più forti in cui un foreground service non può accedere alla fotocamera e al microfono, e neanche alla localizzazione se non ha ricevuto l'autorizzazione al permesso `ACCESS_BACKGROUND_LOCATION`. Da Android 12 (API level 31) nessuna applicazione in background può lanciare un foreground service, tranne in alcuni casi particolari.

Infine Android mette a disposizione numerosi strumenti per mantenere i dati riservati. Di fondamentale importanza è la crittografia dei dati, sia quella basata su file che quella dell'intero disco (rimossa da Android 10 perché limitava le funzionalità al riavvio), essa garantisce che, anche se un componente non autorizzato in qualche modo provasse ad accedere ai dati, non sarebbe in grado di capirli. Vengono anche forniti sistemi di sicurezza supportati dall'hardware, tra cui spicca il Keystore, fondamentale per la generazione e

l'archiviazione sicura delle coppie di chiavi di firma asimmetrica. Anche Android Jetpack fornisce delle librerie di supporto per la sicurezza, tra cui Jetpack Security e Jetpack Preferences. Tali argomenti verranno approfonditi in dettaglio nella sezione della Security, nella sottosezione riguardo la crittografia.

Uso degli identificatori

Per proteggere la privacy degli utenti, un'applicazione deve usare l'identificatore più restrittivo possibile per soddisfare le proprie esigenze.

Un dispositivo offre di fabbrica alcuni identificatori legati all'hardware, persistenti, non modificabili: quali l'International Mobile Equipment Identity (IMEI), il numero di serie, etc. In genere in molti casi le applicazioni dovrebbero evitare di utilizzare questi dati, fornendo ugualmente le diverse funzionalità attraverso altri tipi di identificatori. Oltretutto da Android 10 (API level 29) sono state aggiunte delle restrizioni per gli identificatori non modificabili, ovvero per accedervi l'app deve essere di sistema o proprietaria di un profilo nel dispositivo, avere dei permessi operativi speciali o essere in una whitelist specifica e possedere il permesso privilegiato `READ_PRIVILEGED_PHONE_STATE`; altrimenti viene sollevata una `SecurityException`.

Nei casi di profilazione o di pubblicità l'applicazione deve usare esclusivamente un Advertising ID: si tratta di un identificatore fornito da Google Play Services, attraverso la classe `AdvertisingIdClient`, univoco e reimpostabile dall'utente per la pubblicità. È un requisito necessario per le app in Google Play.

Garantisce agli utenti un controllo migliore, per esempio consentendo la disattivazione degli annunci personalizzati, e agli sviluppatori un sistema semplice per guadagnare dalle loro app. Vanno sempre rispettate le intenzioni dell'utente sia nel resettare l'advertising ID, senza collegare a nuovi identificatori precedenti identificatori o dati connessi, a meno che non ci sia un consenso esplicito, sia nel garantire i flag specificati dall'utente nella configurazione del limite alla quantità di tracking effettuabile nei confronti dell'ID. Inoltre, come richiesto dal Google Play Developer Content Policy, non deve essere possibile ricondursi, tramite l'Advertising ID, a informazioni identificative della persona o associate a identificatori persistenti del dispositivo.

Nella maggioranza dei casi d'uso non pubblicitari, la soluzione raccomandata per identificare l'istanza di un'applicazione in esecuzione su un dispositivo è il Firebase installation ID (FID). Questo identificatore è ottenibile dalla classe `FirebaseInstallations` attraverso il metodo `getToken()`, risulta accessibile solo dall'istanza dell'app per cui è stato creato e rimane valido finché l'app risulta installata, poi viene resettato. Esso garantisce una maggiore privacy rispetto agli ID persistenti, non resettabili, legati all'hardware del dispositivo.

Nel caso in cui un identificatore FID non sia pratico, è consigliato usare un globally-unique ID (GUID) per identificare univocamente l'istanza di un'app. Si tratta di un identificatore numerico pseudo-casuale unico a livello globale, che conviene sia conservato nel app-specific internal storage, anziché nell'external storage condiviso, per evitare problemi relativi alla condivisione dell'identificatore tra le altre applicazioni. Per ottenerlo basta

invocare il metodo `randomUUID()` della classe `UUID` (immutable universally unique identifier).

Infine anche l'uso degli indirizzi MAC dovrebbe essere evitato, dato che sono unici globalmente, non resettabili e sopravvivono ai reset di fabbrica. Infatti da Android 6.0 (API level 23), per proteggere la privacy dell'utente, l'accesso a tali indirizzi MAC è limitato alle sole app di sistema, le app di terze parti non vi possono accedere.

ANDROID: SICUREZZA DEI DATI APPLICATIVI

La piattaforma Android offre numerose funzionalità per ridurre la frequenza e l'impatto delle falle di sicurezza nelle applicazioni. Il sistema è progettato per poter realizzare applicazioni robuste dal punto di vista della sicurezza, sfruttando autorizzazioni predefinite per il sistema e i file ed evitando così decisioni difficili in materia di sicurezza.

Le seguenti funzionalità in Android contribuiscono a costruire applicazioni sicure:

- Android Application Sandbox, che isola i dati e l'esecuzione di ciascuna app dalle altre
- Lo storage organizzato in aree riservate o condivise per le diverse applicazioni, garantisce un accesso controllato e sicuro ai dati
- Permessi per limitare l'accesso alle funzionalità del sistema e ai dati dell'utente
- La comunicazione sicura interprocesso IPC
- Tecnologie in ambiente nativo per ridurre i rischi più comuni associati a un'errata gestione della memoria
- La crittografia simmetrica e asimmetrica per i dati sensibili nel dispositivo: KeyStore, hashing e autenticazione di messaggio

Android Application Sandbox

In Android ogni applicazione viene eseguita all'interno di un processo separato, che possiede la propria istanza della DalvikVM. Android è un sistema Linux multi-user, in cui ogni applicazione corrisponde a un utente diverso, il sistema di default assegna ad ogni applicazione un univoco user ID (UID) Linux (l'ID viene utilizzato solo dal sistema ed è sconosciuto all'applicazione). Il sistema imposta i permessi per tutti i file in un'applicazione in modo tale che solo lo user ID assegnato a tale applicazione vi possa accedere. Android ha come cuore il processo "Zygote", che viene lanciato per primo. Quando si avvia un'applicazione, Zygote viene forkato, e l'heap di Dalvik viene precaricato con le classi e i dati da Zygote. Dalvik, come le macchine virtuali di altri linguaggi, si occupa in automatico del garbage collection nell'heap.

Il meccanismo Sandbox è basato sulla separazione UNIX-style tra i diversi utenti dei permessi sui file e sui processi.

Android usa l'UID per creare un Application Sandbox a livello del kernel, ovvero un ambiente in cui i processi del kernel vengono eseguiti in maniera isolata. Il kernel rafforza la sicurezza tra le app e il sistema in fase di esecuzione attraverso gli strumenti di Linux, cioè gli user e group ID assegnati alle applicazioni. Di default, le applicazioni non possono interagire direttamente con le altre applicazioni e hanno un accesso limitato al sistema operativo, in questo modo non possono compiere azioni (anche malevoli) perchè non hanno gli appropriati privilegi utente.

Poichè l'Application Sandbox caratterizza il kernel, la sicurezza di tale meccanismo si estende anche al codice nativo e alle applicazioni di sistema, ovvero tutto il software che si poggia sul kernel viene eseguito all'interno dell'Application Sandbox: librerie di sistema, framework applicativi, runtime applicativi e tutte le applicazioni.

Per evadere dall'Application Sandbox in un dispositivo correttamente configurato, deve essere compromessa la sicurezza del kernel di Linux. Tuttavia, come le altre funzionalità di sicurezza, le singole protezioni che rafforzano questo meccanismo di sandbox non sono invulnerabili.

