UNIVERSITÀ DEGLI STUDI DI GENOVA
Facoltà di Ingegneria

Corso di Laurea Magistrale in Ingegneria Informatica

FCDroid: un tool per identificare la frame confusion nelle applicazioni mobile

Relatore:
Prof. Alessio Merlo

Correlatore:
Dott. Luca Verderame

Candidato:
Davide Caputo
Matricola n. 3928125

Anno Accademico 2017 – 2018

Indice

Acronimi

API Application Programming Interface

ART Android Runtime

AST Abstract Syntax Tree

CSP Content Security Policy

CSS Cascading Style Sheets

DOM Document Object Model

GID Group Identifier

HTML HyperText Markup Language

IPC Inter Process Communication

NDK Native Development Kit

OS Operating System

SDK Software Development Kit

SOP Same-Origin Policy

UID User Identifier

URL Uniform Resource Locator

XSS Cross-Site Scripting

Elenco delle figure

Elenco delle tabelle

1 Introduzione

Negli ultimi anni le aziende si sono dotate di sistemi digitali per offrire contenuti e servizi ai propri clienti e dipendenti tramite contenuti web e mobile. Il mondo dei sistemi operativi mobile è però eterogeneo (Android, Android Wear, iOS, WhatchOS, etc.) ed ognuno di essi ha il proprio Software Development Kit (SDK) e il proprio linguaggio di programmazione. Per un'azienda che ha l'obiettivo di raggiungere un maggior numero di utenti diventa, quindi, molto costoso sviluppare la stessa applicazione per ogni Operating System (OS), in quanto richiede di mettere in piedi processi di sviluppo e mantenimento spesso costosi in termini di risorse umane.

Per questo motivo spesso le piccole o medie imprese spesso preferiscono sviluppare un'applicazione per un solo OS (es. Android invece che iOS). Queste applicazioni, chiamate native, portano con sé diversi vantaggi, tra cui quello di poter accedere a tutte le *feature* del dispositivo (es. GPS, Contatti, File System, la Gestione del Touch, etc.).

La controparte delle applicazioni native sono le web app. Le web app sono sviluppate usando le tecnologie standard del web (HyperText Markup Language (HTML), Cascading Style Sheets (CSS), JavaScript) ed essendo eseguite in un web browser possono essere utilizzate su qualsiasi smartphone, indipendentemente dal OS sottostante. Questo tipo di applicazioni ha però dei limiti: al contrario delle applicazioni native, esse possono accedere soltanto alle *feature* del device supportate da HTML5 (es. GPS, Camera, LocalStorage).

Per superare i limiti imposti dalle app web (accesso limitato alle *feature* del device) e quelle delle applicazioni native (riscrittura del codice per tutti gli OS mobile) è stato introdotto negli ultimi anni un paradigma di sviluppo misto chiamato **ibrido**. Le applicazioni ibride sono basate sulle tecnologie standard del web, che le rendono eseguibili su ogni smartphone (riducendo i costi di sviluppo) e tramite specifici *plugin* permettono di accedere a tutte le *feature* del device.

Tuttavia le applicazioni ibride pongono diverse sfide dal punto di vista della sicurezza. L'unione dei due mondi comporta infatti una maggiore esposizione a possibili rischi di sicurezza che sono nel caso delle applicazioni ibride provenienti sia dal mondo mobile (Intent Hijacking, storage insicuro dei dati, canali di comunicazione insicuri) che da quello web (Cross-Site Scripting (XSS), Injection (SQL, NoSQL), etc.).

Uno dei problemi che nasce dall'unione di questi mondi è chiamato **Frame Confu-**

sion (Sezione 5), una vulnerabilità che non è presente né nel mondo web né in quello mobile se presi distintamente. Tale vulnerabilità è legata all'utilizzo del componente HTML iframe, spesso utilizzato per contenere *advertisement*, e di alcune API del componente WebView. Queste API permettono a codice JavaScript, contenuto all'interno della WebView, di invocare codice nativo (nel caso di Android codice Java). Il problema è che il componente WebView non riesce a riconoscere da quale dominio avvenga tale chiamata, e quindi, diventa possibile invocare codice nativo anche dall'interno di iframe che contengono annunci pubblicitari.

L'obiettivo di questa tesi è quello di proporre una metodologia per individuare la vulnerabilità della Frame Confusion in maniera automatica. La metodologia proposta combina tecniche di analisi statica e dinamica per superare i limiti proposti da entrambe le metodologie.

Si è quindi sviluppato un prototipo per Android, FCDroid, che è in grado di verificare in maniera automatica la presenza della Frame Confusion assicurando un alto grado di precisione e in tempi ragionevoli.

La necessità di sviluppare questo strumento nasce dal fatto che, nonostante nella letteratura scientifica la Frame Confusion sia una problematica nota, attualmente non esiste ancora uno strumento automatico in grado di verificarne la presenza. Il bisogno di uno strumento automatico è elevato perché verificare tale vulnerabilità a mano è un processo lungo e complesso.

L'implementazione della metodologia in FCDroid ha permesso, inoltre, di validare l'efficacia e la scalabilità della stessa, che è stata utilizzata per analizzare automaticamente le prime 10.000 applicazioni prese dal Google Play Store. Grazie all'utilizzo di FCDroid, sono state inoltre individuate due exploit ai danni di due applicazioni molto diffuse, in quanto scaricate collettivamente da più di 11 milioni di utenti nel mondo.

1.1 Struttura delle tesi

La tesi è organizzata nel seguente modo: viene inizialmente fatta una piccola introduzione sul lavoro che si vuole svolgere, dopodiché viene illustrata la metodologia sviluppata e gli strumenti utilizzata per la sua implementazione, infine vengono esposti i risultati e le considerazioni finali su eventuali sviluppi futuri.

La Sezione 1 offre una panoramica sugli argomenti trattati in questo lavoro di tesi, esponendone anche gli obiettivi finali.

In Sezione 2 viene presentato sinteticamente il sistema operativo Android fornendo una panoramica sull'architettura del sistema e i principali meccanismi di sicurezza dello stesso.

Nella Sezione 3 viene esposto il componente WebView, analizzandone le API di maggior interesse e i relativi problemi di sicurezza che il suo utilizzo comporta

La Sezione 4 illustra l'architettura delle applicazioni ibride. Viene quindi analizzata con dettaglio il middleware Apache Cordova e le implicazioni di sicurezza che si generano dall'unione del mondo mobile e del mondo web.

In Sezione 5 viene esposta dettagliatamente la Frame Confusion e i possibili scenari d'attacco e le contromisure esistenti.

La Sezione 6 sezione mostra le attuali tecniche di analisi presentate dalla comunità scientifica. Viene quindi fatta una distinzione tra le tecniche di analisi statica e quelle di analisi dinamica. Infine vengono analizzate le attuali soluzioni di analisi automatica per le applicazioni ibride.

In Sezione 7 viene esposta nel dettaglio la metodologia adottata in questo lavoro di tesi. Inizialmente è stato necessario individuare le cause scatenati della Frame Confusion, dopodiché viene illustrato e spiegato l'algoritmo utilizzato per identificate in maniera automatica la vulnerabilità. Infine viene esposta l'architettura finale di FCDroid.

La Sezione 8 descrive l'implementazione di FCDroid per le applicazioni Android, discutendo le scelte implementative e gli strumenti utilizzati.

vengono esposti dettagliatamente i due moduli di analisi. Vengono quindi analizzati nel dettaglio gli strumenti utilizzati.

La Sezione 9 riporta i risultati ottenuti dall'utilizzo di FCDroid su un dataset composto da 8.320 applicazioni reali e un esempio di attacco su un'applicazione reale.

Infine in Sezione 10 vengono fatte alcune considerazioni sui risultati ottenuti durante lo svolgimento della tesi, esponendo eventuali spunti per lavori futuri per continuare questo lavoro di tesi.

2 Android

Android [1] è un sistema operativo per dispositivi mobile basato sul kernel *Linux*, sviluppato da Google e dall' Open Handset Alliance (OHA) [2]. È basato su un'architettura a più livelli di astrazione, come visibile in Fig. 1.

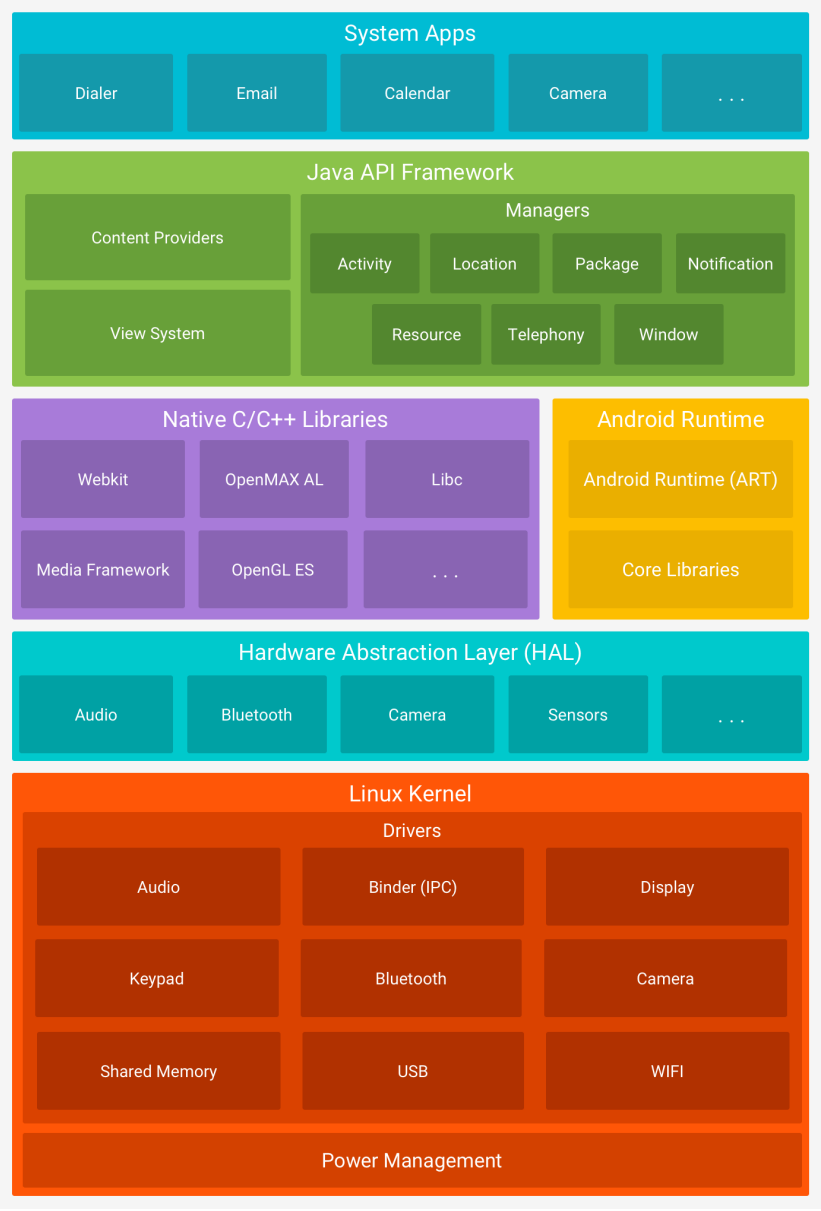


Figura 1: Architettura Android

Le fondamenta dello *stack* Android sono costituite dal **kernel Linux**, il cui utilizzo

[1] `https://developer.android.com/guide/platform/`
[2] `https://www.openhandsetalliance.com/`

permette ad Android di sfruttarne i principali meccanismi di sicurezza: 1) un modello basato sui permessi, 2) l'isolamento e la gestione dei processi, 3) la gestione della memoria e, 4) la gestione dell'Inter Process Communication (IPC).

Sopra il livello kernel è presente l'**Hardware Abstraction Layer (HAL)**, che fornisce interfacce standard per esporre le funzionalità hardware del dispositivo al livello superiore **Java API Framework**. Sopra l'HAL si trovano l'**Android Runtime** e il livello costituito dalle **librerie native**.

L'**Android Runtime** comprende la macchina virtuale, prima della versione 5.0 di Android era la *Dalvik virtual machine*, a partire da quella versione invece è stata sostituita con l'Android Runtime (ART). Questa macchina virtuale è una versione modificata della Java Virtual Machine ottimizzata per essere eseguita su dispositivi con poche risorse fisiche. La macchina virtuale esegue il codice di tutte le applicazione Android in formato *Dalvik executable* `.dex`, questo è un formato speciale di *bytecode* progettato appositamente per Android e ottimizzato per un ingombro minimo di memoria. In questo livello sono inoltre presenti le *Core Libraries* che permettono di accedere a tutte le funzionalità del linguaggio di programmazione Java.

Il livello costituito dalle **librerie native** a è costituito da una serie di servizi e librerie precompilate in C/C++ che sono propedeutiche alle applicazioni e ai servizi di sistema. Queste librerie sono rese disponibili da Android attraverso Native Development Kit (NDK). Esempi di librerie sono Libc, OpenGL ES, etc.

Sopra il livello di ART e le librerie risiede il **Java API Framework**, dove sono presenti tutte le Application Programming Interface (API) di cui si ha bisogno per creare applicazioni Android, fornendo un insieme di componenti riutilizzabili per sviluppare nuove applicazioni. Tra i servizi messi a disposizione vi sono:

- Il **Resource Manager** che fornisce l'accesso alle risorse dell'applicazione come stringhe, immagini e file di layout.
- Il **Notification Manager** abilita tutte le applicazioni a mostrare notifiche nella barra di stato dell'applicazione.
- L'**Activity Manager** gestisce il ciclo di vita delle applicazioni.
- Il **Content Providers** abilita le applicazioni ad accedere a dati provenienti dalle altre applicazioni, come i contatti, o di condividere i propri

In cima allo stack è presente il **livello applicativo**. Il livello applicativo include applicazioni utente e quelle di sistema come ad esempio applicazioni per e-mail, SMS, Calendario e Browser Internet, Chiamate etc.

2.1 Sicurezza Android

Con l'avvento degli smartphone, i problemi di privacy si sono moltiplicati rispetto ai computer tradizionali, i dispositivi mobile, infatti, processano e immagazzinano una grande mole di dati sensibili come: posizione GPS, SMS, contatti, dati derivanti da sensori biometrici etc.

Il sistema operativo più utilizzato per gli smartphone è Android con oltre il 76%[3] di diffusione. Android permette l'installazione di applicazioni provenienti da fonti ufficiali e no, quindi la necessità di proteggere le informazioni sensibili dell'utente è un punto determinante per la sua privacy.

Risulta quindi indispensabile avere meccanismi di sicurezza adeguati per prevenire che applicazioni maligne possano mettere a rischio il device o accedere a fotocamera, microfono o altri sensori, compromettendo dunque l'integrità di applicazioni legittime. In Android, questo meccanismo di prevenzione è implementato in due modi:

- La **Sandbox**,
- I **Permessi**.

2.1.1 Application Sandbox

Android, essendo fondato su di un kernel Linux (Sezione 2), ne eredita anche le caratteristiche in termini di sicurezza. Linux è un sistema multiuser e il suo kernel è in grado di isolare le risorse di un utente da quelle di un altro allo stesso modo in cui isola i processi. In un sistema Linux, infatti, un utente non può accedere ai file di un altro (salvo dietro concessione esplicita), e ogni processo viene eseguito con l'identità dell'utente che lo ha avviato (User Identifier (UID), Group Identifier (GID)).

Android sfrutta questo isolamento degli utenti assegnando un diverso UID ad ogni applicazione in fase di installazione. Inoltre, ad ogni applicazione viene assegnata una directory dati apposita in cui può leggere o scrivere solo l'applicazione specifica. Tutte

[3] `http://gs.statcounter.com/os-market-share/mobile/worldwide`

le applicazioni sono quindi in **sandbox**, sia a livello di processo (ognuna eseguita in un processo dedicato) che a livello di file (ognuna ha una propria directory privata).

2.1.2 Permessi

Le applicazioni, come spiegato nella Sezione 2.1.1, sono soggette al meccanismo del sandbox e per impostazione predefinita possono accedere unicamente ai propri file e a un set limitato di servizi di sistema. Per interagire con il sistema e con le altre applicazioni possono usare i permessi.I permessi sono stringhe che denotano la capacità di eseguire una specifica azione (accedere ad una risorsa fisica, ai dati condivisi, Internet etc.) nei confronti del OS. I permessi vengono definiti dell'applicazione all'interno del file `AndroidManifest.xml` [4], un file di configurazione presente in ogni applicazione e che ne descrive la struttura. I permessi, inoltre, possono essere divisi in diversi gruppi, caratterizzati dal loro livello di protezione, che a sua volta identifica il rischio implicito dello stesso:

- **normal**, valore predefinito che definisce un permesso a basso rischio. Questo tipo di permessi viene concesso automaticamente senza chiedere conferma all'utente (es. `ACCESS_NETWORK_STATE`, `GET_ACCOUNTS`).
- **dangerous**, i permessi con questo livello di protezione consentono l'accesso ai dati utente o permettono un qualche tipo di controllo sul dispositivo (es. `CAMERA`, `READ_SMS`). Vengono concessi solo dopo la conferma dell'utente.
- **signature**, questo tipo di permesso viene concesso unicamente alle applicazione firmate con la stessa chiave dell'applicazione che dichiara il permesso. Questo è il livello di permesso più "forte", perché richiede il possesso di una chiave di crittografia controllata esclusivamente dal proprietario dell'applicazioni (es. `NET_ADMIN`, `ACCESS_ALL_EXTERNAL_STORAGE`). Sono concessi senza notificare l'utente.
- **singatureOrSystem**, permessi concessi alle applicazioni che sono parte dell'immagine del sistema operativo di Android.

[4] `https://developer.android.com/reference/android/Manifest.permission`

3 WebView

Per quanto riguarda la visualizzazione di contenuti web, Android offre diverse possibilità. Come mostrato in Fig. 2, si può utilizzare una applicazione Browser (Android Browser, Chrome, Opera, etc.) oppure una WebView.

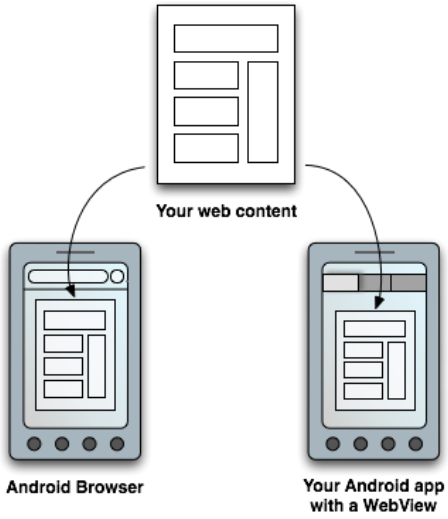


Figura 2: Contenuti Web su Android

La WebView è un componente dell'interfaccia utente che può essere incorporato all'interno di un'applicazione mobile per interpretare HTML, CSS ed eseguire codice JavaScript (chiamato WebView in Android, WKWebView in iOS, WebBrowser in Windows Phone).

Esistono diversi scenari nei quali il suo impiego può essere d'aiuto; il più comune è quando si vogliono fornire delle informazioni che hanno bisogno di aggiornamenti frequenti, come l'*end-user agreement* o la guida per l'utente. Utilizzando infatti una WebView, invece di aggiornare l'applicazione continuamente, si può creare una pagina che la contenga e che viene utilizzata per mostrare il contenuto di una risorsa online, rendendo l'aggiornamento del documento in questione indipendente dall'aggiornamento dell'applicazione.

