SECURITATEA SISTEMULUI DE OPERARE ANDROID

LAURA RUSE
VLAD TRAISTĂ-POPESCU

UNIVERSITATEA POLITEHNICA DIN BUCUREȘTI

Cuprins

Listă de figuri

Capitolul 1

Introducere

În prezent, Android-ul este cel mai utilizat sistem de operare pentru dispozitive mobile, ceea ce face cu atât mai mult ca securitatea oferită de sistemul de operare să aibă o importanță ridicată. Securitatea unui sistem este conferită de mecanismele de securitate și de protecție ale fiecărei componente care face parte din acel sistem.

Arhitectura sistemului de operare Android este împărțită în 3 componente principale:

1. Linux-ul nativ care include pe de o parte kernel-ul care oferă o serie de funcționalități specifice Android-ului care poartă numele de Androidisme și, pe de altă parte, userspace-ul nativ în care sunt incluse: procesul init, sute de biblioteci și zeci de servicii native, precum și Hardware Abstraction Layer (HAL) care asigură interfațarea și comunicarea între middleware-ul Android și driverele hardware.

2. Middleware-ul Android conține următoarele componente: 1) Runtime-ul Android Dalvik sau ART după caz. 2) Procesul Zygote care este părintele tuturor aplicațiilor din Android. 3) Framework-ul Android care oferă toate funcționalitățile necesare pentru dezvoltarea de aplicații în ecosistemul Android. 4) Serviciile de sistem care îndeplinesc funcționalități critice precum: display, conectivitatea la rețea, telefonia mobilă, management-ul consumului de putere, etc.

3. Aplicațiile scrise de către dezvoltatori. Acestea pot fi aplicații de tip stock (preinstalate pe telefon) sau alte aplicații (instalate de pe Google Play sau de pe third-party stores).

În Figura 1.1 este reprezentată arhitectura sistemului Android. După cum se poate observa, arhitectura Android-ului are atât un număr semnificativ de componente, cât și un grad ridicat de interconectare între aceste componente. Aceste caracteristici conduc la un sistem foarte complex ceea ce face ca și mecanismele de protecție și de securitate să prezinte un grad similar de complexitate.