Android ha introdotto nel tempo diverse protezioni per dare robustezza all'Application Sandbox, rafforzando così l'originale Discretionary access control (DAC) basato sull'UID.

Brevemente il Discretionary access control (DAC) è un tipo di controllo dell'accesso basato sull'identità del soggetto o del gruppo a cui appartiene, si contrappone al Mandatory access control (MAC) che è un tipo di controllo dell'accesso da parte del sistema operativo basato sulla sensibilità delle informazioni che la risorsa contiene e l'autorizzazione del soggetto per accedere alle informazioni di tale livello di sensibilità. Con il DAC l'accesso è controllato basandosi solamente sugli user e group ID, mentre MAC con l'abilità di limitare i privilegi associati a un processo in esecuzione, limita lo scope di potenziali danni risultanti dalle vulnerabilità nelle applicazioni e nei servizi di sistema, fornendo un isolamento maggiore alle applicazioni che permettono l'esecuzione sicura di applicazioni inaffidabili.

Le versioni di Android hanno introdotto le seguenti protezioni:

- In Android 5, il modulo di sicurezza del kernel Linux, Security-Enhanced Linux (SELinux), introduce il meccanismo MAC nella separazione tra il sistema e le applicazioni. Tuttavia, le applicazioni di terze parti sono eseguite all'interno dello stesso contesto SELinux, di fatto l'isolamento tra applicazioni continua a essere regolato principalmente dal meccanismo DAC.
- In Android 6, la sandbox SELinux viene estesa per isolare le applicazioni nel confine con ogni utente fisico. Inoltre Android modifica i valori di default dei dati applicativi con nuovi valori più sicuri: ad esempio per le app con `targetSdkVersion >= 24` il valore di default per i permessi DAC per accedere alla directory della home dell'app passa da 751 a 700.
- In Android 8, tutte le applicazioni sono eseguite con un filtro, `seccomp-bpf`, che limita le system call che possono essere usate, così si rafforza il confine tra il kernel e le app
- In Android 9, le app senza privilegi con `targetSdkVersion >= 28` devono essere eseguite in sandbox SELinux individuali, fornendo il meccanismo MAC per ogni app di base. In questo modo si aumenta l'isolamento delle app, prevenendo la sovrascrittura dei valori di default sicuri, ma soprattutto evitando che i dati applicativi diventino accessibili dal mondo esterno.
- In Android 10, le app possiedono una vista limitata del filesystem, senza l'accesso diretto ai percorsi esterni come `/sdcard/DCIM`. Tuttavia le app continuano ad avere accesso completo ai percorsi relativi al proprio package, ovvero package-specific.

Salvataggio dei dati

Uno degli aspetti più importanti per la sicurezza in Android è la gestione dei dati, perché prima che l'utente conceda i permessi di accesso a una determinata applicazione, egli ha tutto l'interesse che i suoi dati personali o sensibili siano salvaguardati il più possibile.

Tutti i dati delle applicazioni possono essere salvati sul dispositivo principalmente in tre modi, che sfruttano:

- Internal storage: spazio di archiviazione nella memoria interna
- External storage: spazio di archiviazione nella memoria esterna
- Content provider: componente che si occupa di custodire i dati strutturati da condividere ad altri componenti o applicazioni al di fuori del relativo gestore

L'**internal storage** è uno spazio di archiviazione interno al dispositivo, disponibile in tutti gli smartphone, limitato e non espandibile. Esso rappresenta la zona più affidabile in cui conservare i dati da cui dipende l'applicazione: infatti ogni app possiede uno spazio, detto app-specific storage, a cui accede senza richiedere alcun permesso, riservato e isolato grazie al meccanismo di sandbox di Android, al quale nessun altro componente vi può accedere. Inoltre, come misura di sicurezza ulteriore, quando l'utente disinstalla un'app, il sistema elimina tutti i file che l'app ha salvato in tale zona, perciò non si deve salvare nulla che l'utente considera persistente indipendentemente dall'app.

L'app-specific internal storage può essere acceduto attraverso i metodi `getFilesDir()` e `getCacheDir()`, che restituiscono rispettivamente le directory riservate ai file e ai dati della cache delle app.

Tuttavia va ricordato che questo spazio ha una dimensione esigua, quindi va utilizzato solo quando i dati applicativi sono sensibili. Se un file non contiene dati privati o informazioni riservate, ma serve solamente a una sola applicazione e a patto che non sia vitale per la sua esecuzione, può essere spostato nello external storage dedicato.

Se si deve salvare dati particolarmente sensibili o privati, la Security Library, che fa parte di Android Jetpack, permette la creazione di `EncryptedFile`, e le relative operazioni di lettura e scrittura attraverso `FileInputStream` e `FileOutputStream`. In fase di creazione vanno fornite le specifiche dell'algoritmo di crittografia e la chiave da usare.

```
String masterKeyAlias = MasterKeys.getOrCreate(MasterKeys.AES256_GCM_SPEC);

File file = new File(context.getFilesDir(), "secret_data");
EncryptedFile encryptedFile = EncryptedFile.Builder(
    file,
    context,
    masterKeyAlias,
    EncryptedFile.FileEncryptionScheme.AES256_GCM_HKDF_4KB
).build();

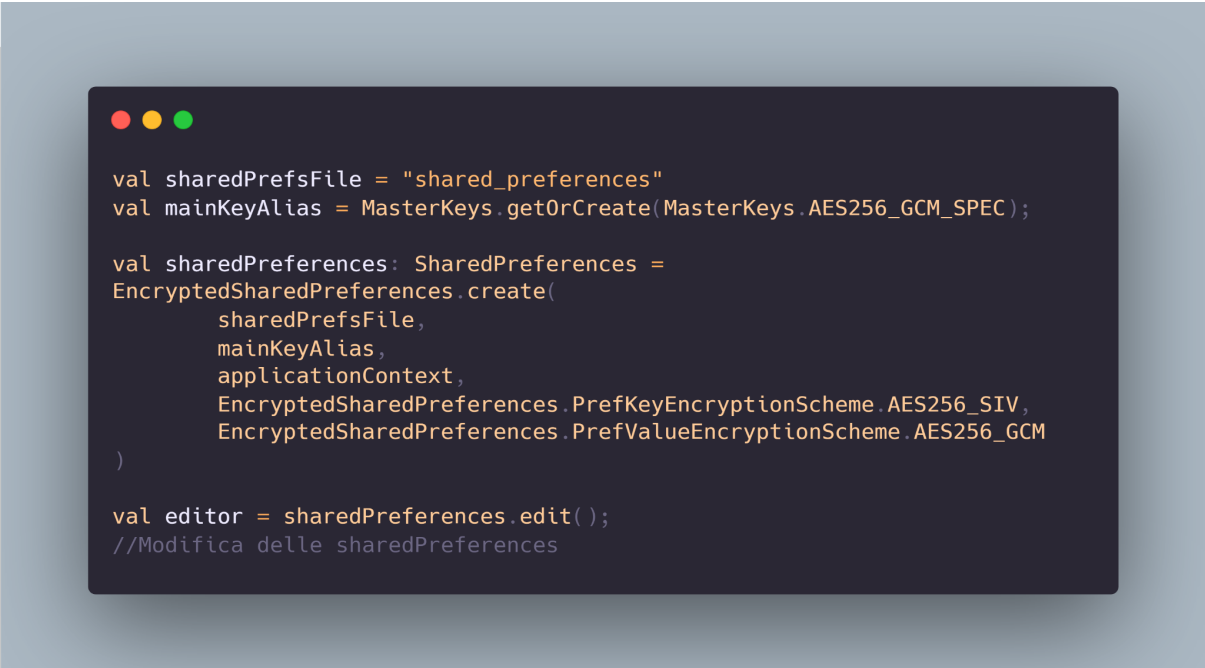
// write to the encrypted file
FileOutputStream encryptedOutputStream = encryptedFile.openFileOutput();

// read the encrypted file
FileInputStream encryptedInputStream = encryptedFile.openFileInput();
```

Quando si devono salvare dati strutturati, che non devono essere accessibili a nessuna altra applicazione e quindi vivono nell'internal storage dell'app, si possono utilizzare le SharedPreferences per dati sotto forma di coppie chiave-valore, oppure i database relazionali per i dati più complessi con più di due colonne, gestibili attraverso le librerie SQLiteDatabase o Room.