Il browser e la WebView hanno un'architettura ad alto livello simile visualizzabile in Fig. 3. L'elemento che ha la responsabilità di visualizzare il contenuto è il *rendering engine*. Ogni browser ne ha uno: Internet Explorer usa Trident, Firefox usa Gecko, Safari usa WebKit, Edge usa EdgeHTML (un fork di Trident), Chrome invece utilizza Blink (un fork di WebKit). Il motore di rendering inizia ottenendo la risorsa richiesta

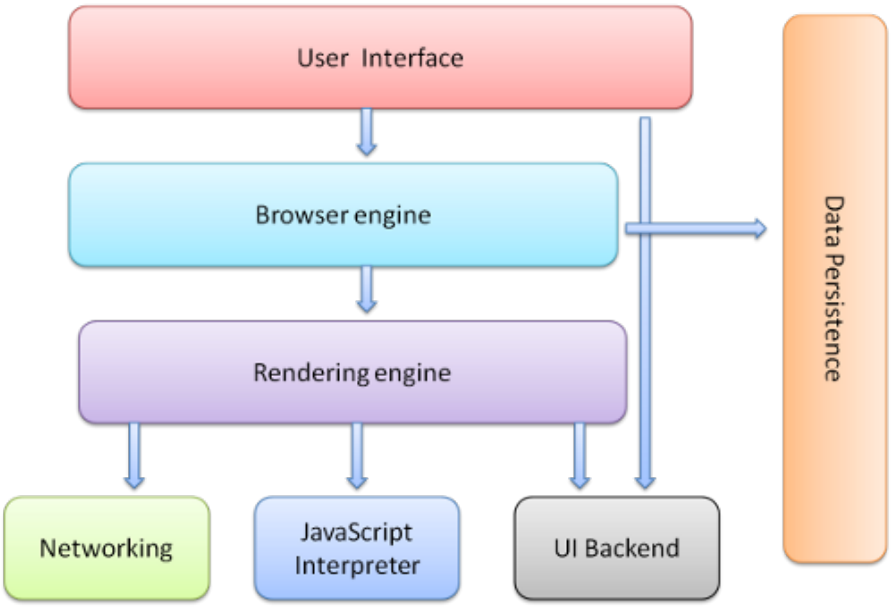


Figura 3: Architettura ad alto livello di un browser

dal livello di rete e dopo segue il flusso in Fig. 4 per infine visualizzare il contenuto sul display.

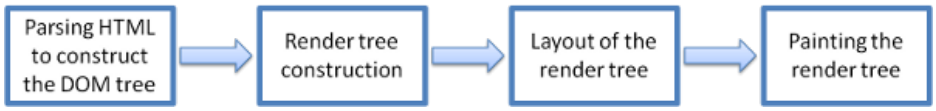


Figura 4: Flow rendering engine

Nel corso degli anni, il componente WebView in Android ha avuto diversi aggiornamenti. Fino alla versione 4.3 Jelly Bean (API 18), le WebView si basavano sul motore di rendering WebKit [5], progetto open source di Apple e utilizzato nel loro browser Safari. A partire dalla versione 4.4 di Android, è stata introdotta una nuova versione del componente WebView basata su Chromium, un progetto sempre open source ma di Google, che utilizza come motore di rendering Blink [6] un fork di WebKit. Questo aggiornamento ha reso la WebView più performante e compatibile con gli standard più moderni del web come HTML5, CSS3 e JavaScript, ma soprattutto sono state risolte alcune vulnerabilità [7] [1]. A partire da Android 5.0, la WebView è diventata un'applicazione di sistema, così da permettere aggiornamenti della stessa indipendentemente dagli aggiornamenti del firmware [8].

[5] https://webkit.org/
[6] https://www.chromium.org/, https://www.chromium.org/blink
[7] https://www.cvedetails.com/cve/cve-2012-6636
[8] https://play.google.com/store/apps/details?id=com.google.android.webview&hl=it

3.1 API WebView

Per ottenere una migliore interazione tra le applicazioni e il loro *embedded* "browser", il componente WebView fornisce diverse API. Queste permettono a metodi all'interno delle applicazioni di invocare o essere invocati da codice JavaScript appartenente alle pagine web caricate nella WebView e di intercettare ed eventualmente modificare eventi all'interno delle pagine.

Una delle API più interessanti è sicuramente *addJavascriptInterface*, metodo che fornisce al codice JavaScript caricato nella WebView un meccanismo per invocare codice Java all'interno dell'applicazione Android.

La controparte di questa API, cioè un metodo che dà la possibilità di invocare codice JavaScript all'interno dell'applicazione, è *evaluateJavascript* reso disponibile a partire dall'API 19, mentre nelle versioni precedenti dal metodo *loadUrl*. Quest'ultimo viene usato anche per caricare eventuali risorse online o in locale, identificate da un'Uniform Resource Locator (URL). Una panoramica della struttura di un' applicazione Android con WebView è illustrata in Fig. 5.

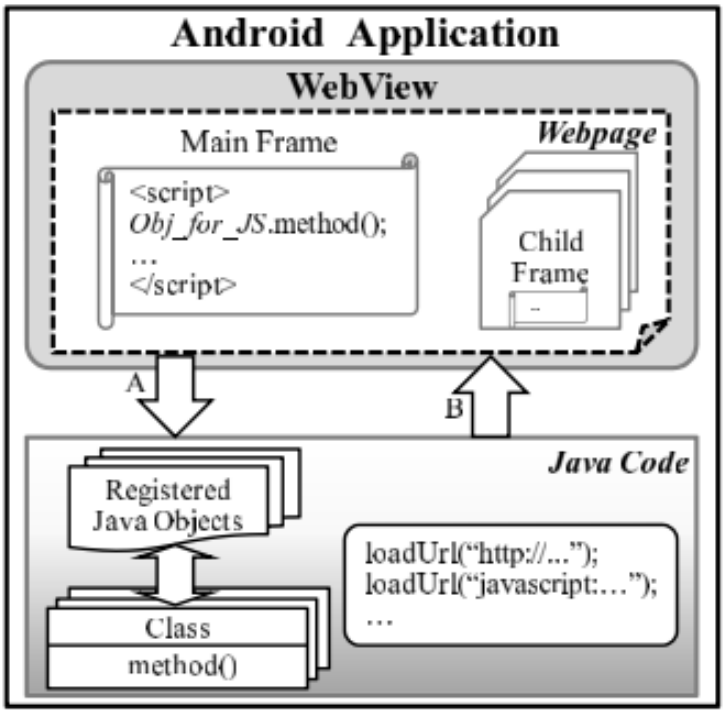


Figura 5: Panoramica applicazioni Android con WebView

3.2 Sicurezza WebView

I metodi elencati nella Sezione 3.1 possono essere molto utili, ma portano diversi problemi di sicurezza.

Le WebView, come già accennato, permettono l'esecuzione di codice JavaScript. Questa funzionalità di default è disattivata per motivi di sicurezza, ma per sfruttare al massimo le funzionalità del componente molto spesso la sua esecuzione viene abilitata, permettendo così ad ogni pagina caricata all'interno della WebView di eseguire codice sia legittimo che malevolo. L'esecuzione di codice JavaScript rende inoltre l'applicazione vulnerabile ad eventuale XSS [9], una vulnerabilità ben nota nel mondo web.

Il maggior rischio deriva però dalla possibilità di invocare codice Java da JavaScript eseguito all'interno della WebView (attraverso il "bridge" costruito dal metodo *addJavascriptInterface*). Questo metodo, infatti, viola il modello di sandbox adottato da tutti i browser, che ha lo scopo di raggiungere sostanzialmente due obiettivi: isolare le pagine web dal sistema e isolare le pagine web di un'origine da quella di un'altra. Il secondo obiettivo è principalmente imposto dal **Same-Origin Policy (SOP)** [10], un concetto fondamentale presente nel mondo della sicurezza web. Quando un'applicazione usa *addJavascriptInterface* per attaccare un'interfaccia alla WebView crea sostanzialmente dei fori nella sandbox. L'utilizzo di questo "bridge" è inoltre globale, cioè ogni pagina caricata all'interno della WebView che ha "abilitato" questa interfaccia può accedervi. Questo metodo è soggetto di studio in molti lavori come [2, 1, 3].

L'utilizzo delle API e delle WebView comporta, dunque, alcuni benefici in fatto di funzionalità ma al contempo anche nuovi problemi di sicurezza non presenti nel mondo mobile (XSS), o addirittura ne inserisce di nuovi non presenti né nel mondo mobile né nel mondo web, come la **Frame Confusion**, approfondita nel Capitolo 5.

Nel corso dell'evoluzione del sistema Android sono stati fatti alcuni miglioramenti di sicurezza al componente WebView. Inizialmente, per esempio, era possibile invocare qualsiasi metodo appartenente all'interfaccia rendendo l'integrazione tra JavaScript e Java estremamente pericolosa. Tuttavia a partire dalla versione 4.2 di Android questo è stato limitato all'invocazione dei soli metodi con l'annotazione *JavascriptInterface*. Nessun di questi miglioramenti però, hanno mai permesso di risolvere criticità di questo

[9] `https://www.owasp.org/index.php/Cross-site_Scripting_(XSS)`
[10] `https://www.w3.org/Security/wiki/Same_Origin_Policy`

componente, che risulta tutt'ora un problema aperto.

4 Applicazioni Ibride

Le applicazioni ibride sono nate per superare i limiti imposti dalle applicazioni native e dalle applicazioni web. Come già accennato nel Capitolo 1, le applicazioni ibride riescono a sfruttare i vantaggi di entrambe le tecnologie: essendo sviluppate utilizzando le tecnologie del web possono essere eseguite su ogni smartphone indipendentemente dall'OS sottostante ed essendo basate su plugin specifici riescono ad accedere a tutte le *feature* del device. Un altro punto a loro favore è rappresentato dai costi ridotti di sviluppo: il core di un'applicazione (la parte che gestisce l'interazione utente), essendo scritto in JavaScript, non viene riscritto per ogni OS e pertanto può essere utilizzato su ogni piattaforma, riducendo così i tempi di sviluppo, il personale impiegato e quindi i costi.

Allo stesso tempo però le applicazioni ibride introducono diverse difficoltà. Molte di esse sono sviluppate utilizzando differenti linguaggi di programmazione e quindi con semantiche diverse. Per esempio quelle per Android possono essere sviluppate in JavaScript per la parte di interazione con l'utente e in Java per accedere alle *feature* del device. L'unione di questi due linguaggi, il primo (JavaScript) estremamente dinamico e non tipizzato, il secondo (Java) statico e tipizzato, in continua comunicazione, rende lo sviluppo di questo tipo di applicazioni vulnerabile ad errori di programmazione molto frequenti.

L'architettura della applicazioni ibride si può dividere in due parti comunicanti tra di loro:

- WebView, responsabile di processare le pagine HTML e di eseguire il codice JavaScript.
- Nativa, che ha il compito di accedere alle feature del device come GPS, Camera, etc.

Esistono diversi framework per sviluppare applicazioni ibride, come ad esempio WebMarmalade, Ionic, Intel XDK, Framework7 e Xamarin [11]. Il più famoso e il più utilizzato al momento è Apache Cordova[12] vi sono oltre il 7%[13] di applicazioni presenti sul Google Play Store sviluppate utilizzando questo framework.

[11] `https://marmalade.shop/en/`, `https://ionicframework.com/`, `https://software.intel.com/en-us/xdk`, `https://framework7.io/`, `https://visualstudio.microsoft.com/xamarin/`

[12] `https://cordova.apache.org/`

[13] `https://www.appbrain.com`

Apache Cordova è nata dalla distribuzione originale chiamata PhoneGap[14]. Le differenze che vi sono fra i due, sono da ricercarsi negli strumenti messi a disposizione, siccome entrambi sono basati sulla stessa libreria: Apache Cordova. I diversi strumenti messi a disposizione da PhoneGap permettono la creazione delle applicazioni anche in cloud permettendo, inoltre, una facile collaborazione.

Un altro framework che utilizza la libreria Apache Cordova è Ionic. A differenza però di PhoneGap e Apache Cordova utilizza anche AngularJS[15], un framework JavaScript.

4.1 Libreria Apache Cordova

La libreria Apache Cordova, utilizzata da diversi framework cross-platform come Ionic, è un esempio di middleware. Un middleware non è altro che un software "intermediario" tra l'applicazione e il kernel. Questa libreria, infatti, permette a sviluppatori di diversi

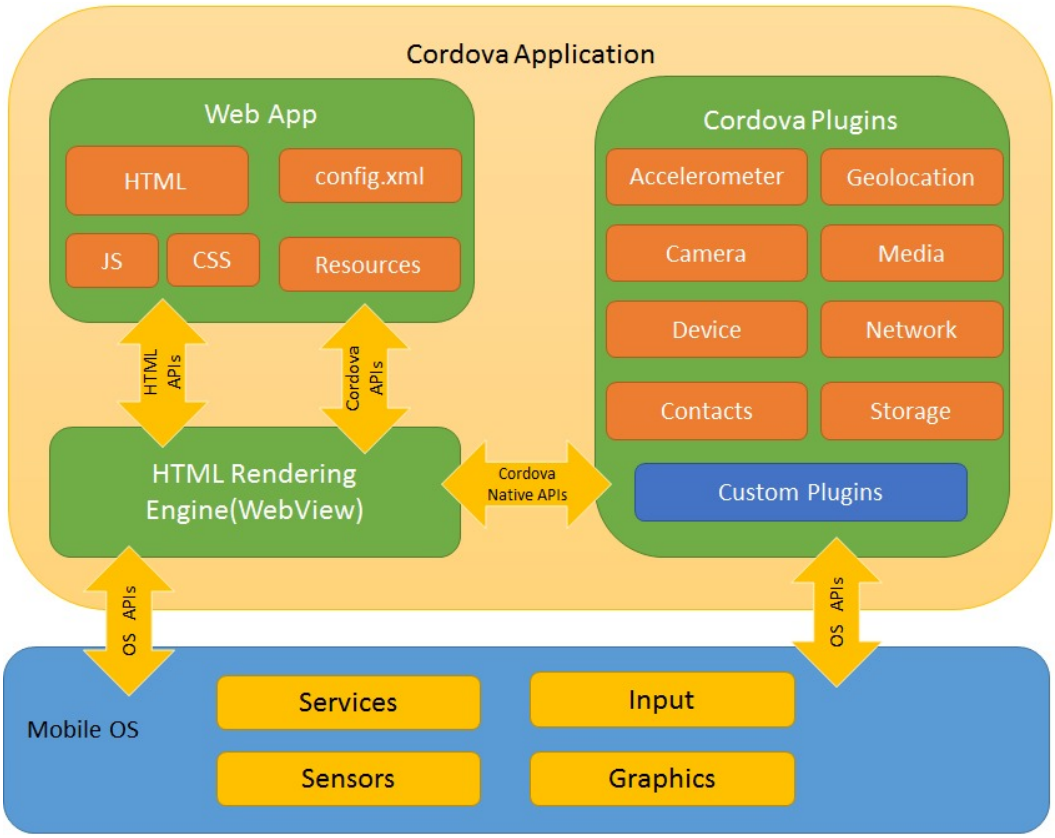


Figura 6: Architettura Cordova

[14] https://phonegap.com/
[15] https://angularjs.org/

sistemi operativi mobile di accedere, attraverso specifici plugin, alle *feature* native del dispositivo tramite il framework web (utilizzando HTML e JavaScript). Come mostrato in Fig. 6, un'applicazione Cordova si può sostanzialmente dividere in 3 parti:

- Web App
- Plugin
- WebView

Web App: In questa parte risiede il codice dell'applicazione. L'applicazione stessa è implementata come una pagina web e quindi include file HTML, CSS, JavaScript, immagini, file multimediali o qualsiasi altra risorsa necessaria al suo corretto funzionamento. L'applicazione viene eseguita all'interno di una WebView (Sezione 3) nel lato nativo dell'applicazione. Uno dei file cruciali è sicuramente il file **config.xml** [16]. Si tratta di un file di configurazione globale che controlla diversi aspetti del comportamento dell'applicazione e al suo interno si possono trovare alcune informazioni riguardanti l'applicazione, fra cui i plugin importati dalla stessa e i domini in *whitelist* (Sezione 4.2).

Cordova Plugin: I plugin sono parte integrante dell'ecosistema Cordova. Essi, infatti, forniscono un'interfaccia a Cordova e ai componenti nativi per consentirne la comunicazione tra di loro. Questi plugin fanno uso dell'API *addJavascriptInterface* illustrata nella Sezione 3.1, quindi rendono possibile l'invocazione di codice nativo da parte di JavaScript. Apachemolto frequenti. Cordova mantiene un insieme di plugin chiamati **Core Plugin**, i quali forniscono all'applicazione accesso alle funzionalità del device come batteria, camera, contatti, file system, etc [17]. In aggiunta a questi plugin, ce ne sono diversi di terze parti che forniscono *feature* aggiuntive non necessariamente disponibili su tutte le piattaforme. Questi sono sviluppati da programmatori indipendenti, come ad esempio cordovarduino o QR Code Scanner [18].

WebView: È il componente che fornisce all'applicazione l'intera interfaccia utente. Nelle applicazione ibride viene utilizzata una sua specializzazione chiamata CordovaWebView.

[16] `https://cordova.apache.org/docs/en/latest/config_ref/index.html`

[17] `https://cordova.apache.org/docs/en/latest/guide/support/index.html#core-plugin-apis`

[18] `https://cordova.apache.org/plugins/?platforms=cordova-android`

4.2 Sicurezza

Le applicazioni ibride da un lato sono applicazioni HTML5, e come tali condividono tutte le loro caratteristiche (es. JavaScript caricato a runtime) e i loro rischi di sicurezza (es. XSS), dall'altro lato condividono le caratteristiche (es. accesso completo al device) e i rischi di sicurezza (privacy leak, SQL injection [19]) delle applicazioni native. È intuibile quindi che tali applicazioni portano con loro diverse sfide dal punto di vista della sicurezza.

Il problema principale nasce dalla possibilità di poter accedere a tutte le *feature* del device direttamente da JavScript. Per poter aver accesso completo al device le applicazioni native fanno uso dei plugin descritti in Sezione 4.1. Questi sfruttano il "bridge" creato dall'API nativa *addJavascriptInterface* del componente WebView per permettere la comunicazione tra la parte JavaScript e Java, come si può osservare in Fig. 7.