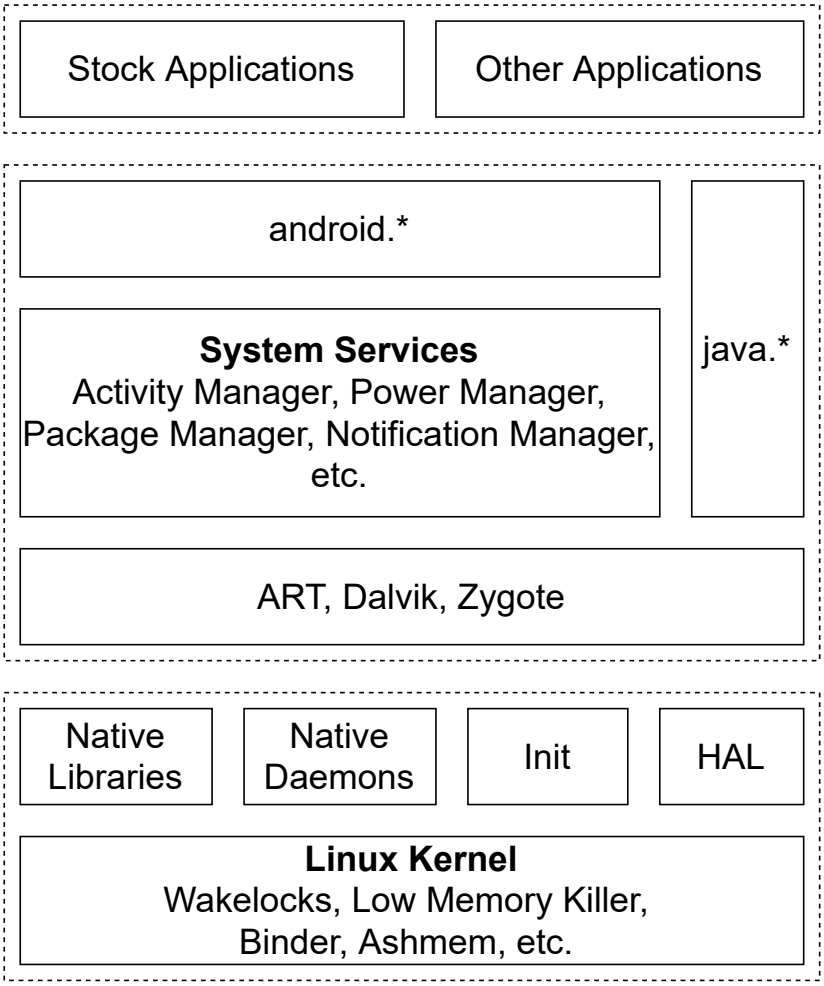

Figura 1.1: Arhitectura Android-ului

Capitolul 2

Mecanisme de securitate

2.1 Obiective de securitate

Se dorește ca Android-ul să fie unul dintre sistemele de operare pentru dispozitive mobile cele mai securizate. Obiectivele de securitate declarate sunt:

- Protejarea aplicațiilor și a datelor utilizatorilor
- Protejarea resurselor de sistem (inclusiv comunicația prin rețea)
- Izolarea aplicațiilor de sistem, de celelalte aplicații și de utilizator

Pentru a îndeplini aceste obiective, Android-ul oferă următoarele mecanisme de securitate:

- Securitate robustă la nivelul sistemului de operare prin kernel-ul Linux
- Sandbox obligatoriu pentru toate aplicațiile
- Semnarea aplicațiilor
- Folosirea permisiunilor
- Comunicație securizată între procese

2.2 Sandbox

Sandboxing-ul pe Android este un mecanism de securitate care are la bază folosirea UID-urilor (user identifier).

Acestea sunt folosite pentru a izola aplicațiile în felul următor. Atunci când o aplicație este instalată, primește un UID unic în sistem (care este identificatorul aplicației). Aplicația va rula într-un proces cu acel UID. În plus, va avea un director dedicat, în care doar acel UID are permisiuni de read-write-execute (rwx).

Astfel se obține un sandbox la nivel de proces și la nivel de fișier. Acest sandbox este implementat la nivelul kernel-ului, folosind mecanismele Unix standard legat de procese, UID și permisiuni pe fișiere.

2.2.1 Director dedicat

Fiecare aplicație are un director de date dedicat (în care poate stoca baza de date, imagini, sau alte fișiere) și permisiuni de read-write-execute (`rwx`) pe acele fișiere, doar pentru UID/GID-ul aplicației. Alte aplicații nu pot citi aceste fișiere deoarece nu vor avea permisiunile necesare (nu exista drepturi pentru others).

Înainte de Android 4.2 se puteau folosi flag-urile `MODE_WORLD_READABLE` și `MODE_WORLD_WRITEABLE` pentru a oferi acces de citire și scriere pe anumite fișiere altor aplicații. Dar aceste flag-uri sunt deprecated din Android 4.2 și nu este recomandată partajarea directă a fișierelor.

2.2.2 UID

Daemon-ii și serviciile de sistem primesc un UID predefinit, care este specificat într-un fișier header numit `android_filesystem_config.h`.

Utilizatorul `root` are UID 0. Prin aplicarea principiului celui mai mic privilegiu, foarte puțini daemoni rulează ca root.

Utilizatorul `system` are UID 1000. Acesta este un utilizator cu privilegii speciale, dar limitate, nu este echivalent cu root. Serviciile de sistem încep de la UID 1000.

Aplicațiile normale au UID-uri începând cu 10000. Acestea sunt asignate dinamic, la instalarea aplicației.

2.2.3 Shared UID

În anumite cazuri speciale, o aplicație poate fi instalată cu același UID ca altă aplicație, reprezentând **Shared UID**. În acest caz, cele două aplicații pot partaja direct fișiere și pot rula în același proces.