In particolare le SharedPreferences permettono di salvare piccole collezioni composte da chiave-valore, un oggetto della classe SharedPreferences punta a un file contenente tali coppie e fornisce i metodi per la lettura e la scrittura. Questo meccanismo non è adatto per condividere dati tra diverse applicazioni, ci sono altri modi che verranno analizzati in seguito. Se si deve usare solo un file con le SharedPreferences all'interno dell'Activity, c'è il metodo getPreferences(), se si devono usare file multipli per le SharedPreferences, condivisi nell'applicazione, c'è il metodo getSharedPreferences() in cui deve essere specificato il nome identificativo. In questo ultimo caso, se si vuole che solo la propria applicazione possa accedere alle informazioni contenute all'interno del file SharedPreferences da lei stessa creato, va specificato nel metodo la modalità MODE_PRIVATE.

Se si vuole usare le SharedPreferences con una sicurezza più elevata, Android Jetpack mette a disposizione la Security Library, contenente la classe EncryptedSharedPreferences. Questa incapsula la classe SharedPreferences e automaticamente crittografa le chiavi e i valori usando due diversi algoritmi: le chiavi sono crittografate con un algoritmo deterministico in modo tale che possano essere ricercate, i valori invece crittografati con l'algoritmo AES-256 GCM. Così facendo le SharedPreferences possono essere modificate in modo più sicuro.

A screenshot of a code editor with a dark background and light-colored text. The code is in Kotlin and demonstrates how to create and edit EncryptedSharedPreferences. It includes comments in Italian. The code is as follows:

```
val sharedPrefsFile = "shared_preferences"
val mainKeyAlias = MasterKeys.getOrCreate(MasterKeys.AES256_GCM_SPEC);

val sharedPreferences: SharedPreferences =
    EncryptedSharedPreferences.create(
        sharedPrefsFile,
        mainKeyAlias,
        applicationContext,
        EncryptedSharedPreferences.PrefKeyEncryptionScheme.AES256_SIV,
        EncryptedSharedPreferences.PrefValueEncryptionScheme.AES256_GCM
    )

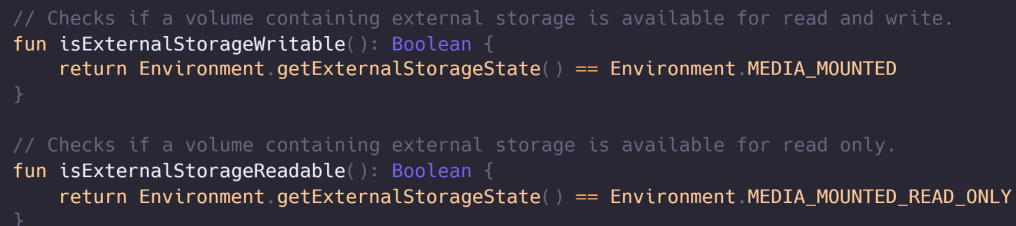
val editor = sharedPreferences.edit();
//Modifica delle sharedPreferences
```

Infine c'è un'area nell'internal storage adibita per il salvataggio dei dati condivisi tra le applicazioni, che devono rimanere anche quando l'utente disinstalla l'app. Possono essere

conservati contenuti multimediali quali foto, video, audio, etc, attraverso la classe `MediaStore`, oppure documenti PDF ed ebook grazie allo `Storage Access Framework`.

L'**external storage** è uno spazio di archiviazione aggiuntivo ospitato su supporti di memorizzazione rimovibili, tra i più utilizzati ci sono le SD card. Questi strumenti permettono di sgravare il peso sull'internal storage, ma non sono supportati da tutti gli smartphone. Le SD card e gli altri volumi rimovibili appaiono nel file system come parte dell'external storage, sono rappresentati da path come `/sdcard`.

Tuttavia questi supporti, come suggerito dall'aggettivo rimovibili, potrebbero non essere sempre disponibili, visto che l'utente può rimuoverli a suo piacimento, sfortunatamente proprio nel momento in cui l'app sta cercando di accedervi o li sta usando. Quindi deve essere inclusa la logica per verificare la disponibilità di tali memorie.



```
// Checks if a volume containing external storage is available for read and write.
fun isExternalStorageWritable(): Boolean {
    return Environment.getExternalStorageState() == Environment.MEDIA_MOUNTED
}

// Checks if a volume containing external storage is available for read only.
fun isExternalStorageReadable(): Boolean {
    return Environment.getExternalStorageState() == Environment.MEDIA_MOUNTED_READ_ONLY
}
```


L'external storage può essere usato per immagazzinare dati abbastanza voluminosi, non sensibili, specifici per una singola applicazione, oppure anche dati condivisibili con altri componenti. Sempre considerando che questi dati potrebbero essere non disponibili per via della rimozione del supporto esterno, quindi vanno evitati dati indispensabili, vitali per l'esecuzione dell'applicazione.

L'external storage contiene delle directory per il salvataggio di file e di dati della cache, accessibili tramite i metodi `getExternalFilesDir()` o `getExternalCacheDir()`, tali spazi conservano informazioni riservate a una sola applicazione e vengono resettati quando essa viene rimossa. Tuttavia, qualsiasi applicazione con gli adeguati permessi per l'external storage vi poteva accedere, allora per maggiore sicurezza, da Android 10 viene introdotto il concetto di "scoped storage", in cui di default le applicazioni hanno accesso solamente alle proprie directory sull'external storage, dette app-specific directory, e ai media da loro creati. Questo app-specific external storage affianca l'app-specific internal storage per mantenere i dati applicativi non condivisi.

Si possono salvare nell'external storage, come nella memoria interna, anche file persistenti che devono essere condivisi con altre applicazioni: i media (le immagini, gli audio, i video) sono gestiti attraverso il `MediaStore`; altri tipi di contenuti da condividere, inclusi i documenti, i file scaricati o i file archiviati su cloud, sono gestiti dallo `Storage Access Framework`.

Se l'applicazione usa dati nell'external storage, si deve verificare che il contenuto dei dati non sia stato corrotto o modificato. Va usata una logica per assicurarsi che le informazioni coincidono con quelle attese, e gestire di conseguenza il caso negativo.

Nell'esempio sottostante si utilizza un procedimento basato sul controllo dell'hash.



```
val hash = calculateHash(stream)
// "expectedHash" si trova in un posto sicuro
if (hash == expectedHash) {
    // Elaborazione
}
else{
    // Dati corrotti
}

fun calculateHash(stream: InputStream): String {
    val digest = MessageDigest.getInstance("SHA-512")
    val digestStream = DigestInputStream(stream, digest)
    while (digestStream.read() != -1) {
        // DigestInputStream legge i dati
    }
    return digest.digest().joinToString(":") { "%02x".format(it) }
}
```

Si consiglia di fornire un accesso rapido ai dati applicativi non sensibili, conservandoli nella cache del dispositivo. Se un'applicazione deve mantenere dati di questo tipo, con dimensione superiore a 1 MB, vanno salvati nella cache dell'app-specific external storage, altrimenti si può sfruttare la corrispondente cache nella memoria interna.