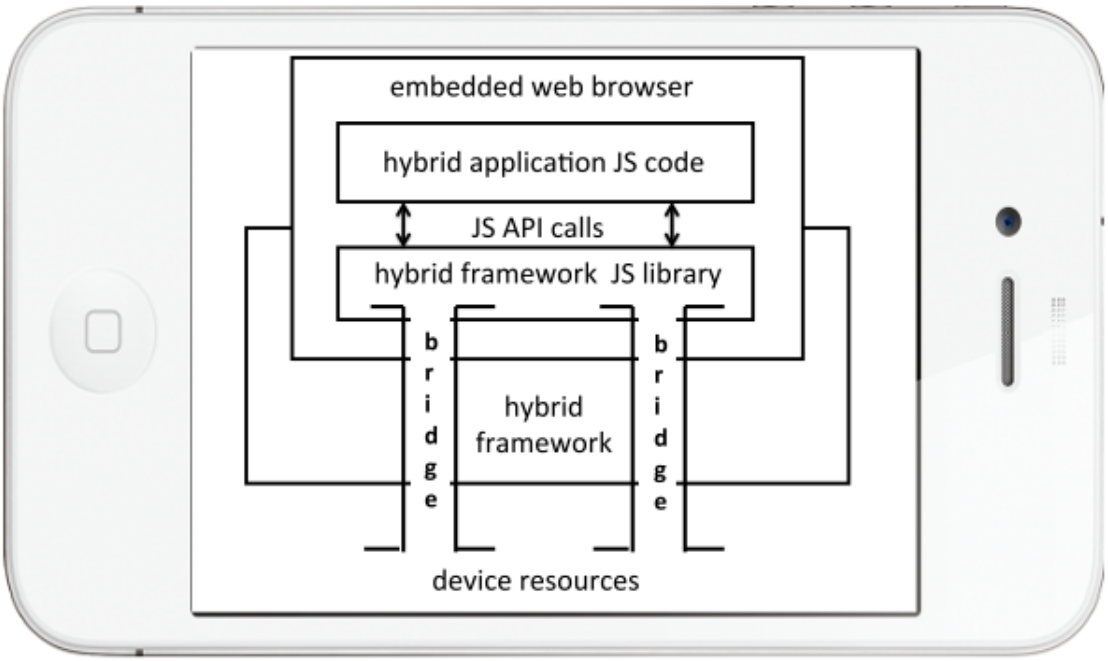


Figura 7: Componenti di un framework ibrido

Come accennato nella Sezione 3.2, tutte le pagine caricate nella WebView hanno libero accesso al "bridge". Per evitare che pagine appartenenti a domini non fidati vengano

[19] `https://www.owasp.org/index.php/SQL_Injection`

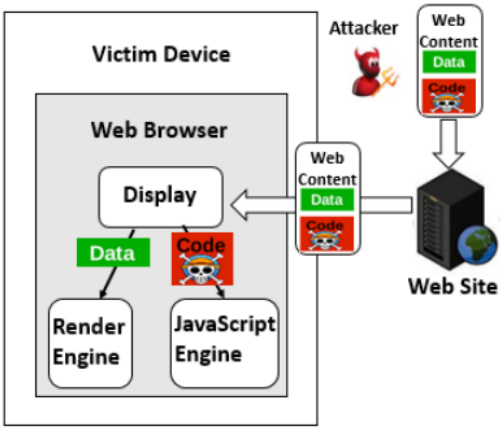
caricate all'interno dell'applicazione, Cordova implementa un meccanismo di sicurezza chiamato **Whitelist** [20]. Questo meccanismo permette a pagine appartenenti a domini "white" (cioè inseriti nella lista) di essere caricate senza problemi, bloccando invece l'accesso a quelle appartenenti a domini non presenti. Di default questo meccanismo è abilitato, ma nella lista è inserito solamente il carattere wildcard "*". Questo carattere permette a tutte le pagine di poter essere caricate senza nessun tipo di restrizioni.

Tra le problematiche più interessanti c'è sicuramente la presenza del XSS, vulnerabilità ben nota nel mondo web ma non presente nel mondo mobile. In questo contesto questa vulnerabilità ha però una superficie d'attacco molto più ampia rispetto al mondo web dove l'unico canale utilizzabile è il "sito". Come si può osservare in Fig. 8,le possibili vie di attacco per le applicazioni ibride sono innumerevoli.

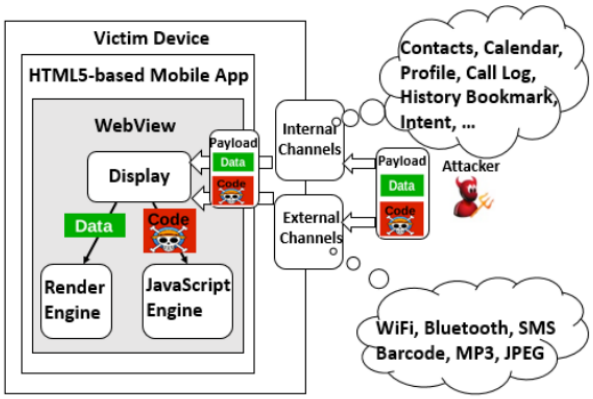
Oltre ad avere una superficie d'attacco più ampia, questa vulnerabilità diventa anche più pericolosa: potendo utilizzare tutte le feature del device ed avendo accesso (permessi permettendo) ai dati personali dell'utente potrebbe essere in grado di mandare SMS, accedere al GPS o ai contatti [4, 5, 6].

Infine, ma non per importanza, c'è il problema legato all'utilizzo di *iframe* che porta alla vulnerabilità della **Frame Confusion** approfondito nel capitolo 5. Tale vulnerabilità che colpisce solo il mondo Android può avere un impatto davvero devastante sul device e sull'applicazione stessa motivo che ha spinto questo lavoro di tesi.

[20] `https://cordova.apache.org/docs/en/latest/guide/appdev/whitelist/index.html`



(a) Attacchi XSS nelle applicazioni web



(b) Attacchi XSS nelle applicazioni ibride

Figura 8: XSS superfici di attacco

5 Frame Confusion

Nel sistema Android le interazioni tra i diversi componenti del sistema sono asincrone per evitare che l'applicazione si blocchi durante l'esecuzione di un task molto lungo e per offrire una migliore *User Experience*. Tali comunicazioni hanno, quindi, la necessità di avere un meccanismo di callback per informare l'iniziatore quando il task sarà completato. Quando si fanno chiamate a codice nativo JAVA da JAVASCRIPT (all'interno della WebView) quest'ultimo non si bloccherà in attesa del risultato, bensì quando la risposta sarà pronta, il codice JAVA all'interno dell'applicazione invocherà una funzione JAVASCRIPT passandole il risultato.

```
Object obj = new Object(){
    @JavascriptInterface
    public void showDomain(){
        mWebView.loadUrl("javascript:alert(document.domain)");
    }
};
mWebView.addJavascriptInterface(obj,"demo");
```

Codice Sorgente 1: Esempio Frame Confusion

Il problema della **Frame Confusion** nasce quando la pagina web contiene dei frame [21]. Un iframe (*Inline frame*) è un documento HTML inserito all'interno di un altro su un sito web. L' elemento iframe viene spesso utilizzato per inserire contenuti provenienti da un'altra fonte, come *advertisemnt*, all'interno della pagina Web. La pagina principale, cioè quella che inserisce al suo interno l'elemento iframe viene chiamata *main frame*, mentre il documento importato all'interno di essa viene definito *child frame*.

Come già accennato in Sezione 3.2, tutte le pagine caricate nella WebView hanno libero accesso al "bridge'; questo però è vero anche per pagine con più frame integrati dove anche i *child frame* possono accedervi. Quando l'interazione parte dal child frame la callback viene eseguita comunque nel contesto del main frame della pagina.

In figura Fig. 9 vengono rappresentati il main e il child frame all'interno del componente WebView.

Nell'esempio in Codice Sorgente 1 viene registrato un oggetto JAVA alla WebView come un'interfaccia di nome "demo" e all'interno di quell'oggetto, viene definito il meto-

[21] `https://www.w3.org/TR/2011/WD-html5-20110525/the-iframe-element.html`

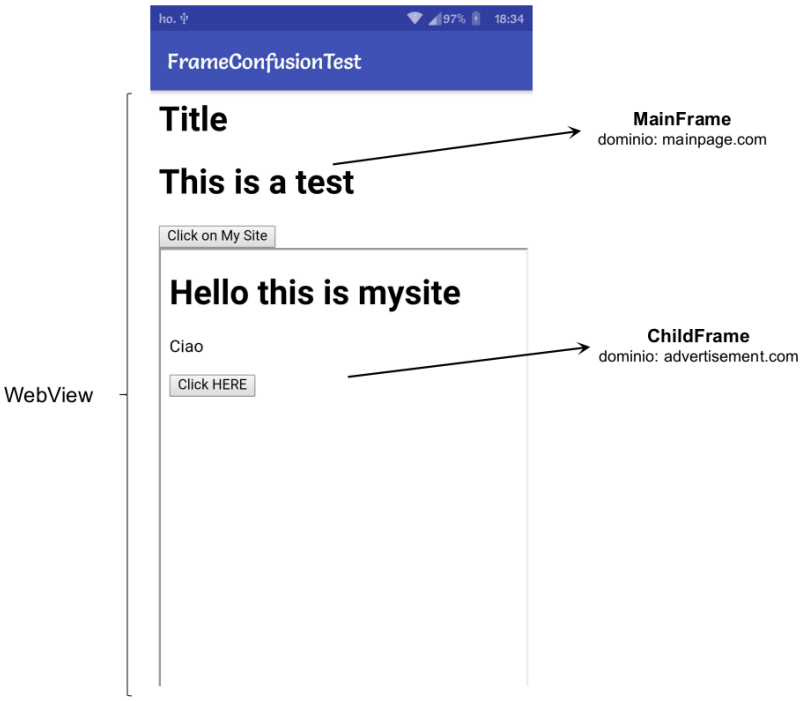


Figura 9: Main e Child frame

do `showDomain`. Usando il metodo `loadUrl` viene chiamata una funzione JavaScript per stampare a schermo il nome del dominio della pagina.

Quando viene effettuata la chiamata `demo.showDomain()` dal child frame, il pop-up che viene aperto mostra però il nome del dominio riferito al main frame e non del child frame come ci si potrebbe aspettare (come in Fig. 10).

In Fig. 10 vi è un esempio di funzionamento del codice sorgente in Codice Sorgente 1. Dopo aver cliccato uno dei due bottoni, un funzione JavaScript richiama il metodo `showDomain` attraverso l'interfaccia. In entrambi i casi, però, il dominio che viene mostrato è quello del main frame.

Il fatto che venga mostrato il nome del dominio del main frame è un'indicazione che il codice JavaScript all'interno del metodo `loadUrl` viene eseguito nel contesto del main frame. Come risultato, la combinazione di *addJavascriptInterface* e *loadUrl* (o nelle versioni più recenti di Android *evaluateJavascript*) crea un canale di comunicazione tra il child frame e il main frame. Un eventuale comportamento corretto del codice in Codice Sorgente 1 è visibile in Fig. 11, dove il l'interfaccia esegue la callback nel giusto dominio.

La creazione di questo canale e quindi la possibilità data a due pagine provenienti da

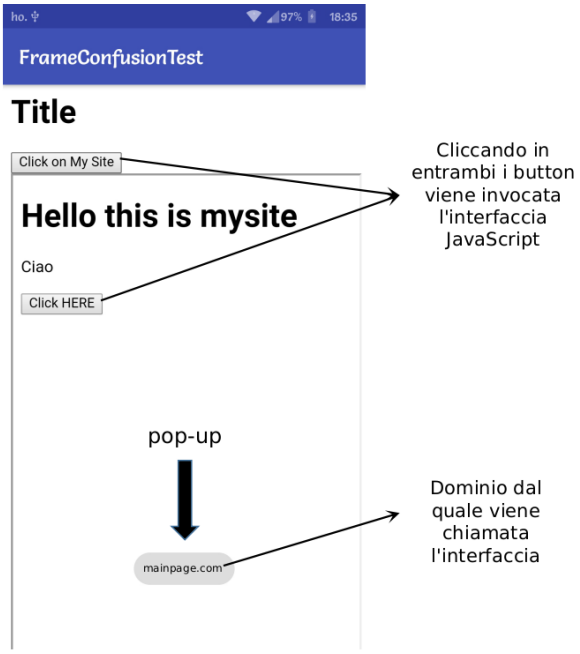


Figura 10: Esempio esecuzione Codice Sorgente 1

domini diversi di comunicare tra di loro può portare a vulnerabilità non presente negli attuali browser (protetti dal meccanismo del Same-Origin Policy SOP). La possibilità che un pagina collocata in un dominio possa invocare codice di un'altra pagina situata in un differente dominio è un grosso problema. Per esempio sarebbe possibile rubare eventuali credenziali d'accesso, cookie di sessione, modificare il DOM della pagina semplicemente creando una pagina personale sul proprio sito web.

A fronte di queste considerazioni, esistono principalmente due scenari di attacco: 1) Attacco dal Child Frame e 2) Attacco dal Main Frame.

5.1 Attacco dal Child Frame

In questo tipo di attacco la web page malevola è contenuta nel child frame. Questa situazione è molto frequente per gli *advertisement* (annunci pubblicitari) che vengono incorporati nelle applicazioni e nei siti web proprio grazie agli iframe. Il principale obiettivo di un eventuale utente malevolo è quello di iniettare codice nel main frame per comprometterne l'integrità.

Nei classici browser questo non può accadere essendo applicata la politica SOP. Essa infatti isola completamente il contenuto del main frame dal child frame se provenienti

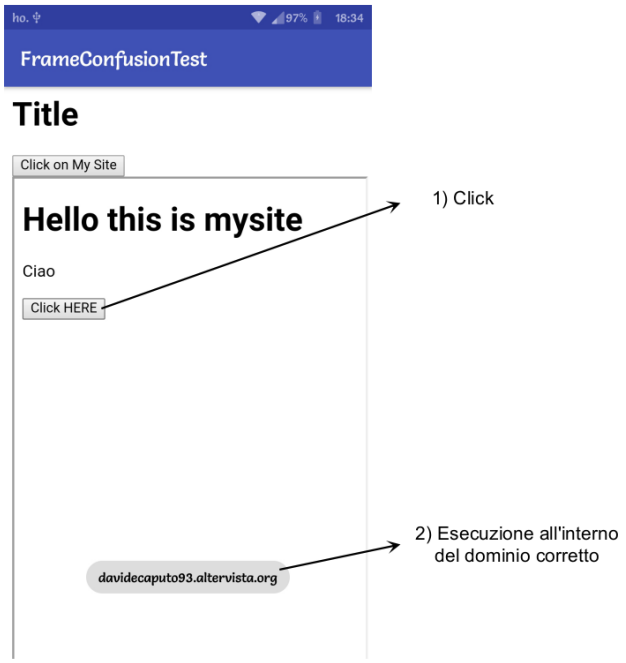


Figura 11: Corretto funzionamento del Codice Sorgente 1

da diversa "origine" (differente dominio, diverso protocollo e porta TCP). Per esempio, il codice JavaScript all'interno di un child frame non può accedere al DOM o ai cookies del main frame. Pertanto, se il contenuto all'interno degli iframe è malevolo, non può e non dovrebbe essere in grado di compromettere la pagina nel main frame.

Per capirne effettivamente la pericolosità si considera come esempio un'applicazione che ha registrato un'interfaccia nella sua WebView chiamata `CameraLauncher`. In questa classe viene registrato un metodo chiamato `fail` utilizzato da Java per mandare messaggi di errore alla pagina web, come è possibile vedere in Codice Sorgente 2.

```
public class CameraLauncher{
    @JavascriptInterface
    public void fail(String paramString){
        String str = "javascript:navigator.camera.fail('";
        str += paramString + "');";
        this.mAppView.loadUrl(str);
    }
}
```

Codice Sorgente 2: Esempio Attacco dal Child Frame

Il metodo `fail()` però è invocabile anche dal codice JavaScript all'interno della WebView sia dalla main che dai child frame. In altre parole, il codice JavaScript all'interno del child frame può utilizzare questa interfaccia per mostrare messaggi di errore nel main frame, aprendo così un canale tra la pagine web principale e il suo iframe. Nell'esempio in Codice Sorgente 2, il parametro `paramString` è vulnerabile non essendo sanitizzato. E' possibile, infatti, inserire codice JavaScript arbitrario che verrà eseguito nel main frame semplicemente passando una stringa come quella in Codice Sorgente 3.

```
    x'); document.location = "http://evilsite.com/cookiestealer.php?c="
        + document.cookie; //
```

Codice Sorgente 3: Stringa Javascript Inject

Il codice malevolo inserito all'interno della variabile `paramString` viene eseguito nel main frame ed è in grado di modificare il DOM della pagina o mandare richieste ad un server esterno malevolo. Nel caso di applicazioni ibride, tale codice potrebbe richiamare funzionalità del OS originale come ottenere contatti, SMS, file personali e mandarli ad un end point controllato dall'attaccante. Risulta essere, pertanto, un esempio analogo al XSS sul browser web. In questo caso, però, il codice malevolo viene iniettato attraverso iframe provenienti da domini diversi rispetto alla pagina principale (vedi Fig. 12).

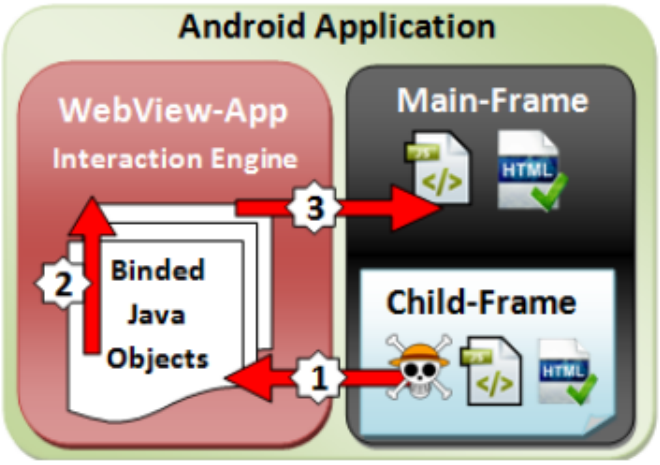


Figura 12: Attacco dal Child Frame

5.2 Attacco dal Main Frame

In questo tipo di attacco l'attaccante deve riuscire a caricare la sua pagina come main frame all'interno della WebView. Ad esempio, un'applicazione che utilizza il protocollo HTTP per caricare le pagine all'interno della WebView può essere alterata per esempio utilizzando un attacco MiTM ([22]).

La pagina originale viene quindi alterata con una sotto il controllo dell'attaccante dove all'interno di essa vi è inserito un child frame contente la pagina legittima dell'applicazione.

Se l'applicazione avesse implementato un meccanismo di sicurezza per prevenire l'uso dell'interfaccia da codice JavaScript non autorizzato come illustrato in Sezione 5.1, in caso di attacco dal main frame sarebbe comunque inutile. Questo meccanismo di sicurezza può essere implementato utilizzando dei "token", che se validi, permettono di utilizzare l'interfaccia.

