Viaggio nei malware Android: l’incubo Ransomware (Parte 3)

(Parte 3)

Introduzione

Nel primo articolo di questa serie dedicata ai ransomware, abbiamo fornito una descrizione generale delle famiglie più comuni legate a questo tipo di attacco.

Nel secondo articolo ci siamo concentrati su un attacco di esempio, descrivendone le caratteristiche generali e gli elementi principali del file AndroidManifest.xml.

In quest’ultima parte, ci concentreremo sull’analisi più approfondita del file **classes.dex**, (che rappresenta anche la parte più tecnica e complessa di un’applicazione), fornendo un piccolo esempio tratto proprio dalla applicazione analizzata nel precedente articolo.

Si premette che questa è un’analisi tecnica che richiede delle competenze di base di programmazione ad oggetti. L’analisi qui contenuta è lungi dall’essere esaustiva, e deve solamente essere considerata come punto di partenza per analisi più approfondite.

Inizieremo la descrizione fornendo una panoramica delle caratteristiche del **classes.dex** (ed in particolare del bytecode Dalvik), e proseguiremo concentrandoci sull’esempio che abbiamo iniziato ad esplorare nel precedente articolo.

Caratteristiche Classes.dex

Il **classes.dex** è un file che contiene il codice relativo a tutte le classi, metodi ed attributi implementati dall’utente. Le istruzioni sono scritte **in Dalvik bytecode**, una

rappresentazione intermedia fra linguaggio macchina (Assembly ARM) e codice sorgente (Java). Le istruzioni in Dalvik bytecode operano su **registri virtuali**, che non coincidono ovviamente con il numero di registri del processore ARM (in Dalvik è possibile infatti usarne un numero molto maggiore), ma che vengono utilizzati per garantire una maggiore flessibilità e velocità rispetto al bytecode Java (che preferisce l'utilizzo di una architettura a stack, priva di registri). Ogni registro inizia con una "v", susseguita da un numero.

Di seguito, riportiamo le istruzioni principali utilizzate nell'analisi del classes.dex. Per un approfondimento (praticamente obbligato) si consiglia di riferirsi alla documentazione ufficiale [1]. Come regola generale, queste istruzioni utilizzano la notazione:

[Istruzione][Registri][Parametri]

- **get/put**: Sono istruzioni che consistono nell'inserimento del valore di un campo nel registro di destinazione (**get**), o viceversa (**put**). Le istruzioni possono essere riferite ad attributi statici (**sget/sput)** o di istanza (**iget/iput**). Notare come il nome delle istruzioni possa essere esteso a seconda del tipo di oggetto trattato. Ad esempio, se viene trattato un tipo fondamentale **char**, allora l'istruzione **iget** diventerà **iget-char.** Per oggetti di tipi non fondamentali si utilizzerà invece l'estensione **object** (es. **iget-object**).

Esempio:

sget-object v4, MyClass.foo

Inserisce il contenuto di MyClass.foo nel registro v4 (e foo è di un tipo non fondamentale).

- **new-instance**: Inizializza una nuova istanza di classe (in maniera molto simile a quella che viene fatta col

comando **new** in Java).

Esempio:

new-instance v4, MyFoo: definisce una nuova istanza della classe MyFoo.

- **const-string**: definisce una stringa costante da inserire in un registro.

Esempio:

const-string v4, "bla": inserisce la stringa "bla" nel registro V4

Un discorso a parte meritano le **invocazioni**. Sono forse il tipo di istruzione Dalvik più difficile da gestire, perché sono molto più lunghe e complesse da leggere. La formulazione generale di tale istruzioni può essere così rappresentata:

Invoke-[Virtual/Direct/Static/Interface] {registry parametri}, L{Nome_Classe}->{Metodi}(Nomi Parametri){Tipo di Ritorno}

Ogni invocazione inizia sempre col nome **Invoke**, susseguito dal tipo di invocazione (se è **Static** si tratta di un metodo statico, se è **Virtual** di un metodo di istanza, etc.), dai **registri che corrispondono ai parametri** (uno o più per parametro), dalla classe del metodo invocato (sempre introdotta dalla lettera **L**), dal **nome del metodo**, dal **nome dei parametri** (nello stesso ordine dei registri) ed infine dal **tipo di ritorno**.