Il **Content provider** è un componente applicativo che offre un meccanismo di salvataggio dei dati con lo scopo di condividerli con altri componenti, interni alla propria applicazione o, più comunemente, con altre applicazioni. La sua visibilità è regolata attraverso l'attributo, nel manifest dell'app, `android:exported`: se assume valore `false` può essere usato solo dai componenti interni all'applicazione che l'ha creato e le altre applicazioni non possono sfruttare il content provider, altrimenti se assume valore `true` le altre app possono accedere ai dati da esso conservati.

Se si sta usando un content provider per condividere dati solamente tra alcune proprie applicazioni, conviene utilizzare un meccanismo di gestione dei permessi basato sulla firma delle app (signature), specificando nel manifest il seguente attributo per il permesso `android:protectionLevel=signature`. In questo modo non si richiede la conferma dell'utente, si fornisce così un'esperienza d'uso migliore e un accesso più controllato al content provider quando le app accedono ai dati usando la stessa chiave per la firma.

La chiave per la firma dell'app è usata per firmare gli APK installati su un dispositivo dell'utente, viene fornita dallo sviluppatore o generata da Google, che poi la usa. Fa parte dei meccanismi di sicurezza di Android, tale chiave non cambia mai durante il ciclo di vita dell'applicazione, è privata e deve essere mantenuta segreta.

Quando si crea un `ContentProvider` visibile a tutte le app, si può specificare solo un singolo permesso per la lettura e la scrittura, oppure più permessi distinti. Tali permessi dovrebbero limitarsi solo a quelli richiesti per l'esecuzione di task esistenti: è più facile aggiungere permessi in momenti successivi per servire nuove funzionalità piuttosto che rimuoverli e impattare l'esperienza dell'utente.

Il `ContentProvider` può fornire un accesso più controllato e granulare ai dati dichiarando nel manifest l'attributo `android:grantUriPermissions`, concedendo i permessi attraverso i flag `FLAG_GRANT_READ_URI_PERMISSION` e `FLAG_GRANT_WRITE_URI_PERMISSION` negli Intent che attivano il componente e che richiedono uno specifico URI. Se tale attributo assume valore `true`, il permesso può essere concesso a qualsiasi dato del content provider, altrimenti con valore `false` (di default) i permessi sono concessi solo a dei sottoinsiemi di dati elencati in `<grant-uri-permission>` sempre nel manifest.

L'interfaccia di accesso a un content provider sfrutta i metodi parametrizzati `query()`, `update()`, `insert()` e `delete()` per evitare possibili tecniche di SQL injection, ovvero attacchi ai database relazionali SQL da parte di terzi malevoli.

Va prestata attenzione anche alla gestione del permesso di scrittura poichè, nei content provider che hanno una struttura prevedibile, tale permesso può trasformarsi in un permesso di lettura: infatti il permesso di scrittura permette di eseguire comandi SQL con clausole `WHERE` creative, il cui risultato una volta analizzato può rivelare la presenza di altri dati e quindi concederne la lettura, ad esempio alcune operazioni di modifica di righe nelle tabelle.

Gestione dei permessi

Come detto in precedenza, tutte le applicazioni Android vivono all'interno del proprio processo isolato su un'istanza separata della DalvikVM, grazie al meccanismo di sandbox. Spesso le applicazioni forniscono funzionalità esterne al proprio ambiente in cui vengono eseguite, quindi hanno bisogno di accedere a dati, funzionalità o dispositivi hardware, come i sensori, che non appartengono a loro. Per fare in modo che questo avvenga, e per garantire allo stesso tempo un'elevata sicurezza per l'utente e il sistema, l'applicazione deve richiedere dei permessi nel `AndroidManifest.xml`.

I permessi non sono tutti dello stesso tipo, attualmente in Android vengono raggruppati in quattro categorie:

- Permessi normal (normali)
- Permessi dangerous (pericolosi)
- Permessi signature (firmati)
- Permessi special

I **permessi normal** permettono di accedere a dati e azioni che presentano un rischio estremamente limitato per la privacy dell'utente e per le operazioni delle altre app.

Ad esempio vi appartengono i seguenti permessi:

- `android.permission.ACCESS_NETWORK_STATE`: accesso allo stato delle rete
- `android.permission.ACCESS_WIFI_STATE`: accesso alle informazioni sulle reti Wi-Fi
- `android.permission.BLUETOOTH`: accesso alla connettività Bluetooth
- `android.permission.INTERNET`: accesso all'apertura di socket di rete
- `android.permission.NFC`: accesso alla connettività Nfc
- `android.permission.QUERY_ALL_PACKAGES`: accesso all'interrogazione delle dichiarazioni nell'AndroidManifest.xml di qualsiasi app normale sul dispositivo
- `android.permission.SET_ALARM`: accesso all'invio in broadcast di Intent per impostare un allarme per l'utente
- `android.permission.SET_WALLPAPER`: accesso per impostare uno sfondo
- `android.permission.VIBRATE`: accesso alla gestione della vibrazione
- `android.permission.WAKE_LOCK`: accesso all'uso del PowerManager WakeLocks per mantenere il processore o lo schermo attivi

I **permessi dangerous** permettono di accedere a dati più sensibili per la riservatezza dell'utente e ad azioni che possono inficiare maggiormente la stabilità del sistema e di altre applicazioni. La posizione e i contatti sono esempi di dati riservati, il microfono e la fotocamera sono sensori che forniscono l'accesso a informazioni sensibili dell'utente.

Ad esempio vi appartengono i seguenti permessi:

- `android.permission.ACCESS_FINE_LOCATION`: accesso alla posizione precisa
- `android.permission.ACCESS_COARSE_LOCATION`: accesso alla posizione approssimativa
- `android.permission.BLUETOOTH_SCAN`: accesso alla ricerca e connessione con dispositivi Bluetooth vicini
- `android.permission.CALL_PHONE`: accesso per iniziare una chiamata senza passare attraverso l'interfaccia per la conferma dell'utente
- `android.permission.CAMERA`: accesso alla/e fotocamera/e
- `android.permission.READ/WRITE_CALENDAR`: accesso in lettura/scrittura agli eventi nel calendario dell'utente
- `android.permission.READ/WRITE_CONTACTS`: accesso in lettura/scrittura ai contatti nella rubrica dell'utente
- `android.permission.READ_EXTERNAL_STORAGE`: accesso alla lettura nell'external storage
- `android.permission.RECORD_AUDIO`: accesso al microfono per la registrazione di audio
- `android.permission.WRITE_EXTERNAL_STORAGE`: accesso alla scrittura nell'external storage, automaticamente sblocca anche l'accesso alla lettura

I **permessi signature** sono particolari permessi richiesti da un'applicazione e definiti in un'altra applicazione, sono accettati dal sistema a tempo di installazione, a patto che le due app siano firmate con lo stesso certificato. In caso contrario non vengono concessi.

Ad esempio vi appartengono i seguenti permessi:

- `android.permission.BATTERY_STATS`: accesso ai dati della batteria
- `android.permission.BIND_AUTOFILL_SERVICE`: deve essere richiesto da un `AutofillService`, per garantire che solo il sistema possa assegnarglielo
- `android.permission.BIND_INPUT_METHOD`: deve essere richiesto da un `InputMethodService`, per garantire che solo il sistema possa assegnarglielo
- `android.permission.BIND_PRINT_SERVICE`: deve essere richiesto da un `PrintService`, per garantire che solo il sistema possa assegnarglielo
- `android.permission.CLEAR_APP_CACHE`: accesso alla rimozione della cache di applicazioni installate sul dispositivo
- `android.permission.MANAGE_EXTERNAL_STORAGE`: accesso ampio allo scoped storage nell'external storage, pensato per poche applicazioni che devono gestire file per conto degli utenti
- `android.permission.MANAGE_MEDIA`: accesso alla modifica e alla rimozione di file multimediali sul dispositivo o su supporti esterni senza la conferma dell'utente, è indispensabile prima possedere i permessi `MANAGE_EXTERNAL_STORAGE` o `READ_EXTERNAL_STORAGE`

I **permessi special** sono definiti dalla piattaforma o da un original equipment manufacturer (OEM), a volte detto "casa madre", e sono associati a particolari operazioni applicative. Spesso vengono fatti per proteggere l'accesso ad azioni particolarmente potenti, come disegnare su altre applicazioni.