[22] `https://www.owasp.org/index.php/Man-in-the-middle_attack`

```
public class Storage{
    @JavascriptInterface
    public void QueryDatabase(SQLStat query, Token token){
        if(!this.checkToken(token))
            return;
        else {
            /* Do the database query task and return result*/
        }
    }
}
```

Codice Sorgente 4: Esempio utilizzo token

Servendosi del "token" (Codice Sorgente 4) solo la pagina autorizzata può accedere all'interfaccia e iniziare la comunicazione con la parte nativa. In tali condizioni, quindi, sarebbe impossibile per la pagina malevola (contenuta nel main frame) iniziare l'interazione. Tuttavia nulla vieterebbe alla pagina legittima contenuta nel child frame di avviarla. In questo caso quando i risultati saranno pronti, l'applicazione invocherà una funzione JavaScript utilizzando i metodi `loadUrl` o `evaluateJavaScript` ed a causa del problema della **Frame Confusion** il risultato della query verrà passato al main frame sotto il controllo dell'attaccante. Questo scenario crea un *information-leak channel* (Fig. 13). Questo tipo risulta quindi più difficile da attuare del precedente poi-

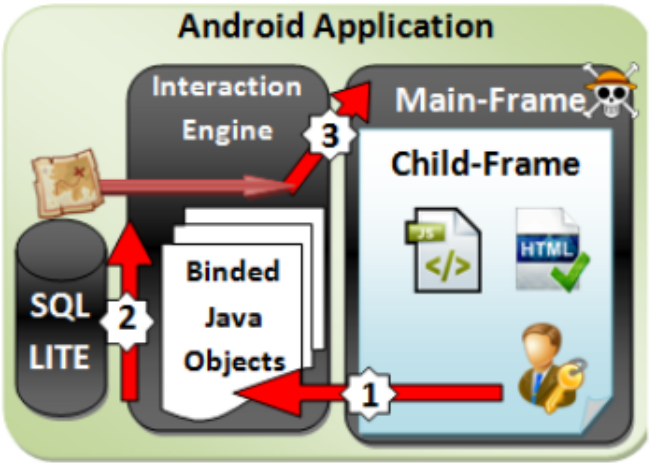


Figura 13: Attacco dal Main Frame

ché richiede che all'interno della WebView venga caricata una pagina sotto il controllo dell'attaccante.

5.3 Meccanismi di sicurezza

L'utilizzo degli iframe, come visto in questo capitolo, è una pratica comune, infatti essi vengono spesso per contenere *advertisement*. Data la loro popolarità sono stati sviluppati alcuni meccanismi di "isolamento" per evitare possibili problemi di sicurezza provocati dal loro utilizzo. I principali meccanismi di sicurezza applicati agli iframe sono:

- Attributo *sandbox*[23] del componente iframe: è stato inserito nello standard HTML5. Questo attributo è in grado di disabilitare ogni possibile interazione all'interno dell'iframe dove viene dichiarato.

- Content Security Policy (CSP), uno standard introdotto per prevenire XSS, click-jacking [24] e altri attacchi del tipo code injection proveniente da contenuti malevoli (es. iframe) all'interno di pagine web fidate [7].

Questi meccanismi di sicurezza, però, di default, sono disattivati. E' quindi compito dello sviluppatore implementarli correttamente.

[23] `https://developer.mozilla.org/it/docs/Web/HTML/Element/iframe`
[24] `https://www.owasp.org/index.php/Clickjacking`

6 Stato dell'arte

Negli ultimi anni le applicazioni ibride stanno diventando sempre più popolari, come conferma un report Gartner [8]. Attualmente nel playstore di Google si stimano 3.000.000 applicazioni e circa il 15% di esse sono ibride [9]. Questa percentuale, in continuo aumento, è dovuta al fatto che questo tipo di applicazioni riesce a sfruttare i punti forti sia delle applicazioni native che delle applicazioni web, riducendo inoltre i costi di sviluppo e diventando quindi una soluzione invitante per le aziende e per la community degli sviluppatori.

Nel corso dell'ultimo decennio la comunità scientifica si è concentrata nello sviluppo di metodologie in grado di analizzare automaticamente le applicazioni mobile per identificare eventuali vulnerabilità, errori nel codice o possibili problemi per la privacy dell'utente (*privacy leak*). La motivazione dietro questo grande impegno è il grande aumento dell'utilizzo degli smartphone (al giorno d'oggi il 68% delle persone al mondo ne ha uno [25]).

Le attuali metodologie di analisi automatica su applicazioni mobile si dividono in tecniche di analisi statica e dinamica. Nel primo caso l'applicazione non viene eseguita, ma viene analizzato il suo codice sorgente ottenuto tramite tecniche di *reverse engineering*. Nelle tecniche di analisi dinamica, invece, viene analizzato il comportamento dell'applicazione a runtime (mentre viene eseguita in un ambiente di test controllato) e si analizza il traffico di rete, le interazioni con i file e con le API di sistema.

Entrambe le metodologie hanno i propri vantaggi e svantaggi. L'analisi statica ha il vantaggio di richiedere (generalmente) meno risorse rispetto a quella dinamica. Essa infatti non richiede che l'applicazione venga eseguita. Durante l'analisi viene quindi esaminato tutto il codice sorgente, anche sezioni di esso non più raggiungibili, ottenendo risultati non sempre del tutto affidabili a causa del gran numero di falsi positivi. Per quanto riguarda l'analisi dinamica, i risultati ottenuti sono corretti ma non sempre completi. La non completezza è legata al fatto che difficilmente tutti gli stati di esecuzione di un'applicazione vengono raggiunti in maniera automatica e in tempo ragionevole, ottenendo in questo caso dei falsi negativi. Un'altra problematica riguardante questa analisi è che non sempre è praticabile. Per effettuarla, infatti, vengono utilizzati diversi strumenti automatici che generano input e/o eventi che simulano le

[25] `https://www.gsmaintelligence.com/`

azioni di un utente reale (click, scroll, etc.) e framework come Xposed[26] o Frida[27] in grado di monitorare il comportamento, i quali, per poter funzionare, hanno spesso la necessità di avere i permessi *root* sul dispositivo di testing. Alcune applicazioni, inoltre, hanno meccanismi di sicurezza in grado di rilevare l'utilizzo di questi framework o di rilevare un dispositivo Android che ha effettuato il *rooting*, rendendo impraticabile l'analisi dinamica.

In questo lavoro di tesi, per ovviare alle problematiche relative ai due approcci, si è deciso di utilizzare una metodologia "ibrida", cioè basata su entrambe le tecniche. Nel lavoro di Chen e altri [10] viene dimostrato come l'utilizzo di un approccio ibrido riduca notevolmente il numero sia di falsi positivi che di falsi negativi.

6.1 Vulnerabilità WebView e Applicazioni Ibride

Nel 2012 è stata scoperta la vulnerabilità più seria per quanto riguarda il componente WebView[28] nelle versioni di Android inferiori alla 4.2 (API 17) (alla data attuale circa l'1.8 %[29]). Questo tipo di vulnerabilità, attraverso l'utilizzo dell'interfaccia *addJavascriptInterface*, permette l'esecuzione di codice Java arbitrario da remoto [1]. Codice Sorgente 5 motiva come la possibilità di invocare metodi JAVA arbitrari può causare ingenti danni all'utente (invio di un sms, manipolazione del file system, etc.). Per rimuovere questa vulnerabilità si è deciso di annotare con *@JavascriptInterface*, a partire dalla versione 4.2 di Android, i metodi invocabili dall'interfaccia, come mostrato in Codice Sorgente 2.

```
// JavaCode
fileUtilsObject = new FileUtils();
webview.addJavascriptInterface(fileUtilsObject, "FUtil");

// JavaScript code
 <script>
 FUtil.getClass()
     .forName('java.lang.Runtime')
     .getMethod('getRuntime',null).invoke(null,null).exec(['id']);
</script>
```

[26] https://repo.xposed.info/module/de.robv.android.xposed.installer
[27] https://www.frida.re/docs/home/
[28] https://www.cvedetails.com/cve/cve-2012-6636
[29] https://developer.android.com/about/dashboards/

Codice Sorgente 5: Exploit Vulnerabilità JavaScript Reflection

Nel lavoro di Neugschwandtner e collaboratori [11] vengono analizzate le possibili vulnerabilità e attacchi effettuabili riguardanti il componente WebView. Lo scopo dell'attaccante è quello di inserire codice malevolo all'interno dell'applicazione, ed eventualmente, attraverso l'interfaccia *addJavascriptInterface*, di invocare codice Java nativo. Come scenari di attacco vengono analizzati la compromissione del server e del traffico di rete. È di particolare interesse la seconda situazione, presa in considerazione anche in questo lavoro di tesi. Per compromettere il traffico di rete viene utilizzato un attacco del tipo MiTM[30]. Come è possibile notare in Fig. 14 il traffico originale viene intercettato e manomesso da un'attaccante che si inserisce tra l'utente e il server. Questo tipo di attacco risulta difficile da eseguire se l'applicazione (il client) effettua connessioni HTTPS e i certificati vengono verificati. Tuttavia, come discusso in [11] le applicazioni presentano molto spesso problemi di sicurezza nelle connessioni di rete. Tra le applicazioni analizzate dagli autori (circa 290.000) il 30% utilizza l'interfaccia *addJavascriptInterface* e il 26.47% (23.048) di esse utilizza connessioni HTTP non protette, rendendo le condizioni di un attacco MiTM molto favorevoli.

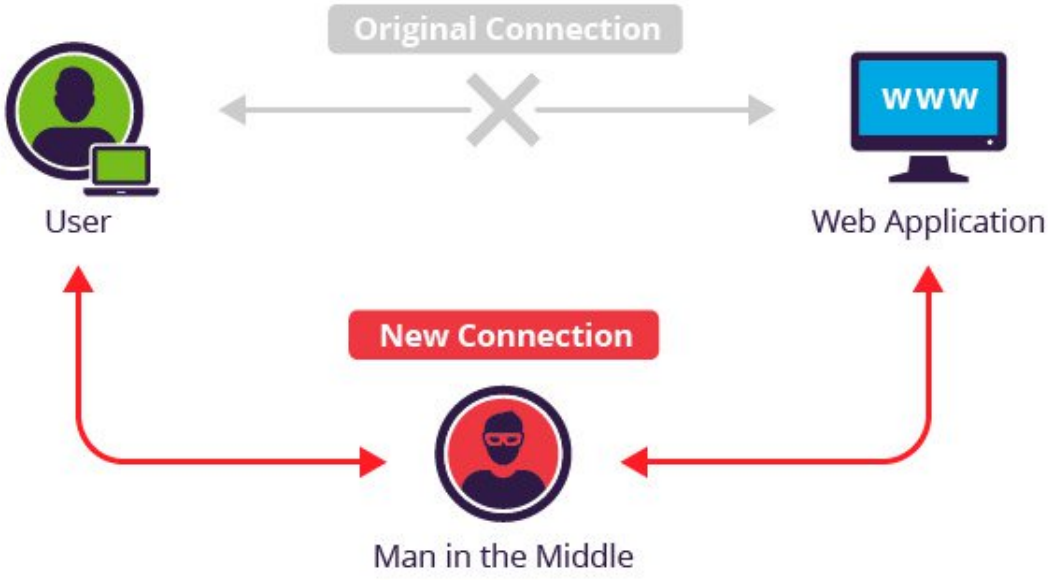


Figura 14: Attacco MiTM

[30] https://www.owasp.org/index.php/Man-in-the-middle_attack

L'attacco descritto precedentemente potrebbe risultare devastante se attraverso l'interfaccia *JavaScript* si potessero invocare API denominate "dangerous", cioè API che hanno la necessità di un permesso esplicito per essere invocate. Questa procedura, cioè la possibilità di invocare le API "dangerous" attraverso l'interfaccia, è uno dei funzionamenti chiave sul quale si basano le applicazioni ibride. Gli eventuali problemi di sicurezza sono analizzati in [12], dove gli autori mostrano l'impatto di un attacco XSS eseguito su applicazioni ibride. Come già accennato in Sezione 4.2 i canali di attacco per effettuare XSS su questo tipo di applicazioni sono numerosi, essendo lo smartphone progettato per comunicare con il mondo esterno (fotocamera, Bluetooth, WiFi etc.). Gli autori di questo lavoro hanno cercato di dimostrare come una semplice applicazione per leggere QRCode o per fare una scansione Bluetooth, se implementata male, può essere sfruttata per rubare o eliminare contatti e file.

Nel corso degli ultimi anni diverse vulnerabilità per il componente WebView e le applicazioni ibride sono state scoperte e documentate. Alcune di queste sono state risolte nel corso delle diverse versioni di Android; per alcune invece (come la **Frame Confusion**) non vi è ancora una soluzione condivisa.

6.2 Tecniche di analisi per applicazioni Android

La letteratura scientifica che tratta metodologie e strumenti per effettuare analisi di applicazioni Android è molto estesa come documentato nei lavori di Bagheri e di Li [13, 14].

L'analisi statica esamina la struttura del programma per identificarne potenziali comportamenti. A tal fine l'analisi statica può utilizzare tecniche del tipo *text mining* che vengono chiamate "leggere" o tecniche più pesanti, ma più accurate, che fanno uso invece di strutture dati per fornire un'astrazione del codice. Le strutture dati più frequenti utilizzate sono:

- Control Flow Graph (CFG) è un grafo diretto dove ogni nodo rappresenta un'istruzione e i flussi di controllo sono gli archi del grafo.
- Call Graph (CG) anche esso è un grafo diretto, dove ogni nodo rappresenta un metodo, e un arco indica la chiamata di (o il ritorno da) un metodo.

- Inter-procedural Control Flow Graph (ICFG) è una combinazione fra i due precedenti dove diversi grafi CFG sono collegati tra di loro utilizzando archi che identificano la chiamata di (o ritorno da) un metodo.

A seconda della struttura dati utilizzata si possono effettuare analisi di diverso tipo, come la Control Flow Analysis, che si basa sul CFG, la quale serve per verificare la correttezza del flusso di controllo alla ricerca di eventuali anomalie come la presenza di codice irraggiungibile. Un'altra tecnica molto interessante che si basa sempre sul CFG è la Data Flow Analysis: questo tipo di analisi serve per analizzare l'evoluzione delle variabili. Una delle tecniche più interessanti che fa uso della Data Flow Analysis è sicuramente la Taint Analysis. In questo tipo di analisi gli oggetti vengono "contaminati" (tainted) e si tiene traccia del loro flusso. Se il flusso di un oggetto contaminato finisce in un punto dove non dovrebbe, cioè un sink, viene generato un allarme. Questo tipo di analisi viene utilizzata sopratutto per rilevare eventuali *privacy leak*.

L'analisi dinamica, come già accennato, ha l'obiettivo di eseguire l'applicazione e monitorarne il comportamento a runtime.

Per effettuare questo tipo di analisi è necessario eseguire l'applicazione ed è quindi fondamentale un sistema accurato di stimolazione dell'applicazione. Tuttavia, a differenza dei classici test funzionali che mirano a dimostrare che un sistema software sia conforme alle specifiche e che esso si comporti correttamente a seguito di un insieme di input predisposto a priori. I test di sicurezza per sopperire a questa mancanza, utilizzano una tecnica di tipo *fuzzing*, tale tecnica esegue l'applicazione con input "random" cercando di esplorare tutti gli stati di esecuzione della stessa.

Durante l'esecuzione dell'applicazione è necessario monitorarne il comportamento. Per fare ciò esistono tecniche di analisi che operano a diversi livelli:

- *App-Level*, a questo livello viene tracciato l'invocazione del metodo JAVA originale nel *bytecode* attraverso un inserimento di un'istruzione di log all'interno del codice originale dell'applicazione o nel framework Android.

- *Kernel-Level*, le tecniche di monitoraggio a questo livello raccolgono le *call system*, usando moduli kernel e strumenti come `strace` o `ltrace`.

- *Virtual Machine (VM)-level*, a questo livello si utilizzano strumenti che intercettano gli eventi che avvengono all'interno dell'emulatore.

6.3 Strumenti di analisi per applicazioni Ibride

Le metodologie disponibili per le applicazioni ibride, a differenza di quelle disponibili per le applicazioni native, sono in un numero ristretto e preservano diversi limiti. Le applicazioni ibride, così come le applicazioni che utilizzano il componente WebView , utilizzano due linguaggi di programmazione (tipicamente Java e JavaScript). L'utilizzo di due linguaggi con semantiche diverse come Java (statico e tipizzato) e JavaScript (estremamente dinamico e non tipizzato) rende lo sviluppo di una metodologia di analisi automatica molto complessa.

HybriDroid [15] è un framework per effettuare analisi statica su applicazioni ibride. Gli autori hanno deciso di focalizzarsi sull'inter-comunicazione tra i due linguaggi, analizzandone le differenze semantiche. Il framework è in grado di identificare i bug (provenienti dalla mancata corrispondenza fra due tipi di dato appartenenti ai due linguaggi diversi) ed è inoltre in grado di identificare se vi sono perdite di informazioni sensibili (*information leakage)*. Uno dei componenti chiave di questo framework è chiamato *StringAnalyzer*. Questo componente effettua principalmente una ricerca all'interno dell'applicazione per trovare gli URL. Essendo però gli URL ricercati staticamente, il risultato finale risulta incompleto (alcuni URL vengono creati dinamicamente) o comprendere URL mai utilizzati e quindi falsi positivi.

Rizzo et al. [4] ha proposto BabelView, uno strumento per effettuare information flow analysis in grado di valutare l'impatto di *code injection attack* che potrebbero manipolare o esporre dati sensibili dell'utente. L'idea è quella di evitare di analizzare diversi linguaggi di programmazione sostituendo l'eventuale codice malevolo JavaScript sconosciuto dell'attaccante. Questo codice viene sostituito con un possibile comportamento malevolo di un malintenzionato che approssimi il meglio possibile l'information flow di un eventuale attacco. Per raggiungere questo obiettivo gli autori hanno strumentato l'applicazione sostituendo tutte le WebView con una loro classe *BabelView* che simula l'interazione con l'interfaccia JavaScript riproducendo ogni possibile comportamento malevolo. Dopodiché viene effettuata una taint analysis per rilevare eventuali information flow in grado di leggere o scrivere informazioni sensibili come risultato di un attacco di injection. Questa proposta ha però delle limitazioni: innanzitutto, essendo basata su una flow analysis, è soggetta a tutte le limitazioni già esposte che riguardano gli approcci statici. Inoltre non viene fatta nessun tipo di dif-

ferenza tra istanze diverse di WebView. Ad ogni WebView possono essere attaccate svariate interfacce e ognuna di esse avrà i propri metodi disponibili. In questo lavoro si è fatta la semplificazione di rendere tutti i metodi disponibili ad ogni istanza di WebView ottenendo come risultato un gran numero di falsi positivi.

In [16] viene descritto Spartan Jester. L'obiettivo del progetto è quello di proporre una metodologia per il controllo dell'information flow (IFC) delle applicazioni ibride. La soluzione combina l'analisi statica utilizzando SPARTA [17] per quanto riguarda la parte Android e l'analisi dinamica, utilizzando invece JEST [18] per JavaScript. SPARTA è uno strumento di verifica in grado di controllare che l'applicazione non abbia information flow malevoli e per raggiungere tale scopo viene annotato il codice sorgente e viene verificato che solo i flow specificati avvengano a runtime. Per quanto riguarda la parte JavaScript è stato utilizzato JEST, che è in grado di monitorare l'information flow di esecuzione. Il codice sorgente viene trasformato in modo tale da essere in grado di auto monitorarsi da solo, evitando le modifiche all'engine di JavaScript. Le problematiche di questo lavoro nascono dall'utilizzo di due metodologie differenti (analisi statica e dinamica), SPARTA e JEST usano differenti approcci per annotare i source e sink e la traduzione da uno all'altro (in questo caso da SPARTA a JEST) viene fatta manualmente. Un'ulteriore problematica è data dalla presenza di alcuni canali in JEST che non sono presenti in SPARTA come i cookie o gli elementi del Document Object Model (DOM).