Cel mai frecvent, shared UID este folosit de aplicațiile de sistem, pentru a partaja resursele mai ușor (ex. system UI și lockscreen). În general nu este recomandată folosirea shared UID pentru aplicațiile care nu sunt de sistem deoarece ar putea introduce vulnerabilități.

Pentru a folosi shared UID, aplicațiile trebuie semnate cu aceeași cheie și trebuie folosit atributul `sharedUserId` în fișierul Manifest. Acest atribut a devenit deprecated din Android 10 (API 29).

2.3 Permisiuni

Un punct central al arhitecturii de securitate a Android-ului este faptul că nici o aplicație nu are în mod implicit permisiunea de efectua operații care să afecteze alte aplicații sau sistemul. În Android, o permisiune este un string ce semnifică abilitatea de a efectua o anumită operație înafara sandbox-ului.

Android-ul vine cu un set de permisiuni predefinite (built-in). Acestea sunt documentate online pe platform API reference și sunt definite în pachetul `android.*`. În plus, atât aplicațiile de sistem cât și cele instalate de utilizator pot defini permisiuni adiționale, numite custom.

2.3.1 Specificarea permisiunilor în Manifest

Denumirea unei permisiuni include numele pachetului care a definit-o, plus string-ul .permission, plus numele efectiv.

În continuare sunt prezentate 2 exemple. Prima este o permisiune built-in definită în pachetul android, iar a doua este o permisiune custom definită de către aplicația Laucher implicită.

```
android.permission.REBOOT
com.android.launcher3.permission.RECEIVE_LAUNCH_BROADCASTS
```

Aplicațiile vor cere permisiuni prin specificarea lor în fișierul `AndroidManifest.xml`. Urmează un exemplu, se folosește tag-ul `uses-permission`. Tot în Manifest se declară și permisiunile custom.

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

Până la Android 6, permisiunile erau oferite la instalare și nu puteau fi modificate sau revocate mai târziu (doar dezinstalarea aplicației). De la Android 6, permisiunile sunt cerute de la utilizator în timpul rulării. În plus, utilizatorul poate revoca ulterior permisiunile în orice moment.

2.3.2 Gestiunea permisiunilor de către Package Manager

Permisiunile sunt asignate fiecărei aplicații, de către serviciul Package Manager. Acesta gestionează o bază de date centralizată cu informații despre pachetele instalate, stocată în `/data/system/packages.xml`.

Pentru a obține informații despre un pachet instalat, în mod programatic, se va folosi metoda `getPackageInfo()` din `android.content.pm.PackageManager`. Aceasta returnează o instanță `PackageInfo` care încapsulează toate informațiile din fișierul `packages.xml`, despre acel pachet.

O listă a tuturor permisiunilor cunoscute de sistem la un moment dat poate fi obținută folosind `pm list permission` în linia de comandă.

2.3.3 Verificarea permisiunilor

O permisiune poate fi verificată în mai multe locuri pe parcursul rulării unei aplicații:

- Atunci când se face un apel în sistem, pentru a verifica dacă aplicația are dreptul să execute anumite metode.
- Atunci când se pornește o activitate, pentru a verifica ce aplicații pot să lanseze activitatea repectivă.
- Atunci când se pornește sau se leagă (bind) la un serviciu, pentru a verifica ce aplicații pot să folosească serviciul.
- Atunci când se trimit sau se primesc mesaje de broadcast, pentru a controla cine poate primi mesajul generat de aplicație sau cine poate trimite un mesaj aplicației.
- Atunci când se accesează un content provider (și un anumit URI, pentru citire, scriere), pentru a verifica ce aplicații pot să folosească content provider-ul.

2.3.4 Nivelul de protecție al permisiunilor

Nivelul de protecție al unei permisiuni indică riscul potențial al acelei permisiuni, dar și modul în care sistemul își dă seama dacă să acorde acea permisiune aplicației în cauză.

În Android avem 4 niveluri de protecție: **normal**, **dangerous**, **signature** și **signatureOrSystem**.