Qualsiasi tipo di permesso richiesto da un'applicazione va dichiarato nel file `AndroidManifest.xml`, usando il tag `<uses-permission>`.

```
//Normal permission
<uses-permission android:name="android.permission.INTERNET" />

//Dangerous permission
<uses-permission android:name="android.permission.CAMERA" />
```

Una volta dichiarati i permessi all'interno del manifest, i permessi normal sono concessi in automatico dal sistema in fase di installazione dell'app per via della loro natura non pericolosa, mentre i permessi dangerous sono concessi in modi diversi in base alla versione di Android.

Originariamente, fino ad Android 5.1 (API level 22), i permessi dangerous venivano accettati in automatico quando l'utente installava l'applicazione. Essi apparivano nel momento dell'installazione in una finestra nello store, non c'era modo di selezionarne alcuni per rifiutarli e nemmeno potevano essere revocati in un momento successivo.

In seguito, da Android 6 (API level 23) ogni singolo permesso dangerous deve essere accettato a runtime dall'utente nel momento dell'utilizzo delle funzionalità che lo richiede. I permessi possono essere revocati selettivamente in ogni momento, se un permesso non è disponibile, può essere richiesto dall'utente. Come detto nelle buone norme per la privacy dell'utente, ogni richiesta pericolosa dovrebbe essere corredata da un messaggio in cui vengono spiegati i motivi per cui è necessario dare l'autorizzazione, per maggiore trasparenza.

Infine da Android 11 (API level 30) è stata aggiunta per i permessi dangerous la modalità "one-time" ("one-shot"), grazie alla quale gli utenti possono concedere il permesso per una sola esecuzione dell'app, permettendole di accedere ai dati o alle azioni sensibili finché l'utente la sta utilizzando, al successivo riavvio il permesso dovrà essere concesso di nuovo. Sempre in questa versione di Android si introduce l'ibernazione delle applicazioni non usate da alcuni mesi, cioè uno stato in cui il loro comportamento viene limitato, tra cui c'è la revoca dei permessi dangerous concessi a runtime.

Attraverso queste tappe è stato creato un modello più protettivo nei confronti della sicurezza dell'utente e allo stesso tempo flessibile verso le attività critiche. Tuttavia dal lato sviluppatore c'è una situazione più complessa da gestire, che coinvolge più scenari: tutti i permessi dangerous sono stati concessi, tutti sono stati negati, una parte è stata negata o revocata. L'applicazione deve contemplare tutti questi casi, eventualmente disabilitando quelle funzionalità per cui i permessi erano indispensabili, rimanendo sempre in grado di lavorare correttamente.

La classe `ActivityCompat`, inclusa in `AndroidX`, mette a disposizione il metodo `checkSelfPermission()` per verificare a runtime se un permesso è stato concesso, e il metodo `requestPermissions()` per richiedere a runtime uno o più permessi dangerous, dichiarati nel manifest, non ancora concessi. I risultati di tale richiesta vengono passati come argomenti al metodo `onRequestPermissionsResult()`, il quale si occupa di gestire il flusso del programma a seconda che il permesso venga fornito o meno.

```

//Si controlla se il permesso ACCESS_FINE_LOCATION è stato concesso o meno
if(ActivityCompat.checkSelfPermission(this, Manifest.permission.ACCESS_FINE_LOCATION) !=
    PackageManager.PERMISSION_GRANTED){

    //Se non è stato concesso si mostra una spiegazione
    if(ActivityCompat.shouldShowRequestPermissionRationale(this,
        Manifest.permission.ACCESS_FINE_LOCATION)){
        //Spiegazione dei motivi del permesso all'utente
    }

    //Richiesta del permesso
    ActivityCompat.requestPermissions(this,
        arrayOf(Manifest.permission.ACCESS_FINE_LOCATION), ID_REQUEST_PERMISSION_LOCATION)
}

...

//Metodo che riceve i risultati dell'invocazione di requestPermissions
override fun onRequestPermissionsResult(requestCode: Int, permissions: Array<String?>,
    grantResults: IntArray){

    if(requestCode == ID_REQUEST_PERMISSION_LOCATION){

        if(grantResults[0] != PackageManager.PERMISSION_GRANTED){
            //Permesso non concesso
            //Gestione della mancanza nelle funzionalità
        }

        else{
            //Permesso concesso
            //Accesso alla posizione
        }
    }
}

```

Un'applicazione dovrebbe richiedere il minor numero di permessi possibili, limitandosi a quelli indispensabili per le funzionalità che deve offrire. Va assolutamente evitata la dichiarazione di permessi mai utilizzati, o dismessi nelle successive versioni.

In particolare un minore accesso ai permessi sensibili riduce il rischio di usarli impropriamente e rende l'app meno vulnerabile agli attacchi esterni, in genere diretti alle informazioni riservate dell'utente.

L'approccio più sicuro, se possibile, prevede di progettare un'applicazione in modo che non richieda alcun tipo di permesso, soprattutto dangerous.

Nei casi in cui è realizzabile, non vanno aggiunti permessi per completare azioni che potrebbero essere completate da altre applicazioni. Basta un Intent per trasmettere la richiesta a un'altra applicazione che ha già i permessi necessari per compierla. Per esempio, se si deve creare un nuovo contatto, invece di richiedere il permesso dangerous WRITE_CONTACTS, si può delegare la responsabilità a un'applicazione che gestisce i contatti.

```
//Intent per deferire la creazione del contatto a un'app che gestisce i contatti
val contactIntent = Intent(Intent.ACTION_INSERT)
contactIntent.type = ContactsContract.Contacts.CONTENT_TYPE

//Se viene trovata un'app adatta
if (contactIntent.resolveActivity(packageManager) != null) {
    startActivity(contactIntent)
}
```

Nel caso in cui si debba ottenere un identificatore univoco, non vanno usati gli identificatori persistenti associati all'hardware, sensibili e che peraltro richiedono il permesso `READ_PRIVILEGED_PHONE_STATE` e che l'app sia in una whitelist di app privilegiate, come già detto nella sezione della privacy. Conviene usare un identificatore globale unico GUID.

```
var GUID = UUID.randomUUID().toString()
```

I permessi visti finora sono definiti dal sistema Android e coprono un vasto caso di situazioni, ma c'è anche la possibilità di creare nuovi permessi personalizzati, anche se è un caso abbastanza raro.

Conviene che tali permessi sfruttino il meccanismo di protezione basato sulla firma: i permessi signature sono trasparenti all'utente e accessibili solamente da applicazioni firmate dallo sviluppatore con lo stesso certificato, lo stesso dell'app che li ha definiti. Alternativamente possono essere dichiarati nel manifest con il tag `<permission>` e le app che li vogliono usare li aggiungono con il tag `<uses-permission>`.

Nella creazione di nuovi permessi dangerous va tenuto conto di diversi fattori: il permesso deve avere un nome espressivo della decisione richiesta all'utente, la stringa deve essere disponibile in diverse lingue, l'utente potrebbe decidere di non dare il permesso perché lo ritiene non chiaro o rischioso e, per ultimo, potrebbe essere richiesto un permesso di cui il creatore non è ancora stato installato. Di conseguenza creare permessi custom non è un'operazione semplice, conviene per quanto possibile usare quelli esistenti offerti dal sistema.