Nel lavoro di [19] viene proposto DroidCIA. A differenza dei lavori precedenti, in questo caso, l'obiettivo non è più analizzare l'information flow ma presentare una nuova metodologia per identificare possibili *Code Injection Attacks* sulle applicazioni HTML5 utilizzando il Call Graph dell'applicazione e avendo come input la lista di api vulnerabili. Lo strumento é tuttavia limitato alla sola analisi statica del codice web presente nelapplicazione non considerando il codice caricato dinamicamente dalle pagine web o il codice Javascript richiamato dalla parte Android.

Negli ultimi anni le tecniche di analisi automatica stanno facendo sempre più uso del Machine Learning come in [20]. In questo lavoro viene presentato un metodo basato sul Deep Learning per identificare *Code Injection Attacks* nelle applicazioni ibride. Gli autori di [20] hanno sviluppato una rete chiamata Hybrid Deep Learning Network (HDLN). La rete viene addestrata su specifiche feature estratte dall'Abstract Syntax

Tree (AST) [31] del codice JavaScript. Come nel lavoro di [19] anche in questo caso viene utilizzato un approccio statico e viene analizzata solo la parte Web dell'applicazione; inoltre è necessario un grande set di applicazioni (sia vulnerabili che non) per poter addestrare la rete in modo esaustivo, requisito necessario facendo uso del deep learning.

In tutti i lavori analizzati non viene mai presa in considerazione la possibilità di analizzare dinamicamente l'applicazione nel suo complesso in modo da coprire anche i casi di codice scaricato dinamicamente dall'applicazione; inoltre il problema della **Frame Confusion** non viene mai preso in considerazione. Lo scopo di questo lavoro di tesi è sviluppare una metodologia ibrida, che unisca gli elementi favorevoli dell'analisi statica e dell'analisi dinamica, in grado di identificare il problema della Frame Confusion.

[31] È un albero che rappresenta la struttura sintattica astratta del codice

7 Metodologia

La metodologia sviluppata in questo lavoro di tesi per l'individuazione della **Frame Confusion** si basa su un approccio "ibrido". Una metodologia viene definita tale se utilizza tecniche sia di analisi statica che dinamica e basa i propri risultati su entrambe. Utilizzando questo approccio si ottengono diversi vantaggi non ottenibili con metodologie che traggono conclusioni solamente da una di esse.

L'utilizzo della sola analisi statica, come già descritto in Sezione 6, ha diversi limiti. Uno dei limiti principali è sicuramente il fatto che viene analizzato tutto il codice che compone l'applicazione, anche quello che effettivamente non verrà mai eseguito a runtime; questo comporta come risultato un numero elevato di falsi positivi. Tra le altre problematiche vi è inoltre l'impossibilità di prevedere gli URL creati dinamicamente e caricati all'interno del componente WebView, un punto fondamentale per l'individuazione della Frame Confusion.

Utilizzare invece la sola analisi dinamica porterebbe ad avere dei risultati finali incompleti. Il limite principale di questa analisi è, infatti, che l'applicazione deve essere stimolata automaticamente e molto spesso in modalità *black box*. Questo significa che non si conoscono a priori tutti i suoi stati e quindi risulta complicata un'esplorazione esaustiva dell'applicazione. Una delle conseguenze che ciò comporta è il grande numero di falsi negativi che vengono generati.

La metodologia ibrida utilizzata in questo lavoro di tesi permetterà di superare la limitazioni delle due tecniche, aumentando la precisione dell'analisi come dimostrato dai risultati sperimentali che saranno descritti in Sezione 9.

In questo capitolo verranno discussi inizialmente le condizioni necessarie alla presenza della vulnerabilità Frame Confusion. Nella seconda sezione verrà esposto in maniera dettagliata l'algoritmo sviluppato per l'individuazione della stessa. Infine verrà presentata l'architettura di FCDroid e verranno analizzati con maggiore dettaglio i moduli che la compongono.

7.1 Individuare la Frame Confusion

Al fine di individuare la metodologia per identificare la **Frame Confusion** nelle applicazioni mobili è stato necessario definire le condizioni sotto le quali tale vulnerabilità

possa verificarsi. Affinché possa verificarsi, la **Frame Confusion** richiede all'interno dell'applicazione:

1) L'utilizzo del componente WebView e il permesso di utilizzare Internet.

2) JavaScript abilitato.

3) L'utilizzo dell'interfaccia *addJavascriptInterface*.

4) L'utilizzo di *iframe* nelle pagine web caricate.

Le prime tre sono condizioni obbligatorie; nel caso una di esse non fosse verificata non sarebbe possibile sfruttare questa vulnerabilità. L'ultima condizione ovvero l'utilizzo del componente *iframe*, è necessaria solamente nel caso si voglia effettuare un attacco proveniente dal *child frame* (Sezione 5.1).

Però, oltre alle condizione necessarie, ci sono delle condizioni che possono favorire la presenza o la facilità di sfruttamento della Frame Confusion. Ad esempio la mancata configurazione di un protocollo sicuro di comunicazione da parte dell'applicazione, ovvero l'uso dell'HTTP o dell'HTTPS con una gestione non corretta dei certificati, comporterebbe una maggiore possibilità di sfruttamento della vulnerabilità. Infatti, un eventuale attaccante, nelle condizioni sopra elencate, sarebbe in grado di portare a termine un attacco del tipo MiTM e quindi successivamente di inserire codice malevolo JavaScript all'interno della pagina legittima richiesta dall'applicazione o di un *advertisement* (annuncio pubblicitario) in grado di accedere all'interfaccia JavaScript. Come già descritto in Sezione 5.1, è pratica comune inserire annunci pubblicitari all'interno di iframe e importarli successivamente all'interno di applicazioni o siti web.

Un'ultima considerazione da fare è relativa ad un'applicazione che ha correttamente configurato il traffico di rete ma che potrebbe contenere *advertisement* esterni; un attaccante potrebbe compromettere una campagna pubblicitaria esistente o crearne una ad-hoc con lo scopo di essere inserita nelle applicazioni benigne. Gli *advertisement* malevoli a questo punto potranno accedere allinterfaccia *JavaScript* ed ai relativi metodi Java sfruttando la Frame Confusion.

7.1.1 Controllo dei meccanismi di sicurezza

Come già discusso in Sezione 5 e in Sezione 3.2 esistono meccanismi di sicurezza per isolare gli iframe (attributo *sandbox*) e per prevenire l'esecuzione di codice JavaScript (metatag CSP). Inoltre se l'applicazione è ibrida vi è un meccanismo aggiuntivo di *Whitelist* discusso in Sezione 4.2 che abilita il caricamento dei domini all'interno dell'applicazione solo se presenti al suo interno.
Tuttavia tali meccanismi non sono abilitati di default all'interno delle applicazioni ed è compito dello sviluppatore configurarli in maniera esplicita. Non essendo quindi abilitati di default, la metodologia sviluppata e presentata in questo lavoro di tesi, controlla la loro corretta implementazione.

La metodologia sviluppata effettua inoltre ulteriori controlli di sicurezza che potrebbero ampliare la superficie d'attacco o l'impatto di esso. I controlli ulteriori che vengono attuati sono:

1) L'individuazione, all'interno dei file JavaScript, di metodi vulnerabili al XSS;

2) L'utilizzo di librerie JavaScript con vulnerabilità note;

3) La corretta implementazione delle connessioni di rete e del traffico generato dall'applicazione, che come descritto in precedenza, potrebe diventare un canale utilizzabile da un eventuale attaccante.

7.2 Approccio Ibrido: Frame Confusion Detector

L'Algoritmo 1 presenta la codifica in pseudo codice dell'algoritmo utilizzato per identificare la **Frame Confusion** e successivamente implementato nel modulo chiamato **Frame Confusion Detector**. Questo modulo prende in input ogni applicazione e verifica che le condizioni discusse in Sezione 7.1.

Inizialmente vengono identificati tutti i permessi richiesti dall'applicazione (riga 1), se tra i permessi richiesti non vi è il permesso Internet allora l'applicazione non può utilizzare il componente WebView e non risulta vulnerabile (riga 2-3). In seguito vengono individuati tutti i metodi invocati (riga 4). Tale lista di metodi viene generata dinamicamente. Qualora non fosse stato possibile analizzare dinamicamente l'applicazione la lista verrebbe comunque in maniera statica a partire dal codice dell'applicazione.

Algorithm 1: Detecting Frame Confusion

Input : Applicazione da analizzare
Output: Vulnerabile Frame Confusion

```
1  listPermissions = getPermissionFromApk(applicazione);
2  if "android.permission.INTERNET" not in listPermissions then
3  |  return False;
4  end
5  listaMetodiInside = getAllMethodsInvoked(applicazione);
6  javaScriptEnabled = False; javaScriptInterface = False;
7  foreach metodo in listaMetodiInside do
8  |  if metodo.getName == "setJavaScriptEnabled" then
9  |  |  listaParametri = getParametersFromMethod(metodo);
10 |  |  if listaParametri contains True then
11 |  |  |  javaScriptEnabled = True;
12 |  |  end
13 |  end
14 |  else if metodo.getName == "addJavascriptInterface" then
15 |  |  javaScriptInterface = True;
16 |  end
17 end
18 if not javaScriptInterface or not javaScriptEnabled then
19 |  vulnerable = False;
20 |  return vulnerable;
21 end
22 fileResources = getAllResourceApk(applicazione);
23 fileURL = getAllUrlRequest(applicazione);
24 fileUrlStatic = getUrlFromMethodLoadUrl(listaMetodiInside);
25 fileToCheck = fileURL union fileResources union fileUrlStatic;
26 foreach file in fileToCheck do
27 |  if isHTMLfile(file) then
28 |  |  if isContainIframe(file) and not fileContainCSP(file) then
29 |  |  |  if not containsSandboxAttribute(file) then
30 |  |  |  |  fileHTMLwithIframe.append (file);
31 |  |  |  end
32 |  |  end
33 |  end
34 |  else if isJsFile(file) then
35 |  |  if usedFunctionToCreateIframe(file) then
36 |  |  |  fileJSwithIframe.append (file);
37 |  |  end
38 |  end
39 end
40 fileUseLibraryVulnerable = getFileWithVulnerableLibrary (fileToCheck);
41 fileVulnerableXSS = getFileVulnerableXSS (fileToCheck);
42 httpRequest = getAllHttpRequest (applicazione);
43 if isHybridApplication(file) then
44 |  whiteList = getWhiteList (applicazione);
45 |  saveResult (whiteList);
46 end
47 saveResult (fileUseLibraryVulnerable);
48 saveResult (fileVulnerableXSS);
49 saveResult (httpRequest);
50 saveResult (listPermissions);
51 if len (fileHTMLwithIframe) > 0 then
52 |  saveResult (fileHTMLwithIframe);
53 |  vulnerable = True;
54 end
55 if len (fileJSwithIframe) > 0 then
56 |  saveResult (fileJSwithIframe);
57 |  maybeVulnerable = True;
58 end
59 return vulnerable,maybeVulnerable;
```

Dopodiché vengono analizzate le condizioni necessarie alla presenza della Frame Confusion. In particolare, per il metodo *setJavaScriptEnabled* si controlla che la sua invocazione avvenga con il parametro `true`, condizione necessaria per attivare l'esecuzione del codice JavaScript all'interno della WebView (di default è disattivata) (riga 7-10). Per il metodo *addJavascriptInterface*, invece, si verifica che la sua invocazione avvenga da parte dell'applicazione. L'esecuzione di tale metodo è necessaria per creare il *bridge* che permette di invocare codice nativo Java da JavaScript (riga 11-12). Qualora le condizioni appartenenti ad entrambi i metodi siano verificate l'analisi prosegue (riga 13-15).

In seguito vengono ricavati tutti i file di risorse contenuti all'interno dell'applicazione e tutti gli URL appartenenti alle pagine richieste dall'applicazione durante l'analisi dinamica (riga 22-23). Vengono inoltre recuperati tutti gli URL passati come parametro al metodo *loadUrl* (riga 24). A questo punto i risultati dell'analisi statica e dinamica vengono uniti (riga 25).

I file così individuati vengono filtrati in base all tipologia (riga 27 e 34):

- Nel caso di file `html` viene controllata la presenza dell'elemento *iframe* e la mancanza del meta tag CSP (riga 28); se questa condizione è verificata viene fatto un'ulteriore controllo verificando se nell'elemento iframe compare l'attributo *sandbox* (riga 29): in caso di riscontro negativo il file `html` viene marcato come vulnerabile (riga 30).

- Nel caso di file in formato JavaScript, viene controllato se al suo interno è presente un metodo in grado di aggiungere l'elemento *iframe* al DOM (riga 35-36). I file positivi a questo riscontro vengono aggiunti alla lista di potenziali vulnerabili.

Nel passaggio successivo vengono controllati i file caricati all'interno del componente WebView alla ricerca di vulnerabilità XSS o che fanno uso di librerie vulnerabili (riga 40-41), mentre per gli URL caricati dall'applicazione vengono memorizzati quelli che utilizzano il protocollo HTTP (riga 42).

Se l'applicazione risulta essere ibrida viene controllato inoltre se implementa il meccanismo di whitelist (discusso in Sezione 4) e in tal caso viene ottenuta la lista di domini e viene memorizzata (riga 44-45). Vengono inoltre memorizzati i file che fanno uso di librerie vulnerabili, i file potenzialmente vulnerabili al XSS, le richieste effettuate sotto protocollo HTTP e file vulnerabili al Frame Confusion (se esistenti) (riga 47-50).

Infine vengono salvati, se esistenti, i file potenzialmente vulnerabili alla Frame Confusion e l'applicazione viene identificata come vulnerabile se esistono file `html` marcati come tali altrimenti potenzialmente vulnerabile se esistono solo file `js`. Tali file sono marcati come potenziali vulnerabili perché vengono identificate le funzioni che possono creare *iframe* a runtime, ma la loro effettiva esecuzione non è stata osservata.

7.3 Architettura di FCDroid

Al fine di implementare la metodologia descritta in Sezione 7.2 è stato necessario progettare un'architettura in grado di integrare i diversi moduli di analisi. In Fig. 15 è rappresentata l'architettura di riferimento di FCDroid.

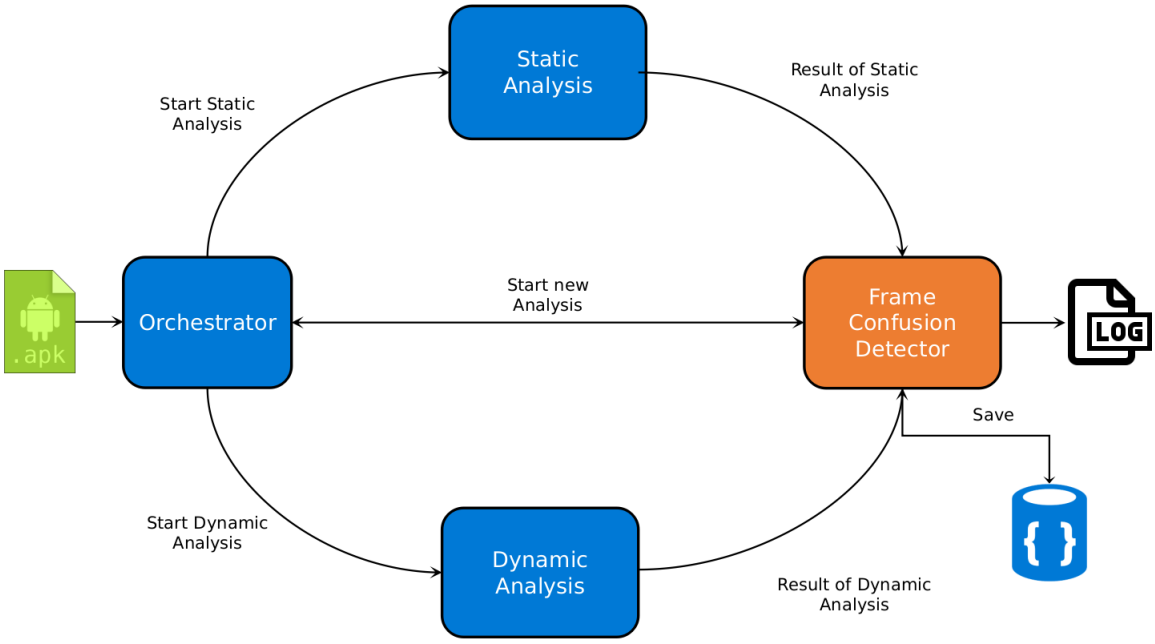


Figura 15: Panoramica FCDroid

Il modulo di analisi statica ha il compito di analizzare l'applicazione senza eseguirla e di ottenere ed esaminare i permessi dichiarati dalla stessa, le risorse che la compongono (file `html`, `xml` e `js`) e il codice sorgente ottenuto tramite tecniche di *reverse engineerning*.

Il modulo di analisi dinamica, invece, ha il compito di eseguire l'applicazione automaticamente simulando le azioni di un utente reale. Durante questa fase il modulo ha inoltre la funzione di monitorare sia il traffico di rete prodotto dalla stessa sia le API di sistema invocate.

Quando i moduli responsabili dell'analisi hanno finito di esaminare l'applicazione inviano i risultati ricavati al cervello di quest'architettura: il **Frame Confusion Detector**. Esso integra i risultati e verifica se l'applicazione è vulnerabile alla Frame Confusion utilizzando Algoritmo 1 descritto in Sezione 7.2.

Dopodiché il **Frame Confusion Detector** salva i risultati e comunica all'**Orchestrator** la possibilità di iniziare una nuova analisi.

L'**Orchestrator** è stato inserito per permettere l'integrazione e la parallelizzazione dei due moduli di analisi. Esso ha il compito di gestire la coda delle applicazioni in attesa di essere analizzate, di coordinare la parallelizzazione dei moduli di analisi e di verificare il corretto funzionamento di tutto il sistema. Il **Frame Confusion Detector**, oltre identificare se un'applicazione sia o meno vulnerabile (implementando l'algoritmo in Sezione 7.2 ha anche le mansioni di memorizzare i risultati in un database e di scrivere il report di analisi sotto forma di file di `log`. In tale report vi è un riassunto dell'analisi e vengono riportate tutte le informazioni relative a possibili problemi di sicurezza. In particolare il report indica:

- Elenco delle connessioni HTTP effettuate dall'applicazione.
- Permessi dichiarati e non dichiarati ma utilizzati dall'applicazione.
- File JavaScript che utilizzano funzioni vulnerabili al XSS.
- File che utilizzano librerie JavaScript potenzialmente vulnerabili.
- Elenco dei file (sia HTML che JavaScript) all'interno dell'applicazione e richiesti dinamicamente che contengono il tag *iframe*.

L'implementazione dettagliata dell'architettura di FCDroid verrà descritta nel capitolo successivo.

8 Implementazione

Al fine di mostrare la bontà dell'approccio, il lavoro di tesi propone un'implementazione dell'architettura di FCDroid per l'individuazione della Frame Confusion nelle applicazioni Android.

In questo capitolo vengono discussi gli strumenti e le tecnologie utilizzate e in seguito viene illustrata l'implementazione completa del sistema.

Per mantenere l'approccio scalabile e parallelizzabile si è deciso di implementare l'architettura definita in Fig. 15 utilizzando i microservizi. I microservizi sono piccole unità operative indipendenti che comunicano tramite API basate sul protocollo HTTP. L'utilizzo di un'architettura basata sui microservizi porta a diversi vantaggi: oltre la scalabilità e la parallelizzazione, ogni microservizio può essere sviluppato in un linguaggio di programmazione diverso avendo la sola necessità di esporre delle API comuni per reciproca comunicazione. La loro indipendenza rende inoltre molto semplice una eventuale sostituzione o miglioramento.