Qui vanno sottolineati due aspetti importanti. Il primo riguarda il numero dei parametri. Nella maggior parte dei casi, il numero dei parametri è sempre **più grande di uno** rispetto al numero dei parametri nominali. Questo perché il primo parametro corrisponde all'istanza della classe o alla classe stessa (sostanzialmente l'equivalente del **this** in Java).

Il secondo aspetto concerne la rappresentazione dei tipi di ritorno. Per i tipi fondamentali, si utilizza una notazione abbreviata ad una sola lettera. Ad esempio, **int** corrisponde a **I**, **void** a **V**, e **boolean** a **Z**. Non è scopo di questo articolo descrivere tutti i tipi, i quali possono essere facilmente individuati dalla documentazione Dalvik ufficiale [1].

Infine, è bene ricordare che le istruzioni invoke sono solitamente susseguite da una istruzione **move-result** (anche questa espandibile, ad esempio con **object**), in quanto il risultato dell'invocazione deve essere inserito dentro un registro.

A titolo di esempio, si consideri la seguente invocazione:

invoke-virtual{v0,v1}, LMyClass->foo(I)V

Il metodo chiamato è foo, appartenente alla classe MyClass. Il metodo prende in ingresso un intero (rappresentato dalla lettera I), corrispondente a v1. v0 è l'oggetto this, relativo a MyClass. Notate quindi come il numero di registri usato sia più grande di uno rispetto al numero di parametri. Il metodo non ritorna alcun valore, quindi il tipo di ritorno è void (V).

Analisi Ransomware

Veniamo ora all'analisi del ransomware vero e proprio (che abbiamo iniziato ad analizzare nel precedente articolo). Come osservazione preliminare, è importante osservare che questo campione ha una complessità tecnica abbastanza elevata. Pertanto, in questo articolo si daranno solo degli spunti di base per capire come procedere ad un'analisi poi più approfondita.

ApkTool crea dei file in formato **.smali**, che consente una lettura semplificata del bytecode. In particolare, dentro la cartella "smali" verrà creata una struttura di cartelle che riflette quella dei package. Ad esempio, se il package ha il

nome “foo.bar”, dentro la cartella “smali” vi sarà una sottocartella “foo”, che conterrà a sua volta “bar”. All’interno di queste cartelle, ApkTool cercherà di creare **un file .smali per classe**, in maniera simile a quello che succede con il codice sorgente Java.

Di norma, si inizia l’analisi del campione partendo dall’attività con intent-filter MAIN/LAUNCHER (vedi il precedente articolo). Nel nostro caso, tale attività corrisponde a **com.zynga.FarmVilleTropicEscape.FarmVilleTropicEscape.**

Andiamo a cercare quindi il corrispondente file **FarmVilleTropicEscape.smali** nella cartella **com/zynga/FarmVilleTropicEscape (**Nota bene: verranno create anche cartelle con altri package, che però non sono di nostro interesse, essendo associate a pacchetti di sistema**).** Il primo metodo da cercare è **onCreate**, che rappresenta ciò che viene immediatamente eseguito al caricamento dell’attività.

L’idea generale è quella di ricostruire il **grafo dei metodi implementati dall’utente** partendo dal metodo sopra citato, in modo da poter seguire un primo percorso di analisi. In questa prima fase, si presta meno attenzione ai metodi di sistema, ovvero quei metodi utilizzati nella documentazione principale di Android. Una volta determinato il grafo, si esplora metodo per metodo più nel dettaglio alla ricerca di chiamate sospette (ovviamente non bisogna solo guardare il tipo di invocazione, ma anche e soprattutto dove queste agiscono, quindi i loro parametri). La cosa più importante in assoluto è che **non è necessario analizzare ogni singola riga di codice per avere una idea sommaria sulle azioni svolte dall’applicazione.** Una volta sviluppata la giusta “sensibilità”, ci si può soffermare sugli aspetti che risaltano maggiormente: stringhe legate all’invio di sms, siti web da contattare, e via discorrendo.