Comunicazione interprocesso

Finora sono state analizzate le tecniche con cui Android garantisce una gestione sicura dei dati manipolati dalle applicazioni, e come la protezione, assicurata dalla piattaforma, nei confronti dell'utente e dell'intero sistema passi attraverso i permessi.

Adesso si vogliono mettere in luce alcuni metodi per rafforzare la comunicazione tra le applicazioni, con particolare riguardo alla sicurezza nella trasmissione e ricezione dei dati.

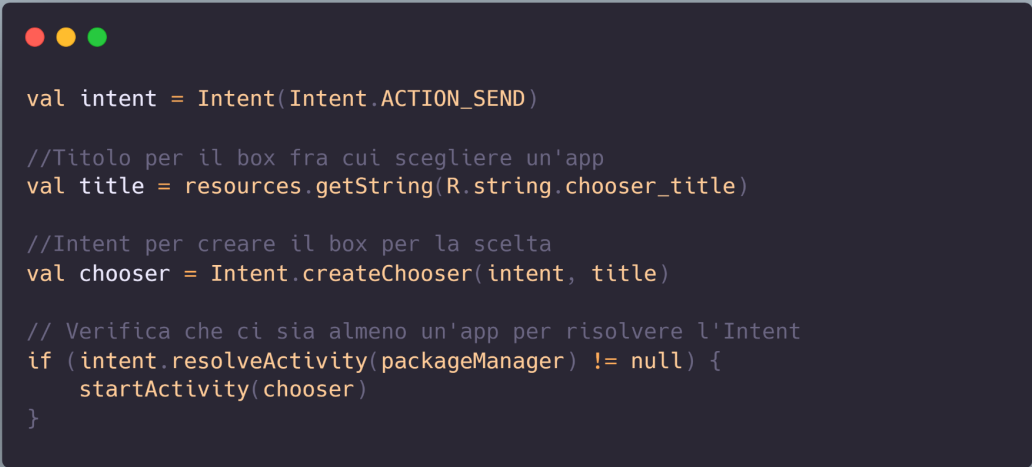
Come già evidenziato, Android gestisce i processi in spazi di indirizzamento diversi e un processo non può accedere direttamente alla memoria di altri processi (process isolation). Quindi un processo che vuole offrire servizi ad altri, solo attraverso dei meccanismi implementati dal sistema operativo è in grado di interagire esternamente con i processi e gestire dati condivisi. Questo concetto è la comunicazione inter-processo detta IPC. Android fornisce i seguenti meccanismi ad alto livello per gestire la IPC: Intent, Service, Binder o Messenger, BroadcastReceiver.

Uno dei principali meccanismi ad alto livello per la comunicazione inter-processo asincrona riguarda gli *Intent*. Un oggetto di tipo Intent non è altro che un messaggio usato per richiedere un'azione a un altro componente applicativo. Esistono due tipi di Intent: gli Intent espliciti specificano il nome completo della classe del componente, si usano in genere nelle proprie applicazioni per lanciare un'Activity o un Service di cui si conosce il nome; gli Intent impliciti non specificano il componente, bensì l'azione che deve essere compiuta da un altro componente di qualche app. Ad esempio, se un'app possiede un indirizzo e volesse mostrarlo su una mappa, non è necessario costruire un'Activity per fare ciò, basta creare una richiesta di visualizzazione usando un Intent implicito e il sistema si occupa di trovare e lanciare un'app adatta alla gestione delle mappe.

Il meccanismo degli Intent viene usato anche dal sistema per la diffusione di messaggi in broadcast, a tutte le app che si sono registrate per riceverli.

Un Intent esplicito garantisce più sicurezza, le informazioni vengono di certo trasmesse al componente specificato, nulla può andare storto.

Un Intent implicito può essere risolto da più applicazioni sul dispositivo, è Android che in automatico verifica gli Intent filter dichiarati nei manifest e filtra le app adatte per compiere una data azione. In ogni caso è buona norma mostrare un box in cui l'utente può scegliere in autonomia a quale app trasferire i dati, sulla base della fiducia che ci ripone.



```
val intent = Intent(Intent.ACTION_SEND)

//Titolo per il box fra cui scegliere un'app
val title = resources.getString(R.string.chooser_title)

//Intent per creare il box per la scelta
val chooser = Intent.createChooser(intent, title)

// Verifica che ci sia almeno un'app per risolvere l'Intent
if (intent.resolveActivity(packageManager) != null) {
    startActivity(chooser)
}
```

Per garantire una sicurezza più elevata, il trasmettitore dell'Intent implicito può assicurarsi che venga ricevuto solo dalle applicazioni che hanno un permesso specifico, specificando il nome del permesso alla creazione dell'Intent.

Dal lato sviluppatore qualsiasi componente applicativo, Activity, Service, Content Provider o Broadcast Receiver, se esportato (`android:exported="true"`), può ricevere dall'esterno solo Intent espliciti e solo alcuni Intent impliciti, in accordo con gli Intent filter definiti nel Manifest. Tuttavia tali filtri non garantiscono una forma di sicurezza, un'altra app malevola può potenzialmente lanciare lo stesso il componente usando un Intent esplicito, basta che in qualche modo ne abbia ricavato il nome.

Quindi se il componente deve essere usato solo all'interno della propria applicazione, è importante non esportarlo, dichiarando `android:exported="false"`. Altrimenti se è necessario esportarlo, per renderlo accessibile alle altre applicazioni, si deve usare una politica di maggior sicurezza basata sui permessi per limitare l'accesso. Si possono usare i permessi signature se la comunicazione avviene tra app firmate ugualmente dallo stesso sviluppatore.

Per minimizzare le trasmissioni di Intent poco sicuri, in fase di ricezione le informazioni aggiuntive immagazzinate come "extras" devono attraversare una fase di filtraggio, in cui vengono elaborate e analizzate. Soprattutto quando questi dati sono controllati dall'utente e non vengono validati correttamente, possono raggiungere componenti sensibili e compromettere l'applicazione. Molto comuni sono le tecniche di SQL injection, cross-site scripting (XSS) e le vulnerabilità delle applicazioni web.

Il *Service* è un altro strumento per la IPC, si tratta di un componente che permette a un'applicazione di eseguire operazioni di lunga durata in background, mentre l'utente non ci sta interagendo, o di fornire funzionalità utilizzabili da altre applicazioni.

Di default i Service non sono esportati, di conseguenza non possono essere usati esternamente da altre applicazioni. Se devono essere esportati, conviene per maggiore

sicurezza proteggerli definendo dei permessi custom, tali permessi poi devono essere dichiarati dalle applicazioni che vogliono interagire con il Service. Android mette a disposizione nella classe Context il metodo `checkCallingPermission()`, che riceve come parametro la stringa del nome del permesso e controlla se il processo chiamante il Service possiede il particolare permesso.

Altri due meccanismi di IPC in Android sono *Binder* e *Messenger*, nello specifico essi permettono il meccanismo di chiamata di procedura remota (RPC) tra processi client e server, il client può eseguire remotamente metodi del server come se fossero eseguiti localmente. Binder è basato sull'implementazione di una classe derivata da Binder che implementa *IBinder*, l'interfaccia base che descrive il protocollo astratto per interagire con un oggetto remoto. Invece Messenger incapsula un oggetto che implementa *IBinder* e realizza una comunicazione basata sulla trasmissione di messaggi tra i processi. Binder è limitata a processi all'interno dello stesso spazio di indirizzamento, questo non vale per Messenger. Binder e Messenger non sono componenti dichiarati nel manifest e quindi non possono essere protetti attraverso la gestione dei permessi, generalmente ereditano gli attributi dal Service o dell'Activity in cui sono implementati.