Il linguaggio di programmazione scelto per l'implementazione è stato Python 3.6.3 per una serie di motivi esposti nelle seguenti sezioni.

L'implementazione completa di FCDroid è visibile in Fig. 16. Nelle seguenti sezioni verranno analizzati con maggiore dettaglio i moduli che la compongono in maggior dettaglio.

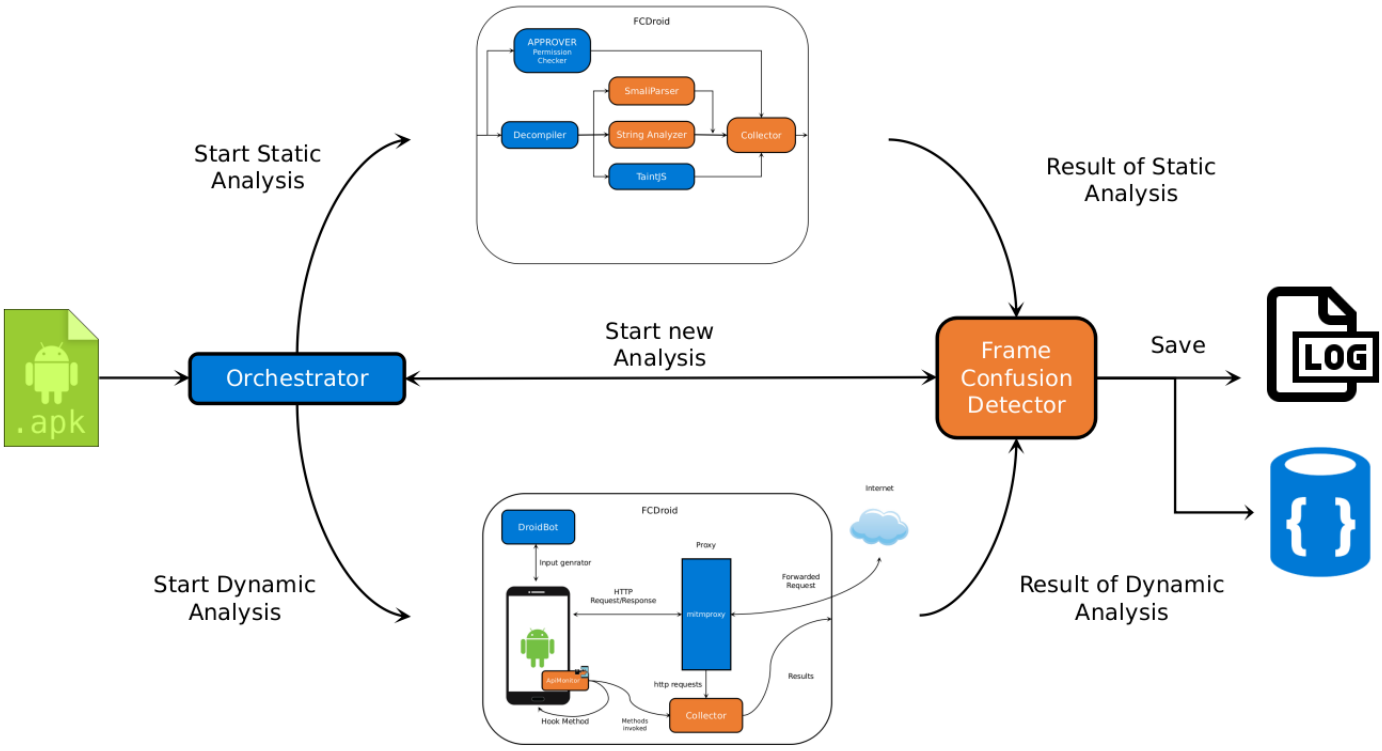


Figura 16: Implementazione dell'architettura di FCDroid

8.1 Analisi Statica

In Fig. 17 viene presentata una panoramica dell'architettura di FCDroid per la sola analisi statica.

I moduli che la compongono sono: `Decompiler`, `Permission Checker`, `TaintJS`, `SmaliParser`, `String Analyzer` e `Collector`. I risultati dell'analisi statica vengono inviati al **Frame Confusion Detector** che ha il compito di implementare la metodologia descritta in Sezione 7 e la cui implementazione verrà descritta in seguito. In questa sezione verranno analizzati con maggior dettaglio i moduli che compongono l'analisi statica.

Decompiler, questo modulo si occupa di decompilare l'applicazione. Per fare ciò si utilizza uno strumento per effettuare *reverse engineering* di applicazioni Android chiamato Apktool [21]. Questo tool è in grado sia di decompilare il *Dalvik bytecode* dell'applicazione in un linguaggio intermedio chiamato Smali [22] sia di decodificare le risorse all'interno dell'applicazione nella loro forma originale come per esempio i file

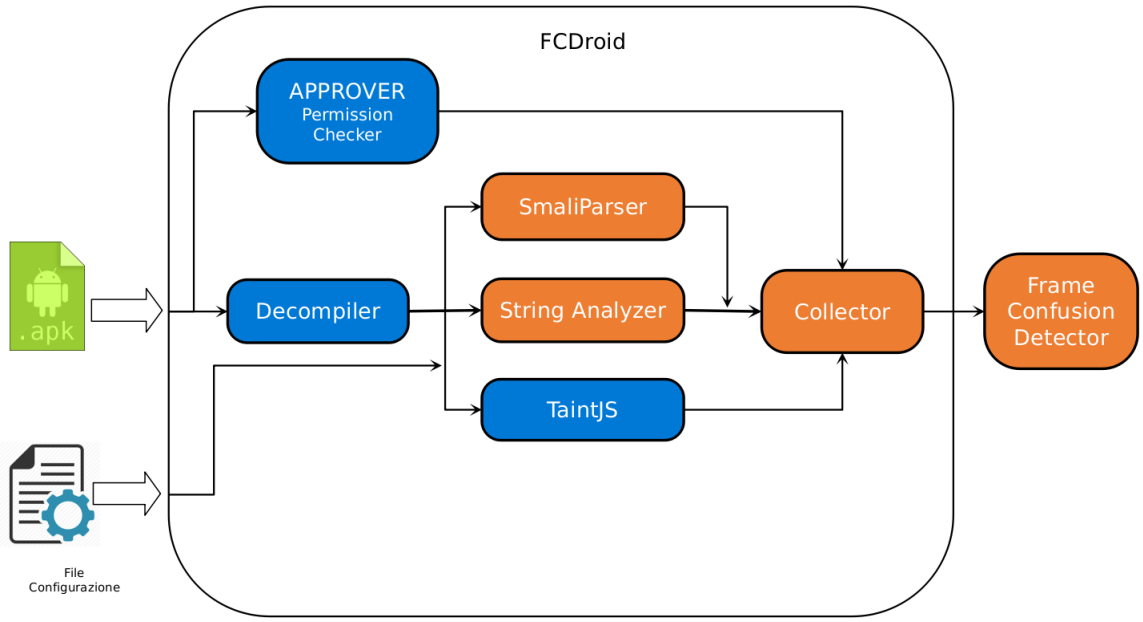


Figura 17: Architettura FCDroid analisi statica

HTML [32]. Inoltre raggruppa tutte le informazioni dell'applicazione in un file chiamato `apktool.yml`. Dopo aver decompilato l'applicazione i vari file vengono passati come input agli altri moduli in particolare: il codice Smali viene passato allo **SmaliParser** e allo **String Analyzer**; i file HTML e JavaScript vengono inviati allo **String Analyzer** e al **TaintJS**.

String Analyzer è uno dei moduli più importanti per quanto riguarda l'analisi statica. Esso prende come input l'applicazione decompilata dal Decompiler ed effettua due tipi di analisi, eseguite in parallelo. Vengono innanzitutto ricercati all'interno del codice Smali tutti gli URL utilizzando uno strumento da linea di comando chiamato `grep`. In parallelo vengono, inoltre, ispezionati tutti i file HTML e JavaScript contenuti all'interno dell'applicazione. Viene ricercato all'interno di questi file eventuali tag *iframe* o funzioni che possano crearli a runtime ("`document.createElement("iframe")`"). Questo viene fatto utilizzando uno libreria per Python chiamata *BeautifoulSoup* [33]. *BeautifoulSoup* è un libreria che facilita l'estrazione di informazioni da file HTML e XML, necessario per analizzare eventuali sezioni di codice JavaScript all'interno di file HTML. Inoltre per ogni file HTML contenente l'elemento *iframe* viene inoltre con-

[32] `https://developer.android.com/guide/topics/manifest/manifest-intro`
[33] `https://pypi.org/project/beautifulsoup4/`

trollata la presenza del meta tag `Content-Security-Policy`, e degli eventuali valori dei suoi attributi. Dopodiché i vari risultati sono inviati al **Collector**.

Il modulo **SmaliParser** prende in input la lista di classi e la loro implementazione rappresentata in codice Smali. Dopodiché per ogni classe ne viene effettuato il parsing per ricavare tutte le informazioni riguardanti i metodi chiave come *loadUrl*, *setJavaScriptEnabled* e *addJavascriptInterface* e i loro parametri.

Per quanto riguarda il metodo *loadUrl* vengono identificati tutti gli URL passategli come argomenti, ad esclusione di quelli creati dinamicamente, ed eventualmente codice `JavaScript` eseguito all'interno della WebView.

Per il metodo *setJavaScriptEnabled*, come è descritto in Algoritmo 1 in Sezione 7.2, viene controllato che il parametro passatogli sia *True*, essendo JavaScript di default disattivato.

Infine viene controllato che venga "attaccata" almeno una interfaccia al componente WebView, semplicemente controllando che sia presente la chiamata *addJavascriptInterface* all'interno del codice decompilato dell'applicazione.

Il risultato di questo analisi è una lista di parametri appartenenti ai metodi selezionati all'interno dell'applicazione.

TaintJS è un modulo basato sul tool **JSPrime** [23], sviluppato in JavaScript. JSPrime è uno strumento che analizza staticamente la sicurezza del codice JavaScript per identificare eventuali DOM-XSS [34]. Viene inizialmente generato l' AST [35] e per ogni variabile ottenuta da un metodo categorizzato come *source* viene seguito il suo flusso fino ad un eventuale metodo etichettato come *sink*. Il flusso della variabile viene seguito per identificare eventuali manipolazioni di essa, se così non fosse allora viene generato un allarme.

L'inserimento di questo modulo è stato necessario per valutare la possibilità e l'impatto eventualmente di poter eseguire codice JavaScript arbitrario nella WebView della vittima. L'output di questo modulo è una lista di possibili file JavaScript vulnerabili al DOM-XSS.

Permission Checker è un modulo appertenente ad Approver [24] che prende in input l'applicazione (il file `apk`) e produce in output una lista di permessi (in formato `JSON`).

[34] `https://www.owasp.org/index.php/DOM_Based_XSS`

[35] È un albero che rappresenta la struttura sintattica astratta del codice sorgente

Nel dettaglio il modulo di analisi raggruppa i permessi in 4 diverse categorie: 1) Dichiarati. 2) Dichiarati e utilizzati. 3) Dichiarati e non utilizzati. 4) Non dichiarati ma utilizzati.

Questo modulo è in grado di identificare anche i permessi che non sono dichiarati nel file `AndroidManifest.xml` ma che in realtà sono necessari per invocare alcuni metodi presenti nel *bytecode* dell'applicazione. Questi permessi, infatti, non sono reperibili utilizzando altri tool come Androguard, ma sono necessari al modulo Frame Confusion Detector per identificare tutti i permessi concessi staticamente all'applicazione o che potrebbero essere concessi all'applicazione a runtime. L'utilizzo di questo modulo è necessario per l'individuazione del permesso `android.permission.INTERNET` necessario al componente WebView per caricare risorse esterne.

Il **Collector** ha il compito di unire i risultati provenienti dagli altri moduli, normalizzarli e di inviarli al **Frame Confusion Detector** che avrà il compito di analizzarli.

8.2 Analisi Dinamica

Per la parte di esecuzione a runtime delle applicazioni e il monitoraggio del traffico di rete, FCDroid utilizza un ambiente emulato ed un proxy come mostrato in Fig. 18

I moduli che compongono la parte di analisi dinamica sono: `Emulatore`, `DroidBot`, `Exposed Framework` e `ApiMonitor`, `Collector` e `Mitmproxy`. Anche per la parte di analisi dinamica i risultati vengono inviati al **Frame Confusion Detector**.

In questa sezione verranno analizzati con maggior dettaglio i moduli che compongono l'analisi dinamica e le motivazioni che hanno portato a scegliere tali strumenti.

Il modulo di analisi dinamica utilizza un **Emulatore** per poter installare ed analizzare dinamicamente le applicazioni a runtime.

Un emulatore è un componente hardware o software che consente ad un sistema (chiamato *host*) di comportarsi come un altro (chiamato *guest*), tipicamente consente al sistema host di eseguire software designato per il sistema guest. In questo contesto lo scopo nell'utilizzare un emulatore è quello di permettere l'installazione e quindi l'esecuzione di applicazioni designate specificamente per il sistema Android.

La scelta su quale tipo di emulatore è stata dettata dalle performance dello stesso. Tipicamente esistono due tipi di emulatore, i quali si distinguono in base all'architettura delle CPU mobile che emulano (x86 e ARM). La prima architettura è poco diffusa

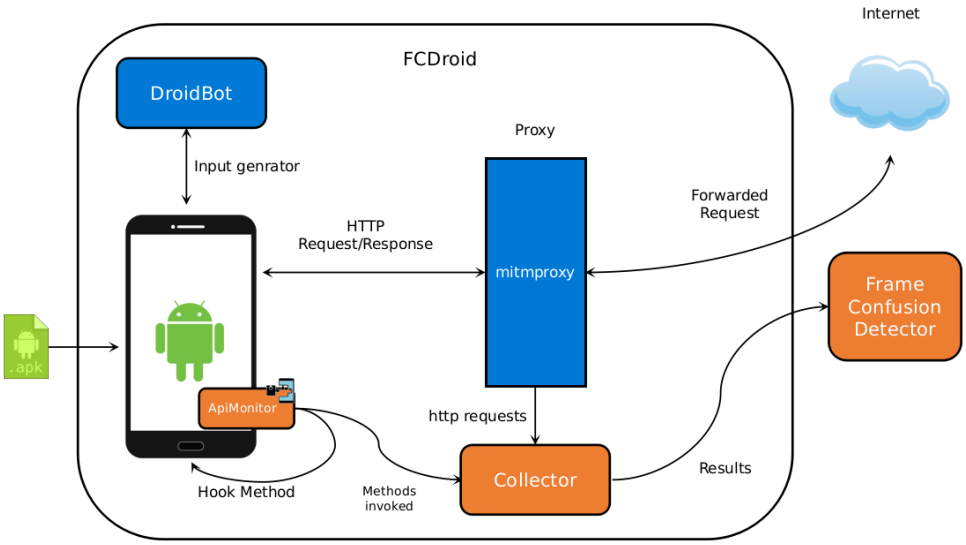


Figura 18: FCDroid Architettura Analisi Dinamica

nel mondo mobile a causa del suo consumo energetico (a favore delle prestazioni), ma essendo compatibile con l'architettura di tutti i sistemi desktop, l'emulatore corrispondente risulta essere efficiente. La seconda, invece, è molto diffusa nei sistemi mobile essendo a basso consumo (un fattore chiave per i dispositivi alimenti a batteria), ma il corrispondente emulatore ha prestazioni inferiori rispetto a quello basato su x86.

Poiché le applicazioni possono essere sviluppate sia per un'architettura in particolare che per entrambe e non conoscendo a priori quali applicazioni saranno analizzate, si è deciso di utilizzare un emulatore basato su Android-x86 .

Android-x86 è un progetto *open-source* che ha come obiettivo quello di rendere eseguibile il sistema operativo Android su dispositivi basati su processori Intel x86 e AMD. Si è deciso di utilizzare questo tipo di emulatore poiché al proprio interno ha un layer chiamato *ARM translation* che permette ad applicazioni che utilizzano librerie per ARM di essere eseguite anche su dispositivi x86, con prestazioni nettamente superiori all'emulatore ARM classico.

Mitmproxy [25] è un proxy HTTPS interattivo, gratis ed open source, necessario per monitorare il traffico di rete dell'emulatore. L'utilizzo di un server proxy è stato essenziale per essere in grado di monitorare e memorizzare il traffico di rete che viene generato dall'applicazione. Esso, infatti, fungendo da intermediario tra il client (in questo caso il device Android) e il server di destinazione, ha la possibilità di monitorare e memorizzare tutto il traffico generato, il quale successivamente verrà inviato al **Frame**

Confusion Detector per essere analizzato.

Una delle funzionalità aggiuntive di questo modulo è l'individuazione di problematiche relative alla verifica dei certificati. Infatti una loro cattiva implementazione come l'accettazione di tutti i certificati (anche quelli self-signed) o non effettuando il certificate pinning (controllo che il certificato ricevuto sia quello aspettato), rende l'applicazione vulnerabile ad un attacco MiTM nonostante venga utilizzato il protocollo HTTPS.

Tutte le gli URL delle richieste effettuate dall'applicazione vengono quindi memorizzate e inviate al Collector che successivamente le invierà al **Frame Confusion Detector** che si occuperà di analizzarle.

DroidBot [26] uno strumento per esplorare in modo automatico l'interfaccia grafica delle applicazioni tramite la generazione di input che simulano le azioni di un'utente reale, necessario per effettuare analisi dinamica.

L'obbiettivo dell'utilizzo di uno strumento come **DroidBot** è quello di riuscire simulare le azioni di un utente reale cercando di attivare tutte le funzionalità che l'applicazione rende disponibili in modo da poter monitorare le API di sistema invocate e il traffico di rete che verrebbero generati da un utilizzo "normale". Lo scopo finale è quello di poter monitorare eventuali comportamenti che potrebbero generare un ambiente favorevole alla Frame Confusion.

Per poter stimolare dinamicamente l'applicazione, DroidBot genera input di test che sono basati su un sistema a transizione di stati generato *on-the-fly* durante l'esecuzione dell'applicazione. A differenza di molti generatori di input esistenti che si affidano all'analisi statica e alla instrumentazione dell'applicazione per generare gli input, DroidBot lavora in modalità *black-box*, cioè non ha bisogno di conoscere a priori la struttura dell'applicazione. Sebbene questo renda difficile l'attivazione alcuni stati specifici, il *trade-off* consente a DroidBot di funzionare con qualsiasi applicazione (offuscate/crittografate o che non rendono possibile la strumentazione) e su una vasta gamma di dispositivi. La Fig. 19 mostra l'architettura di DroidBot.

I moduli principali che lo compongono sono:

- *Adapter* che ha il compito di fornire un'astrazione del device e dell'applicazione sotto test. Ha il compito di mandare i comandi al device, di processare l'output dei comandi, di mantenere la connessione con il device e di risolvere alcuni problemi tecnici come la compatibilità con diverse versioni di Android.