Naturalmente, in questa sede verrà proposta solo una piccola parte di tale grafo e di tale analisi.

Dal metodo **onCreate** è possibile individuare otto invocazioni. Di queste otto, ci concentriamo sulle istruzioni di tipo **virtual**, che rappresentano metodi di istanza per l'oggetto che viene creato. Sono presenti quattro metodi di questa tipologia, e noi ne analizzeremo, per ragioni di spazio, solo due:

- **FarmVilleTavropicEscape;->concates(Ljava/lang/String;)Ljava/lang/String**
- **FarmVilleTropicEscape;>startService(Landroid/content/Intent;)Landroid/content/ComponentName**

Il primo metodo è sicuramente uno dei più sospetti. Infatti, il nome farebbe pensare alla funzione **concat**, il cui ruolo è quello di concatenare due stringhe. Tuttavia, il parametro in ingresso della funzione, definito dall'istruzione **const-string v1, "1f26b9af2f3340212eb8d1d1b1fbba3d"**, è molto strano. Da notare anche il nome della classe dell'oggetto: **FarmVilleTavropicEscape**. Ovviamente, il nome è utilizzato per confondere l'analista. Andiamo quindi nel file **FarmVilleTavropicEscape.smali** e cerchiamo il metodo **concates**. Il metodo è notevolmente complesso, ma è possibile notare alcuni aspetti "generali". In questa sede ne indicheremo due:

1. Sono presenti diversi riferimenti alle funzioni della classe **crypto**, tipicamente usata per funzioni di crittografia. Questa famiglia di funzioni è notevolmente usata dai ransomware per criptare i dati o per nascondere delle informazioni all'occhio dell'analista.
2. All'interno di questo metodo, è possibile trovare due **const-string** che si susseguono: **const-string v1, "getIn"** e **const-string v2, "stance".** Viene poi chiamata la funzione **concat** sulle due stringhe (da non confondere con concates). Dopodiché, il suo risultato verrà usato come parametro per il metodo **getMethod** relativo alla famiglia **lang.reflect**. Questo significa che un metodo di nome "**getInstance"** verrà chiamato attraverso la **reflection**. La reflection è una tecnica che consente di

accedere ad un metodo o ad un attributo indirettamente, semplicemente specificandone il nome come stringa e dandolo come parametro ad una funzione della famiglia java.lang.reflection. Ovviamente, tale strategia non avrebbe senso rispetto ad una chiamata diretta perché è ovviamente molto più lenta dal punto di vista computazionale, ma aumenta la difficoltà di lettura del metodo stesso.

Metodi come **concates** sono ricchi di complessità e non è possibile, in un'analisi come quella proposta qui, fornire troppi dettagli tecnici. La primissima conclusione che è possibile trarre, però, è che l'analisi statica non sarà sufficiente a comprendere a fondo il funzionamento del malware, in quanto le strategie di encryption sono spesso "decodificate" a runtime. Allo stesso tempo, però, da una primissima analisi è già possibile intuire che il programma presenta delle "anomalie" che vanno sicuramente approfondite, e che la presenza di routine di criptazione potrebbe essere (ma non necessariamente) associata ad un ransomware. Quindi, anche con un'analisi manuale "abbozzata", è subito possibile individuare degli aspetti sospetti.