Se richiedono dei controlli di autenticazione o di accesso più stringenti, vanno scritti nel codice delle relative due classi. In particolare per verificare se il chiamante ha dichiarato il permesso necessario si può usare, come nel Service, il metodo `checkCallingPermission()` di Context.

Infine, l'ultimo protagonista della comunicazione inter-processo è il *Broadcast Receiver*, un componente applicativo che permette di ricevere eventi mandati in broadcast: questi eventi possono essere di interesse generale (es. riguardo la batteria, l'accensione o lo spegnimento di connessioni, etc) e quindi l'invio è riservato al sistema Android, oppure anche eventi custom inoltrati dalle singole applicazioni.

I messaggi mandati in broadcast sono trasmessi attraverso degli Intent, come già detto, e un Broadcast Receiver deve registrarsi per riceverli, specificando l'azione associata all'Intent desiderato e l'eventuale permesso per accedere a informazioni sensibili (permessi dangerous). Tale registrazione può essere fatta nel Manifest con l'attributo `<receiver>` al cui interno viene specificato un `<intent-filter>`, oppure in modo programmatico (obbligatorio per gli Intent impliciti da API level 26) nel codice dell'applicazione sempre creando un oggetto `IntentFilter`, in entrambi i casi nel manifest deve essere dichiarato anche l'uso del permesso associato.

Validazione dell'input e vulnerabilità del codice nativo

Uno dei problemi più comuni sulla sicurezza delle applicazioni è la validazione dei dati in entrata. La scelta di linguaggi type-safe riduce in parte questi problemi. Un linguaggio è detto type-safe quando gli errori di tipo vengono rivelati a tempo di compilazione o di esecuzione, con adeguati controlli. Android offre delle contromisure a livello di piattaforma, che permettono di limitare l'esposizione delle applicazioni a questi problemi ed è fortemente consigliato usarle ove possibile.

I problemi più comuni nella programmazione in codice nativo sono legati al buffer overflow, allo use-after-free, agli errori off-by-one. Buffer overflow è la scrittura oltre i limiti di

dimensione del buffer e la sovrascrittura di aree adiacenti di memoria, con il rischio che queste ultime siano utilizzate per altri scopi. È quindi molto probabile che si abbia una inconsapevole perdita di informazioni. Anche l'utilizzo di risorse deallocate è causa di questo. La deallocazione prevede di liberare uno spazio di memoria usato fino a quel momento, in quanto non più necessario. L'errore off-by-one infine è un errore di tipo logico e si verifica spesso nei loop, quando c'è un'iterazione in più del dovuto. Bisogna stare molto attenti quando si allocano risorse o si commettono operazioni di scrittura o di conteggio. Android, come gli altri sistemi operativi moderni, implementa delle tecnologie come la Address Space Layout Randomization (ASLR) e la Data Execution Prevention (DEP), per mitigare l'uso di una vulnerabilità creata questo tipo di errori, ma non risolve il problema.

Anche i linguaggi dinamici e basati su stringhe come SQL e JavaScript sono vulnerabili dal punto di vista della validazione dell'input. La scelta migliore per richiedere accesso a un database SQL o a un content provider è l'utilizzo di query parametrizzate. Limitare i permessi in sola lettura o sola scrittura è un buon modo per ridurre il rischio di problematiche. È consigliato inoltre controllare la struttura del formato dei dati e verificare che quelli in input la rispettino. Attuare una sostituzione di caratteri o cancellarne alcuni di specifici potrebbero sembrare delle buone soluzioni, ma in realtà sono pratiche da evitare.

La libreria crittografica di Android

Android mette a disposizione una robusta libreria per la crittografia e l'autenticazione dei dati che contiene sia le implementazioni degli algoritmi più moderni e sicuri, e sia quelle di algoritmi ormai deprecati per mantenere un certo grado di backward compatibility.

Per la generazione delle chiavi crittografiche sono offerte diverse classi a seconda del tipo di crittografia desiderata, in particolare citiamo:

- KeyGenerator per la generazione di chiavi simmetriche, spesso riferite nella documentazione Android come *segreti*. Alcuni algoritmi supportati sono AES, Blowfish e DES;
- KeyPairGenerator per la generazione di coppie di chiavi pubbliche e private. L'algoritmo più noto nella crittografia asimmetrica è decisamente RSA, ma la classe ha anche degli utilizzi per la firma di certificati digitali;
- KeyAgreement per la negoziazione di chiavi tra due entità collegate da un canale insicuro. I due unici algoritmi supportati sono Diffie-Hellman e la sua versione a curve ellittiche.

Per l'implementazione degli algoritmi è offerta invece un'unica classe, chiamata Cipher, che va opportunamente configurata prima dell'utilizzo. Normalmente, la configurazione della classe Cipher è composta da tre scelte: l'algoritmo di crittazione, la modalità di suddivisione a blocchi e il padding. Un esempio di configurazione compatta preso dalla documentazione di Android è il seguente: *AES/GCM/NoPadding*. La modalità di suddivisione a blocchi potrebbe comportare delle configurazioni aggiuntive, come ad esempio vettori di inizializzazione (IV nel codice, spesso chiamati *nonce* nella documentazione) o tag di autenticazione. La classe Cipher cercherà di fare del proprio meglio nel configurare queste opzioni aggiuntive in modo sicuro, ma quando non è possibile sarà responsabilità del

programmatore fornire ulteriori configurazioni sotto forma di oggetti `ParameterSpec` (per esempio, `GCMParameterSpec` per decriptare un messaggio con modalità GCM).

La configurazione di un algoritmo di crittografia può cambiare sensibilmente sia la complessità con la quale vengono gestiti i dati crittografati che la loro resistenza nei confronti di determinati tipi di analisi, e per un programmatore non è sempre detto che sia chiara la differenza tra le diverse modalità utilizzabili. Fortunatamente, la documentazione Android raccomanda chiaramente l'utilizzo di AES in modalità CBC o GCM con chiavi da 256 bit, entrambe modalità resistenti a numerosi attacchi crittografici conosciuti.

AES (Advanced Encryption Standard) è l'algoritmo più utilizzato per la crittografia simmetrica e, al momento, non esistono attacchi pratici che vadano a minare la sua efficacia, anche per chiavi più piccole di quella consigliata fino ad arrivare ai 128 bit. Tuttavia, il solo algoritmo AES non è sufficiente per criptare grandi quantitativi di dati: occorre infatti suddividere i dati in ingresso in blocchi (di lunghezza massima di 16 byte) ed eseguire delle ulteriori operazioni su di essi. È a questo scopo che esistono le modalità di suddivisione a blocchi: AES ne supporta diverse, alcune anche estremamente deboli, come la Electronic Code Book (ECB), e altre più moderne.

Le modalità Cipher Block Chaining (CBC) e Galois Counter Mode (GCM) suggerite, sono due modalità sicure che richiedono la generazione di un vettore di inizializzazione casuale per essere utilizzate correttamente. La classe `Cipher` si occupa in automatico della generazione del vettore e, dal punto di vista dello sviluppatore, occorre progettare l'applicazione in modo che ogni IV associato ad un messaggio cifrato venga salvato in memoria. Fortunatamente, il valore dell'IV non è un'informazione utile per eseguire un attacco e dunque può essere memorizzato in chiaro. Da un punto di vista analitico, la presenza di un IV generato casualmente permette di ottenere messaggi cifrati diversi a partire dallo stesso messaggio originale e della stessa chiave, rendendo il testo sicuro contro analisi di frequenza e attacchi dov'è noto il testo in chiaro.