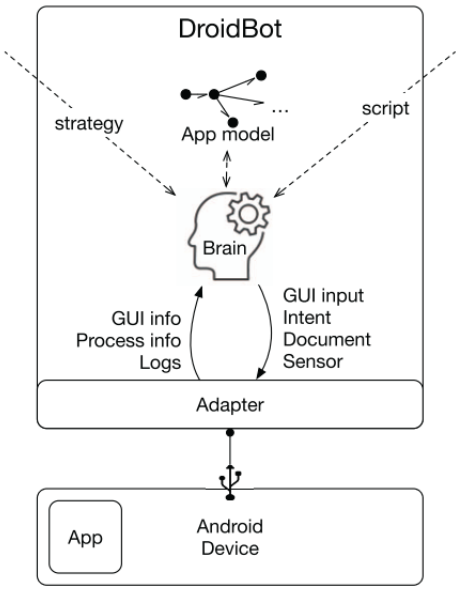


Figura 19: Panoramica DroidBot

- Il modulo *Brain* riceve le informazioni dell'applicazione prodotte dal modulo *Adapter* a runtime e gli invia input di test basandosi su queste informazioni. La generazione di input di test si basa su un grafo di transizione di stato costruito *on-the-fly*. Ogni nodo del grafo rappresenta uno stato del dispositivo, mentre l'arco tra ciascuna coppia di nodi rappresenta l'input di test che ha attivato la transizione di stato.

Per poter monitorare le API che vengono invocate dall'applicazione e il valore dei parametri passati come argomento, FCDroid utilizza un modulo chiamato **ApiMonitor**, basato sul framework **Xposed**.

Xposed è un framework, che richiede permessi di root per eseguire dei moduli aggiuntivi che possono cambiare il comportamento del sistema e delle applicazioni senza modificare l'APK stessa. Il modulo ApiMonitor sfrutta Xposed per monitorare il comportamento di ogni metodo eseguito dall'applicazione in fase di test, salvando su un file `JSON` la sua invocazione e il relativo valore dei parametri passategli come argomento.

La lista di API con i relativi parametri generati dal modulo ApiMonitor, viene quindi inviata in formato `JSON` al Collector che successivamente si occuperà di inoltrarla al **Frame Confusion Detector**.

Il modulo **Collector** ha il compito di collezionare e normalizzare i dati provenienti dal modulo **ApiMonitor** e dal **mitmproxy** e di inviarli al **Frame Confusion Detector** che avrà il compito di analizzarli.

8.3 Frame Confusion Detector

Il **Frame Confusion Detector** è il cuore del sistema ed implementa la metodologia vista in Sezione 7; il modulo ha il compito di determinare se un'applicazione sia vulnerabile o meno alla Frame Confusion, salvare i risultati in modo persistente e di produrre un file di report dettagliato sui risultati ottenuti dall'analisi.

Il Frame Confusion Detector può essere diviso logicamente in due sezioni distinti: la prima relativa all'analisi statica, la seconda relativa a quella dinamica.

Per quanto riguarda l'analisi statica il Frame Confusion Detector richiede al Collector la lista dei metodi invocati dall'applicazione e i relativi parametri (generati dallo SmaliParser) e i permessi dichiarati dall'applicazione (ottenuti dal Permission Checker). I metodi richiesti sono quelli descritti in Algoritmo 1 nella Sezione 7.2. Viene anche recuperata la lista di URL passati come parametro al metodo *loadUrl* (riga 24 Algoritmo 1). In questa lista di URL, vengono inoltre evidenziate tutte le richieste che utilizzano il protocollo HTTP.

Per quanto concerne l'analisi dinamica, il **Frame Confusion Detector**, richiede al Collector (contenuto all'interno del modulo di analisi dinamica) la lista delle richieste HTTP e HTTPS effettuate dall'applicazione e la lista delle API invocate con in relativi parametri. La prima è fornita dal modulo **mitmproxy**, la seconda invece grazie al modulo **ApiMonitor**. Dalla lista di API vengono quindi identificati i metodi necessari all'individuazione della Frame Confusion (Sezione 7.1). Per tutte le occorrenze del metodo *loadUrl* vengono salvati gli URL (passati come parametro) e vengono confrontati e uniti a quelli provenienti dal proxy.

Dopo aver ottenuto tutte le informazioni necessarie, il **Frame Confusion Detector** identifica se l'applicazione sia vulnerabile o meno alla Frame Confusion. Tra i vari risultati che il Frame Confusion Detector memorizza vi sono anche tutti i file potenzialmente vulnerabili al XSS (identificati dal modulo TaintJS) e i file che utilizzano librerie con vulnerabilità note. Quest'ultimi sono identificati grazie allo strumento **RetireJS** [27].

Per la memorizzazione dei risultati si è scelto di utilizzare un database NoSQL. E' stato scelto un database non relazionale per la sua scalabilità ma sopratutto a causa della incapacità di definire a priori la struttura logica dei risultati ottenibili a fine analisi (per esempio le API da voler monitorare possono differire da applicazione ad applicazione).

Dopo aver memorizzato i vari risultati il **Frame Confusion Detector** genera un file di report riassuntivo. Un esempio di tale file è descritto nella sezione successiva.

8.4 Esempio di Output

In questa sezione verrà analizzato un esempio di file di output ottenuto dopo aver effettuato un'analisi.

In Fig. 20 sono visibili le prime righe di un esempio di file di report. Sono visibili i permessi dichiarati e utilizzati e non richiesti ma usati (righe 3-10). Tra i permessi è evidenziato la dichiarazione del permesso relativo a Internet, fondamentale per il componente WebView, inoltre è stata messa in evidenza la tipologia dell'applicazione.

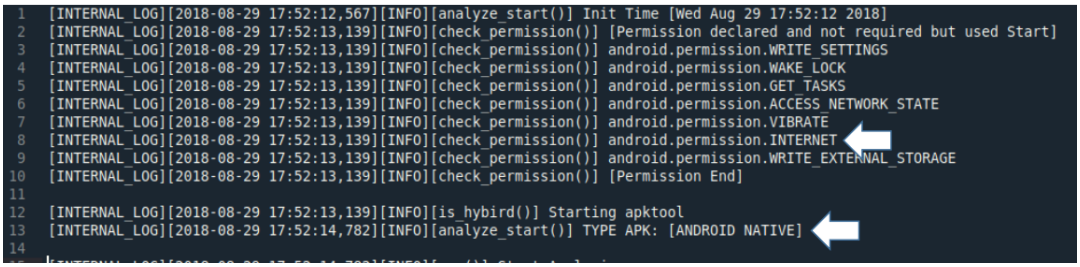


Figura 20: Esempio di output FCDroid

Nella Fig. 21 invece sono riconoscibili gli URL memorizzati dal modulo di **mitmproxy**.

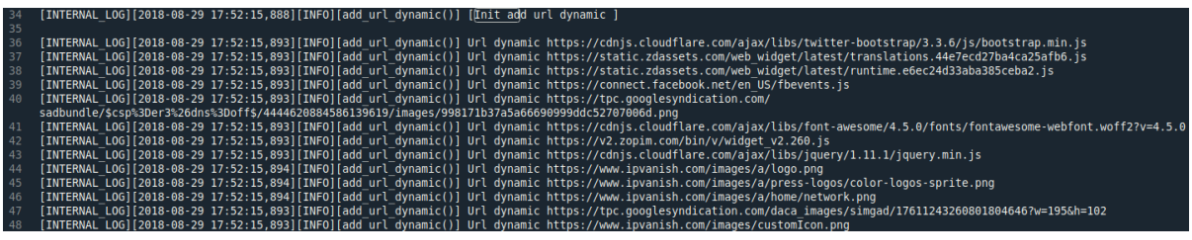


Figura 21: Esempio di output FCDroid (2)

Un esempio di file vulnerabile è visibile in Fig. 22, dove viene indicato il file sospetto (riga 336) e il tag iframe trovato al suo interno(riga 338, 340, 342, 344)), con i relativi

parametri. Inoltre è possibile visionare la mancanze degli eventuali meccanismi di sicurezza (riga 347-348).

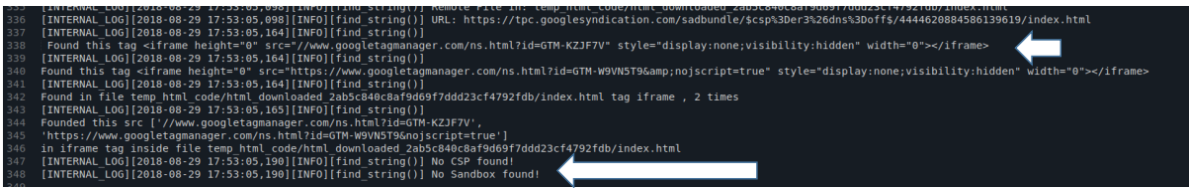


Figura 22: Esempio di output FCDroid (3)

Nelle righe 492 e 493 in Fig. 23 vengono mostrati i metodi *setJavaScriptEnabled* e *addJavascriptInterface* e come la loro esecuzione è stata controllata (dinamicamente). Nella riga 494 invece viene illustrata la vulnerabilità dell'applicazione in esame e i file sospetti che hanno causato l'allarme (righe 495-497).

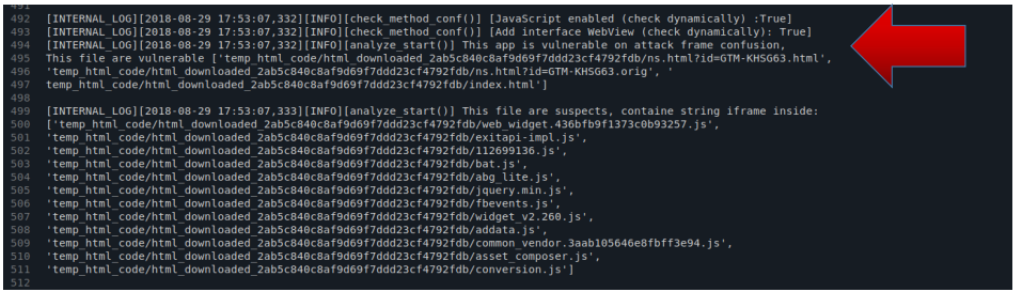


Figura 23: Esempio di output FCDroid (4)

Infine in Fig. 24 vengono mostrati i risultati delle analisi supplementari effettuate da FCDroid. Nelle righe 515-516 le informazioni relative alle connessioni HTTP effettuate dall'applicazione. Mentre nelle righe 520-533 tutti i file JavaScript potenzialmente vulnerabili al XSS.

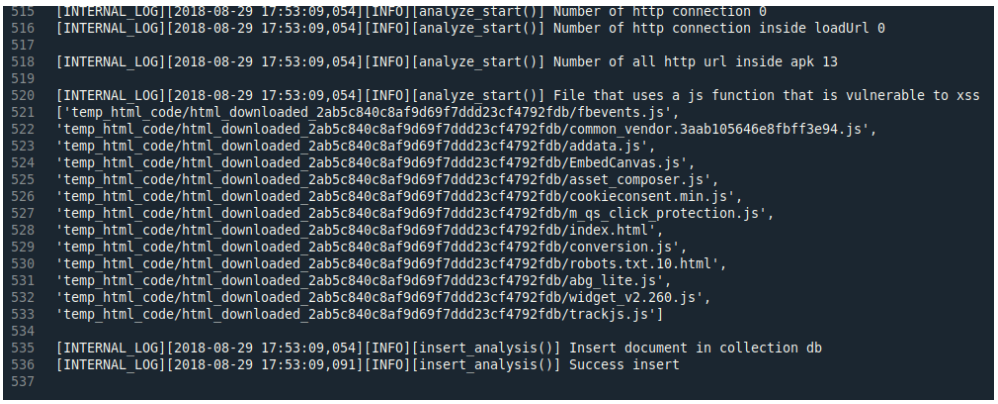


Figura 24: Esempio di output FCDroid (5)

9 Risultati

In questa sezione verranno illustrati i risultati ottenuti testando l'efficacia di FCDroid, valutandone la bontà e la scalabilità analizzando un dataset composto di applicazioni reali disponibili sul Google Play Store[36]. L'ambiente di test utilizzato per eseguire le seguenti analisi è visibile in Tabella 1.

CPU	Intel i7-7500U
RAM	8 GB
Sistema operativo	Ubuntu 16.04
Linguaggio di programmazione	Python 3.6.3

Tabella 1: Specifiche ambiente di analisi

Il Dataset preso in esame è formato da 8.320 applicazioni scaricate a partire dalla top 10.000 presente sul Google Play Store.

9.1 Studio del Dataset

In questa prima fase sono state analizzate le applicazioni del dataset con il fine di:

1) Classificare le loro caratteristiche (presenza di librerie native, dimensione, framework utilizzati, etc.).

2) Verificare l'eventuale predisposizione alla vulnerabilità di Frame Confusion, secondo i pre-requisiti dell'algoritmo Sezione 7.2.

3) Valutare la presenza di condizioni che favoriscano lo sfruttamento della vulnerabilità (presenza di connessioni non protette).

Le 8.320 applicazioni sono state quindi inizialmente analizzate in termini di dimensioni, presenza di librerie native e tipologia di applicazioni. Per quanto riguarda la dimensione, il valore medio e la varianza è visibile in Tabella 2.

[36] `http://play.google.com/store/apps`

Dimensioni Applicazioni	
$dimensione_{media}$ [MB]	σ [MB]
31.2	25.3

Tabella 2: Deviazione standard e media della dimensione in MB delle applicazioni

In Tabella 3 è visibile invece la percentuale di utilizzo delle librerie native all'interno del dataset. Le applicazioni che utilizzano librerie native solo per ARM risultano numerose al contrario delle applicazioni che non utilizzano librerie native.

ARM	37%
ARM e x86	40%
No librerie native	23%

Tabella 3: Utilizzo librerie native nel dataset

La Tabella 4 mostra la distribuzione dei target SDK dichiarati dalle applicazioni nel file `AndroidManifest.xml`; è interessante notare come le applicazioni inferiori alla versione 16 siano ancora così numerose.

Una ulteriore analisi ha permesso di valutare il numero di applicazioni ibride presente all'interno del dataset e quante applicazioni abbiano al proprio interno almeno un file HTML. Come mostrato in Tabella 5, si può notare come, le applicazioni che abbiano almeno un file HTML al proprio interno siano quasi la totalità del dataset, e quindi siano potenzialmente vulnerabili al problema della Frame Confusion. Le applicazioni ibride, invece, risultano essere poco popolari nella top 10.000.

Dopo la fase di classificazione delle app, si é passato ad analizzare il dataset alla ricerca delle pre-condizioni necessarie alla presenza della Frame Confusion. Per prima cosa le applicazioni sono state analizzate staticamente, utilizzando il solo modulo statico di FCDroid alla ricerca dei metodi *setJavaScriptEnabeld* e *addJavaScriptInterface*.

Le percentuali così ottenute , visibili in Tabella 6, appaiono tuttavia estremamente alte; dopo un'attenta analisi manuale a campione dove sono state analizzate 30 applicazioni, solamente il 40% di esse effettivamente attivavano l'interfaccia o abilitavano

Target SDK	Numero applicazioni
26	1676
27	1611
25	1258
23	1029
22	619
21	428
24	413
19	393
17	156
16	125
28	97
18	88
15	77
20	75
10	45
11	25
13	25
9	23
8	13
4	3
7	2
12	1

Tabella 4: Target sdk e numero di applicazioni

Applicazioni che hanno almeno un file HTML all'interno	94%
Applicazioni ibride	1.31%

Tabella 5: Percentuali dataset file html e applicazioni ibride

l'utilizzo di JavaScript. E' stato quindi subito chiaro come il solo approccio statico per analizzare le applicazioni avesse introdotto dei falsi positivi. Si è deciso, pertanto, di ripetere la stessa analisi ma di monitorare le API analizzando dinamicamente le applicazioni con il solo modulo di analisi dinamica di FCDroid.

In Tabella 7 è presente un confronto fra i risultati prodotti dalle due analisi.

Risultati Analisi Statica (basati su 8320 applicazioni)	
Applicazioni con JavaScript abilitato	99.95%
Applicazioni con l'interfaccia disponibile	99.95%
Applicazioni con interfaccia disponibile e JavaScript abilitato	99.94%

Tabella 6: Risultati Analisi Statica basati su 8320 applicazioni

Risultati Analisi (basati su 8320 applicazioni)		
	Analisi Statica	Analisi Dinamica
Applicazioni con JavaScript abilitato	99.95%	48.5%
Applicazioni con l'interfaccia disponibile	99.95%	44.4%
Applicazioni con interfaccia disponibile e JavaScript abilitato	99.94%	44.4%

Tabella 7: Confronto analisi statica e analisi dinamica

Come è possibile osservare in Tabella 7, le applicazioni che utilizzano effettivamente le API di interesse a runtime sono state ridotte del 50% rispetto a quelle identificate dalla sola analisi statica. Questo risultato è una conferma del fatto che l' analisi statica può includere sezioni di codice che non vengono effettivamente mai eseguite a runtime, aumentando il numero dei falsi positivi.

Poiché dai risultati si evince come il componente WebView sia largamente utilizzato all'interno delle applicazioni (circa il 44,4%), si è quindi deciso di monitorare le API più significative ad esso associate. Tali API sono: *loadUrl*, *evaluateJavascript* e le API analizzate precedentemente *setJavaScriptEnabled*, *addJavascriptInterface*. I risultati sono visibili in Tabella 8.

In Tabella 9, invece, sono presenti le percentuali di utilizzo delle API più significative

setJavascriptEnabled	48.5%
loadUrl	46.1%
addJavaScriptInterface	44.4%
evaluateJavascript	40,04%

Tabella 8: Percentuale di utilizzo delle API WebVIew nella top 10.000

setJavascriptEnabled e addJavaScriptInterface	44.4%
loadUrl e evaluateJavascript	38.7%
loadUrl, evaluateJavascript, setJavascriptEnabled e addJavaScriptInterface	38.6%

Tabella 9: Percentuale di utilizzo simultaneo delle API WebVIew nella top 10.000

utilizzate simultaneamente. Dal tabella è possibile osservare come l'utilizzo di tutte le API significative sia molto frequente nelle applicazioni analizzate.

Si è quindi deciso di analizzare il numero di applicazioni che utilizzano i metodi *setJavaScriptEnabled* e *addJavaScriptInterface* e che carichino almeno una pagina HTML all'interno della WebView (e quindi potenzialmente vulnerabili). Andando a valutare quante di esse implementino almeno un meccanismo di sicurezza. E' risultato che solo il 13.8% di tali applicazioni implementano il meccanismo di sicurezza CSP, discusso in Sezione 5, che avrebbe permesso di mitigare la Frame Confusion.

Nell'ultima parte di questa fase di test si sono valutate alcune delle condizioni che favoriscono l'*exploit* della vulnerabilità Frame Confusion; in particolare sono state valutate le applicazioni che, soddisfando tutti i pre-requisiti, presentano connessioni non sicure (HTTP) all'interno del metodo *loadUrl*.

Le applicazioni che utilizzano un protocollo non sicuro per accedere a risorse online sono 305, come visibile in Tabella 10, ovvero circa il 10% di quelle che utilizzano tutti e quattro i metodi significativi del componente WebView. Queste applicazioni sono vulnerabili ad attacchi MiTM, nel quale un'attaccante potrebbe compromettere facil-

Applicazioni che utilizzano tutte le API e protocollo HTTP	
Numero di applicazioni	305

Tabella 10: Applicazioni vulnerabili al MiTM

mente il traffico di rete, accedere facilmente all'interfaccia JavaScript, ed invocare, quindi, codice nativo Java.