Veniamo ora al secondo metodo. È abbastanza evidente che questo metodo si propone di inizializzare ed eseguire un **servizio**. Per capire quale viene inizializzato, basta osservare le istruzioni antecedenti:

new-instance v0, Landroid/content/Intent;

invoke-virtual {p0}, Lcom/zynga/FarmVilleTropicEscape/FarmVilleTropicEscape;->getApplicationContext()Landroid/content/Context;

move-result-object v1

const-class v2, Lcom/zynga/**FarmVilleTropicEscape/FarmVvailleTropicEscape**;

invoke-direct {v0, v1, v2}, Landroid/content/Intent;-><init>(Landroid/content/Context;Ljava/lang/Class;)V

Queste quattro istruzioni si possono leggere così: inizializza un **nuovo Intent** (un massaggio da mandare ad un altro componente) su v0, prepara **la sorgente del messaggio** su v1 (ovvero la nostra classe attuale), inizializza la nuova classe **FarmVvailleTropicEscape** (notare anche il nome leggermente distorto) su v2 e inizializza l'intent, che verrà tradotto nella richiesta da parte dell'attività attuale di avviare il servizio.

L'analisi dei due metodi sopra descritti mostra quali sono le prossime classi da esplorare a fondo: **FarmVilleTavropicEscape e FarmVvailleTropicEscape.** Ripetendo il procedimento sopra descritto, è possibile analizzare in maniera approfondita il file classes.dex.

Ovviamente, in molti casi (e anche in questo) non basta l'analisi manuale per avere un quadro completo del funzionamento dell'applicazione, ma è possibile comunque ricorrere a diverse tecniche automatiche di analisi statica e dinamica per semplificare l'analisi. Una panoramica degli strumenti utilizzabili è disponibile su [2]. L'analisi manuale è comunque importante, dato che aiuta l'analista a comprendere più facilmente le zone "critiche" dell'applicazione su cui concentrare la propria analisi.

In conclusione, questo articolo, assieme ai due precedenti, ha voluto riportare solo un "assaggio" della complessità dei ransomware e della disciplina del reverse engineering su Android. Questo mostra anche come la rilevazione di tali attacchi possa essere complessa e ricca di insidie. Per ulteriori approfondimenti, è caldamente consigliato lo studio approfondito della documentazione ufficiale di Android e degli internals Dalvik [1].

Riferimenti:

[1] Dalvik Bytecode. https://source.android.com/devices/tech/dalvik/dalvik-bytecode

[2] Mobile Security Wiki. https://mobilesecuritywiki.com/

A cura di: **Davide Maiorca**



L'Ing. Davide Maiorca (http://pralab.diee.unica.it/it/DavideMaiorca) ha conseguito presso l'Università degli Studi di Cagliari la Laurea Specialistica in Ingegneria Elettronica (con punti 110/110 e lode) nel 2012, ed il Dottorato (PhD) in Ingegneria Elettronica ed Informatica nel 2016. Nel 2017, la sua tesi di dottorato è stata annoverata dal CLUSIT (Associazione Italiana per la Sicurezza Informatica) fra i migliori lavori a livello nazionale nel campo della sicurezza informatica. Lavora per il Pattern Recognition and Applications Lab (PRALab – Dipartimento di Ingegneria Elettrica ed Elettronica, Università degli Studi di Cagliari – Diretto dal prof. Fabio Roli) dal 2012, dove attualmente ricopre il ruolo di post-doc. I suoi campi di ricerca includono l'analisi di malware all'interno di documenti e applicazioni multimediali (PDF, Microsoft Office, Flash), l'analisi di malware Android, e l'Adversarial Machine Learning. Da novembre 2013 ad aprile 2014 è stato Visiting Student presso la Ruhr-Universität Bochum, nel gruppo System Security guidato dal Prof. Dr. Thorsten Holz, dove ha lavorato a tecniche di reverse engineering per applicazioni Android. Dal 2016, è docente del seminario "Mobile Forensics" presso il Dipartimento di Ingegneria Elettrica ed Elettronica, Università degli Studi di Cagliari. È autore di numerose pubblicazioni su conferenze e riviste internazionali, e fa parte di diversi comitati scientifici internazionali. Svolge, inoltre, attività di consulenza tecnica in ambito legale.

https://www.ictsecuritymagazine.com/articoli/viaggio-nei-malware-android-lincubo-ransomware-parte-3/