Inoltre, la modalità GCM offre un livello di sicurezza ancora più alto sui dati crittografati al costo di complicare leggermente il codice di configurazione del Cipher. La differenza sostanziale tra GCM e CBC è l'aggiunta di un messaggio di autenticazione alla fine del testo crittografato. Questo permette di scardinare possibili attacchi più avanzati, come il padding oracle attack, che modificano il testo crittografato e traggono nuove informazioni dalla sua (spesso fallita) decriptazione. In aggiunta, la modalità GCM è più performante poiché permette di eseguire le operazioni sui blocchi in parallelo e non necessita di alcun padding per essere utilizzata correttamente.

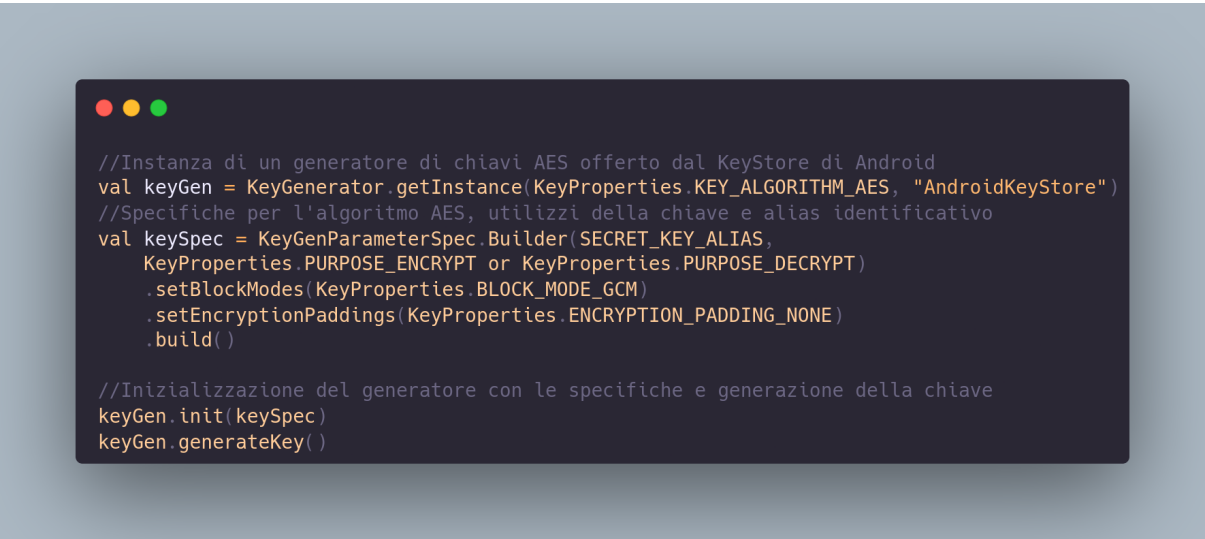
Gli algoritmi di hashing e di autenticazione di messaggio sono due ulteriori primitive della libreria crittografica di Android e possono essere utilizzate in aggiunta alla crittografia per verificare, rispettivamente l'integrità e l'autenticità di un messaggio. Android implementa la famiglia di algoritmi SHA per l'hashing crittografico, mentre utilizza la famiglia HMAC (che basa il suo funzionamento su SHA) nel calcolo dei tag di autenticazione.

Infine, per andare a completare una panoramica sulla sicurezza della crittografia in Android occorre parlare del `KeyStore`. Il `KeyStore` è un'interfaccia software ad uno storage dedicato al salvataggio di chiavi crittografiche e di certificati di autenticazione presente nei moderni dispositivi embedded. Dal punto di vista hardware, questo storage è un modulo del dispositivo con una propria CPU dedicata, dotato di memoria e che implementa un algoritmo

sicuro per la generazione di numeri casuali. I dettagli implementativi del modulo hardware variano da produttore a produttore ma i punti principali che li accomunano sono i seguenti:

- Sicurezza hardware: il modulo è progettato per evitare l'estrazione forzata dei suoi contenuti. Inoltre la chiave viene direttamente generata in modo casuale dalla CPU interna, andando così a minimizzare il rischio di creazione di una chiave debole;
- Sicurezza software: ogni chiave ha associata ad essa un insieme di parametri che ne determinano gli utilizzi, da chi è posseduta ed un alias per poter essere richiamata. Ogni tentativo di accesso alla chiave da parte di altre applicazioni, o anche di uso improprio da parte dell'applicazione proprietaria, verrebbe bloccato e genererebbe una `SecurityException`.

Per poter generare una chiave all'interno del `KeyStore` occorre specificarlo come provider ad un generatore di chiavi, come nella figura seguente.



```
//Istanza di un generatore di chiavi AES offerto dal KeyStore di Android
val keyGen = KeyGenerator.getInstance(KeyProperties.KEY_ALGORITHM_AES, "AndroidKeyStore")
//Specifiche per l'algoritmo AES, utilizzi della chiave e alias identificativo
val keySpec = KeyGenParameterSpec.Builder(SECRET_KEY_ALIAS,
    KeyProperties.PURPOSE_ENCRYPT or KeyProperties.PURPOSE_DECRYPT)
    .setBlockModes(KeyProperties.BLOCK_MODE_GCM)
    .setEncryptionPaddings(KeyProperties.ENCRYPTION_PADDING_NONE)
    .build()

//Inizializzazione del generatore con le specifiche e generazione della chiave
keyGen.init(keySpec)
keyGen.generateKey()
```

CONCLUSIONE

In questo lavoro abbiamo evidenziato la storia e l'evoluzione del sistema operativo Android dal punto di vista della sicurezza. In passato Android aveva sottovalutato questo aspetto ed era poco restrittivo nei confronti delle applicazioni, trascurando la tutela verso l'utente. Con il passare del tempo si è prestata sempre più attenzione alla privacy e alla gestione di informazioni sensibili, aggiungendo nuove funzionalità e fornendo diversi strumenti.

E' gradualmente aumentata la trasparenza e la protezione dei dati nei confronti dell'utente. In particolare si è creato un meccanismo più complesso e sicuro nella concessione dei permessi, che avviene soltanto quando è necessario. In precedenza i permessi venivano concessi in automatico con l'installazione e questo comportamento superficiale provocava poca chiarezza all'interno delle applicazioni. Inoltre i permessi diventano revocabili, quindi un utente può ritirare il proprio consenso nei confronti di un permesso in qualsiasi momento. Vengono poi aggiunte migliorie per quanto riguarda i servizi di localizzazione e la regolamentazione dell'accesso ai dati per le applicazioni in background.

Anche dal lato dello sviluppatore c'è stata una cospicua evoluzione in ambito sicurezza. Sono stati migliorati degli aspetti riguardo la crittografia, l'organizzazione della memoria e il salvataggio dei dati e dei file. Gli strumenti forniti per lo sviluppo risultano quindi più completi e coprono più casi d'uso. La documentazione di conseguenza è stata ampliata e resa più comprensibile.

Sitografia:

- <https://www.treccani.it/enciclopedia/diritto-alla-riservatezza/>
- <https://www.zerounoweb.it/analytics/data-management/sicurezza-informatica-cioe-disponibilita-integrita-e-riservatezza-dei-dati/>
- <https://www.ibm.com/it-it/topics/mobile-security>
- <https://developer.android.com/privacy/best-practices>
- <https://support.google.com/googleplay>
- <https://source.android.com/security/app-sandbox>
- https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/6/html/security-enhanced_linux/chap-security-enhanced_linux-introduction
- <https://developer.android.com/training/articles/security-tips>
- <https://owasp.org/www-community/attacks/Cryptanalysis>
- <https://developer.android.com/guide/topics/security/cryptography>
- <https://developer.android.com/training/articles/keystore>

Bibliografia:

- S. Rodotà, voce Riservatezza, in Enciclopedia Italiana Treccani,
- Ryan Farmer, "A Brief Guide to Android Security", 2012.