9.2 Test con FCDroid

Nella secondo fase di test sono state analizzate le applicazioni con FCDroid per identificare quante applicazioni fossero vulnerabili alla Frame Confusion. I risultati, visibili in Tabella 11, sono quindi confrontati con quelli ottenuti dalle fasi di analisi prese singolarmente.

Risultati Analisi Frame Confusion (basati su 8320 applicazioni)		
Analisi Statica	Analisi Dinamica	FCDroid
3.7%	1.2%	1.96%

Tabella 11: Confronto analisi statica e analisi dinamica con FCDroid

I valori in Tabella 11 mostrano come FCDroid si ponga proprio tra le due tipologie di analisi: l'utilizzo della metodologia ibrida è in grado di ridurre sia i falsi positivi (prodotti dall'analisi statica) che i falsi negativi (prodotti dall'analisi dinamica). FCDroid è dunque in grado di mostrare risultati molto più accurati superano di limiti delle singole tecniche di analisi.

Si sono quindi fatte ulteriori analisi sulle 163 applicazioni evidenziate come vulnerabili da FCDroid. In particolare è stata analizzata:

1) La presenza del protocollo HTTP.

2) L'utilizzo dell'interfaccia JavaScript con invocazione di metodi che richiedono permessi dangerous per essere invocati.

I risultati di quest'analisi sono visibili in Tabella 12. In particolare il 41.2% delle applicazioni evidenziate come vulnerabili da FCDroid invocano metodi *dangerous*. Questo valore risulta molto elevato, infatti, da tali applicazioni potrebbe essere possibile invocare metodi utilizzando l'interfaccia JavaScript per inviare SMS, avviare chiamate, utilizzare la fotocamera etc. Il 15.3% delle applicazioni invece utilizza il protocollo HTTP per caricare pagine o iframe all'interno della WebView, essendo quindi vulnerabili ad attacchi MiTM. Sono di particolare interesse inoltre le applicazioni che abbiano presentato entrambi i comportamenti : utilizzo del protocollo HTTP e utilizzo di API dangerous. Tali applicazioni, che coprono il 9.2% delle applicazioni vulnerabili alla Frame Confusioni sono state soggette ad ulteriori analisi manuali.

Risultati Analisi Applicazioni Vulnerabili (basati su 163 applicazioni)	
Applicazioni che utilizzano metodi dangerous	41,2%
Applicazioni che utilizzano protocollo HTTP	15,3%
Applicazioni che utilizzano metodi dangerous e protocollo HTTP	9,2%

Tabella 12: Risultati Analisi Applicazioni Vulnerabili

Di queste applicazioni, due di esse, vulnerabili alla Frame Confusion hanno un grosso impatto sugli utilizzatori. In una di esse, infatti, è stato possibile avviare una chiamata, nella seconda, invece, si è potuto effettuare un attacco del tipo Denial Of Service (DOS) sull'applicazione. Un esempio di questi attacchi verrà discusso nell'ultima sezione.

FCDroid durante la fase di controllo della presenza della vulnerabilità di Frame Confusion, analizza anche le applicazioni alla ricerca di potenziali vulnerabilità XSS. Dall'analisi complessiva dell'intero dataset sono state individuate librerie di versioni con vulnerabilità note nel 3.21% delle app, mentre il 61% delle librerie JavaScript prese sotto esame può essere soggetta a potenziali attacchi XSS.

9.2.1 Considerazioni sul tempo di analisi

Per dimostrare la scalabilità dell'approccio in questa sezione si analizzeranno le prestazioni di FCDroid. I tempi di analisi sono visibili in Tabella 13. Il tempo fissato per analizzare dinamicamente le applicazioni è stato di 100 secondi, questo valore è stato scelto come buon compromesso per rendere l'approccio scalabile e per stimolare sufficientemente l'applicazione. Un tempo inferiore non avrebbe stimolato sufficientemente l'applicazione, invece l'utilizzo di tempi maggiori non ha portato ad ulteriori benefici. La stimolazione delle applicazioni, infatti, rimaneva limitata ad una sola pagina, nella quale venivano ripetute sempre le stesse azioni.

Analisi Statica	
t_{medio} [s]	σ [s]
23.8	33.6
Analisi Dinamica	
t_{medio} [s]	σ [s]
100	0
Frame Confusion Detector	
t_{medio} [s]	σ [s]
10.2	15.3

Tabella 13: Tempi di Analisi

Si può notare come il tempo medio di esecuzione per l'analisi statica sia relativamente basso. Sommando i 100 secondi di analisi dinamica e i 10.2 secondi necessari a FCDroid si ottiene un tempo di analisi medio per ogni applicazione di circa 134 secondi. Questo valore dimostra come l'approccio sia scalabile, inoltre, il tempo di analisi dinamica può essere aumentato o diminuito a seconda delle esigenze.

9.3 Esempio di attacco su applicazione reale

In questa sezione viene illustrato un attacco effettuato su un'applicazione reale. L'applicazione in questione è `com.hnsmall` disponibile su Google Play Store (Fig. 25) e

rivolta ad un pubblico asiatico. L'applicazione si presenta come uno classico store on-line in cui comprare oggetti per la casa e indumenti in offerta on-line e presenta oltre 10.000.000 di download (Fig. 26).

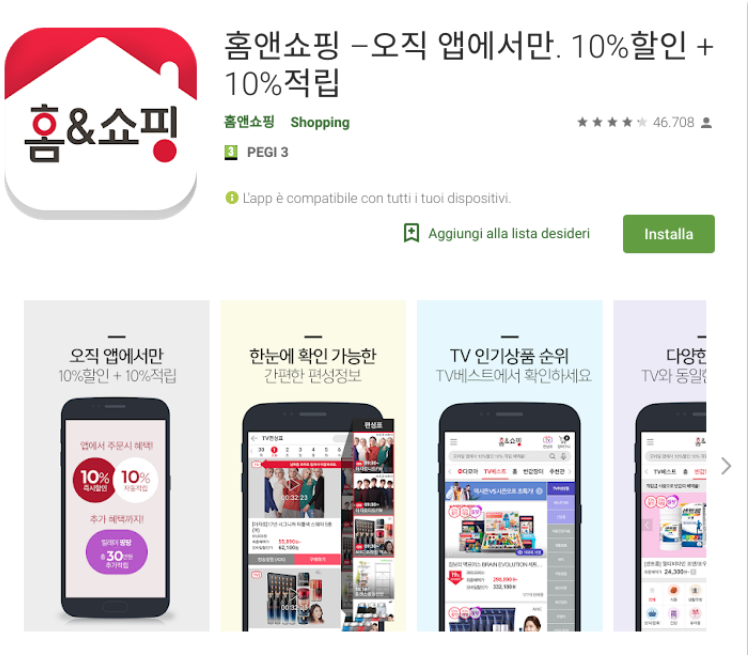


Figura 25: Applicazione sul Google Play Store

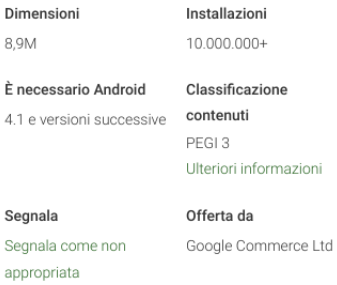


Figura 26: Informazioni sul Google Play Store

L'applicazione utilizza il componente WebView per caricare l'home page del sito utilizzando il protocollo HTTP (Fig. 27). All'interno di questa pagina vi sono diversi iframe non visibili a schermo (impostano l'attributo `display:none`) e vengono caricati anch'essi con il protocollo HTTP .

Inoltre l'applicazione abilita diverse interfacce JavaScript, la più interessante è chiamata `goArgsCall`. Questa interfaccia annota un metodo chiamato `callAndroid()` che avvia una chiamata telefonica.

Figura 27: Schermata iniziale applicazione vulnerabile

In Fig. 28 è illustrato l'attacco nelle sue fasi. Nella prima fase viene intercettata la richiesta HTTP proveniente dall'applicazione. Tale richiesta non viene alterata ma viene inoltrata semplicemente al server di destinazione. La risposta del server, invece, viene intercettata e alterata inserendo un iframe con all'interno la funzione che si occuperà di accedere all'interfaccia e avviare la chiamata (`goArgsCall.callAndroid()`). Dopo l'alterazione la risposta viene inoltrata all'applicazione. La WebView, all'interno di essa, processerà la pagina ricevuta, compreso il codice inserito dall'attaccante, non potendo distinguerlo dall'originale, che quindi avvia la chiamata.

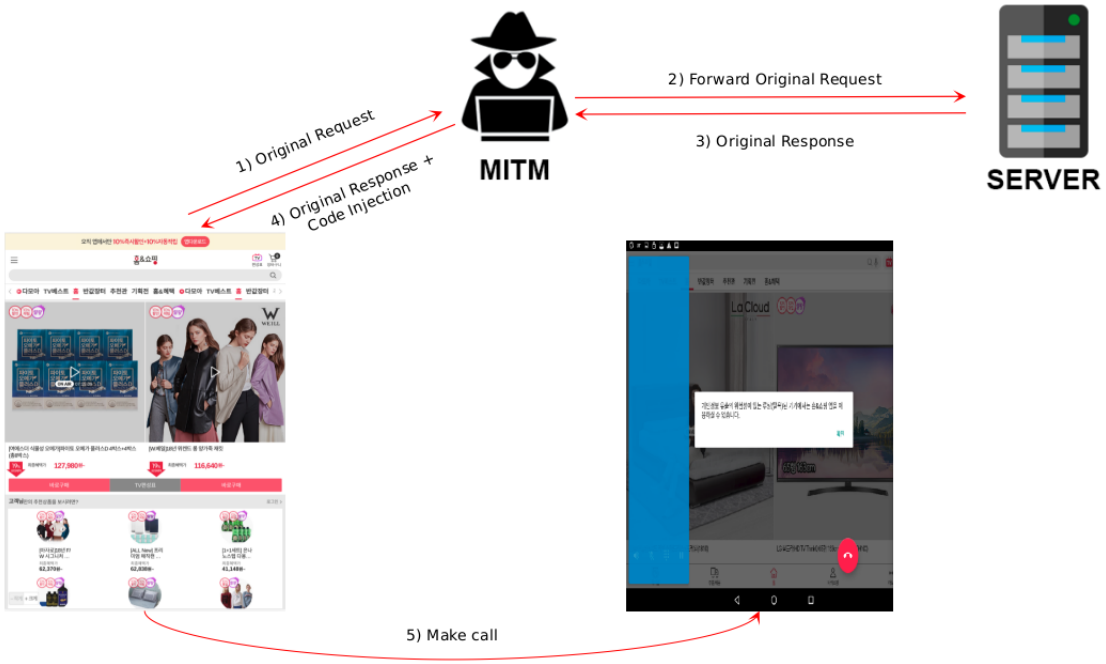


Figura 28: Spiegazione attacco

10 Conclusioni e Sviluppi futuri

In questa tesi è stato proposto FCDroid, una metodologia per identificare la vulnerabilità della Frame Confusion utilizzando un approccio ibrido. Tale vulnerabilità, anche se nota alla comunità scientifica, non è mai stata trattata con le dovute attenzione, una prova è il fatto che non sia stata ancora risolta e che non esista uno strumento che sia in grado di identificarla in maniera automatica.

Inizialmente la metodologia ideata era basata solamente sull'analisi statica. I risultati prodotti, però, anche se molto promettenti inizialmente si sono rilevati falsati dopo un'attenta analisi manuali. In seguito si è quindi deciso di sviluppare una metodologia ibrida che sia in grado di aggregare in maniera automatica i risultati prodotti dall'analisi statica e dall'analisi dinamica. Per valutarne la bontà e la scalabilità si sono quindi analizzate le prime 8.320 applicazioni provenienti dalla top 10.000 presenti nel Google Play Store. I risultati ottenuti hanno potuto dimostrare come le applicazioni che utilizzano il componente WebView e usufruiscono delle API responsabili alla formazione della Frame Confusion siano davvero numerose. Infatti, oltre il 40% delle applicazioni presenti nella top 10.000 fanno uso di tale componente, e circa il 2% delle applicazioni invece sono risultate vulnerabili. Tale vulnerabilità, è stata sottovalutata dalla comunità scientifica, è stato infatti possibile: effettuare chiamate e effettuare un

attacco DOS su applicazioni con svariati milioni di utenti. Ci si chiede che tipo di attacchi si potrebbero portare a termine sfruttando la Frame Confusion se si estendesse l'analisi alle prime 100.000 applicazioni.

L'obiettivo nel creare tale metodologia è stato dimostrare come tale vulnerabilità sia stata sottovalutata e come, nel corso degli anni, con l'avvento delle applicazioni ibride e con la tendenza degli sviluppatori a unire sempre di più elementi provenienti dal mondo web e dal mondo mobile sia diventata sempre più pericolosa.

Sviluppi Futuri

Questo lavoro di laurea lascia spazio a diversi lavori futuri. Sicuramente uno dei più urgenti è quello di sviluppare una *patch* di sicurezza nel sistema Android, che di default, abiliti la possibilità di accedere all'interfaccia JavaScript solamente a domini inseriti in una whitelist. Dopodiché uno dei possibili sviluppi futuri è quello colmare il vuoti all'interno della letteratura scientifica e di sviluppare una metodologia di analisi di sicurezza automatica per le applicazioni ibride, le quali utilizzando elementi web e mobile in costante comunicazione tra di loro rende le attuali metodologie di analisi inefficaci.

Riferimenti bibliografici

[1] D. R. Thomas, A. R. Beresford, T. Coudray, T. Sutcliffe, and A. Taylor, "The lifetime of android API vulnerabilities: Case study on the JavaScript-to-Java interface," *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 9379, pp. 126–138, 2015.

[2] T. Luo, H. Hao, W. Du, Y. Wang, and H. Yin, "Attacks on WebView in the Android system," *Proceedings of the 27th Annual Computer Security Applications Conference on - ACSAC '11*, p. 343, 2011. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2076732.2076781

[3] E. Chin and D. Wagner, "Bifocals: Analyzing webview vulnerabilities in android applications," *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 8267 LNCS, pp. 138–159, 2014.

[4] C. Rizzo, L. Cavallaro, and J. Kinder, "BabelView: Evaluating the Impact of Code Injection Attacks in Mobile Webviews," *21st International Symposium on Research in Attacks, Intrusions and Defenses*, 2017. [Online]. Available: http://arxiv.org/abs/1709.05690

[5] J. Yu and T. Yamauchi, "Access control to prevent malicious javascript code exploiting vulnerabilities of webview in android OS," *IEICE Transactions on Information and Systems*, vol. E98D, no. 4, pp. 807–811, 2015.

[6] X. Jin, X. Hu, K. Ying, W. Du, H. Yin, and G. N. Peri, "Code Injection Attacks on HTML5-based Mobile Apps," *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security - CCS '14*, pp. 66–77, 2014. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2660267.2660275

[7] C. S. Policy, "https://content-security-policy.com." [Online]. Available: https://content-security-policy.com

[8] Gartner, "https://www.gartner.com/newsroom/id/2429815." [Online]. Available: https://www.gartner.com/newsroom/id/2429815

[9] A. Brain, "https://www.appbrain.com." [Online]. Available: https://www.appbrain.com

[10] H. Chen, H. F. Leung, B. Han, and J. Su, "Automatic privacy leakage detection for massive android apps via a novel hybrid approach," *IEEE International Conference on Communications*, 2017.

[11] M. Neugschwandtner, M. Lindorfer, and C. Platzer, "A View to a Kill: WebView Exploitation," *Leet*, 2013. [Online]. Available: http://publik.tuwien.ac.at/files/PubDat_223415.pdf

[12] W. Bao, W. Yao, M. Zong, and D. Wang, "Cross-site Scripting Attacks on Android Hybrid Applications." *Iccsp*, pp. 56–61, 2017. [Online]. Available: http://dblp.uni-trier.de/db/conf/iccsp/iccsp2017.html{#}BaoYZW17

[13] H. Bagheri, J. Garcia, S. Malek, A. Sadeghi, H. Bagheri, J. Garcia, and S. Malek, "A Taxonomy and Qualitative Comparison of Program Analysis Techniques for Security Assessment of Android Apps," *ISR Technical Report*, vol. 43, no. 6, p. 24, 2016.

[14] L. Li, T. F. Bissyandé, M. Papadakis, S. Rasthofer, A. Bartel, D. Octeau, J. Klein, and L. Traon, "Static analysis of android apps: A systematic literature review," *Information and Software Technology*, vol. 88, pp. 67–95, 2017.

[15] S. Lee, J. Dolby, and S. Ryu, "HybriDroid: static analysis framework for Android hybrid applications," *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering - ASE 2016*, pp. 250–261, 2016. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2970276.2970368

[16] J. Sexton, A. Chudnov, and D. A. Naumann, "Spartan Jester: End-to-end information flow control for hybrid android applications," *Proceedings - 2017 IEEE Symposium on Security and Privacy Workshops, SPW 2017*, vol. 2017-December, pp. 157–162, 2017.

[17] M. D. Ernst, R. Just, S. Millstein, W. Dietl, S. Pernsteiner, F. Roesner, K. Koscher, P. Barros, R. Bhoraskar, S. Han, P. Vines, and E. X. Wu, "Collaborative Verification of Information Flow for a High-Assurance App Store." [Online]. Available: http://dx.doi.org/10.1145/2660267.2660343.

[18] A. Chudnov and D. A. Naumann, "Inlined Information Flow Monitoring for JavaScript." [Online]. Available: https://www.cs.stevens.edu/{~}Naumann/publications/ccs2015.pdf

[19] Y. L. Chen, H. M. Lee, A. B. Jeng, and T. E. Wei, "DroidCIA: A novel detection method of code injection attacks on HTML5-based mobile apps," *Proceedings - 14th IEEE International Conference on Trust, Security and Privacy in Computing and Communications, TrustCom 2015*, vol. 1, pp. 1014–1021, 2015.

[20] R. Yan, X. Xiao, G. Hu, S. Peng, and Y. Jiang, "New deep learning method to detect code injection attacks on hybrid applications," *Journal of Systems and Software*, vol. 137, 2018.

[21] R. Winiewski and C. Tumbleson. Apktool - A tool for reverse engineering Android apk files. [Online]. Available: http://ibotpeaches.github.io/Apktool/

[22] B. Gruver. Smali - Assembler/Disassembler for the dex format. [Online]. Available: http://github.com/JesusFreke/smali/

[23] Nishant Das Patnaik, Sarathi Sabyasachi Sahoo . (2013) JSPrime. [Online]. Available: https://dpnishant.github.io/jsprime/

[24] Talos S.r.l.s. (2016) Approver. [Online]. Available: http://www.talos-security.com/

[25] A. Cortesi, M. Hils, T. Kriechbaumer, and contributors, "mitmproxy: A free and open source interactive HTTPS proxy," 2010–, [Version 4.0]. [Online]. Available: https://mitmproxy.org/

[26] Y. Li, Z. Yang, Y. Guo, and X. Chen, "DroidBot: A lightweight UI-guided test input generator for android," *Proceedings - 2017 IEEE/ACM 39th International Conference on Software Engineering Companion, ICSE-C 2017*, pp. 23–26, 2017.

[27] O. Erlend. RetireJS - Scanner detecting the use of JavaScript libraries with known vulnerabilities. [Online]. Available: https://retirejs.github.io/retire.js/