Nivelul de protecție **normal** definește o permisiune cu un risc scăzut pentru sistem sau alte aplicații. Permisiunile cu acest nivel de protecție sunt acordate automat aplicațiilor, fără să fie nevoie de confirmarea utilizatorului. Exemple de permisiuni cu nivelul normal: `ACCESS_NETWORK_STATE`, `GET_ACCOUNTS`.

Permisiunile cu nivelul de protecție **dangerous** oferă aplicației acces la datele utilizatorului sau o formă de control asupra dispozitivului. Android-ul va afișa o fereastră cu informații despre permisiunile cerute de aplicație la instalare sau la runtime (în funcție de versiunea de Android). Exemple de permisiuni cu nivelul dangerous: `CAMERA`, `READ_SMS`.

Permisiunile **signature** sunt acordate aplicațiilor doar dacă sunt semnate cu aceeași cheie cu care a fost semnată aplicația care a declarat (creat) acea permisiune. Acesta este cel mai înalt nivel de protecție. Permisiunile built-in sunt folosite doar de către aplicațiile de sistem care sunt semnate cu cheia de platformă. Exemple de permisiuni cu nivelul signature: `NET_ADMIN`, `ACCESS_ALL_EXTERNAL_STORAGE`.

Permisiunile **signatureOrSystem** sunt un compromis: acestea sunt acordate aplicațiilor care sunt parte din imaginea de sistem sau sunt semnate cu cu aceeași cheie cu care a fost semnată aplicația care a declarat acea permisiune. Asta permite vendor-ilor să aibă aplicații preinstalate fără să folosească cheia de platformă.

2.3.5 Grupuri de permisiuni

Permisiunile pot face parte din grupuri de permisiuni. Grupurile de permisiuni dangerous sunt cele care pot afecta experiența utilizatorului. Exemple de grupuri de permisiuni dangerous: Calendar, Camera, Contacts, Location, Phone, SMS, Sensors, Storage, Microphone.

Până în Android 6, sistemul întreba utilizatorul dacă oferă permisiunile aplicației la momentul instalării. Dar întreba pentru tot grupul de permisiuni, nu pentru una individuală.

De la Android 6, dacă o aplicație are nevoie de o anumită permisiune dangerous și aplicația are deja primită o permisiune din același grup, utilizatorul nu va mai fi întrebat, ci sistemul va oferi automat acea permisiune. Dacă nu are nici o permisiune din grupul respectiv, va fi întrebat utilizatorul dacă acceptă acel grup de permisiuni, însă doar permisiunea din acel grup va fi acceptată.

2.3.6 Verificarea permisiunilor la nivel de kernel

Accesul la fișierele normale, la dispozitive și la socket-uri este gestionat ca în orice sistem Linux, de către kernel, pe baza UID și GID.

Permisiunile sunt asociate cu anumite GID-uri suplimentare care sunt verificate atunci când se fac operații de nivel scăzut. Mapările pentru permisiunile built-in se găsesc în fișierul `/etc/permission/platform.xml`.

De exemplu, permisiunea `INTERNET` are mapat GID-ul numit `inet`. O aplicație nu poate crea socket-uri de rețea dacă nu deține permisiunea `INTERNET`. Atunci când aplicația vrea să creeze un socket, kernel-ul va verifica dacă face parte din grupul `inet`.

2.3.7 Verificarea statică a permisiunilor

La nivelul framework-ului avem două tipuri de enforcement (verificare) al permisiunilor: static și dinamic.

În modul static, sistemul ține evidența permisiunilor asociate cu fiecare componentă și verifică dacă procesele care apelează au permisiunile necesare înainte de a permite accesul. Prin urmare, enforcement-ul static este realizat de către runtime environment.

Avantajul este că se separă deciziile de securitate față de logica de business. Dezavantajul constă în faptul că această metodă este mai puțin flexibilă.

Enforcement-ul static se face atunci când o aplicație încearcă să interacționeze cu o componentă declarată de către altă aplicație, printr-un intent. Pentru ca verificarea să fie făcută cu succes, componenta destinație trebuie să aibă atributul `android:permission` în Manifest, iar aplicația apelantă trebuie să aibă tagul `<uses-permission>` în Manifest, amândouă referindu-se la aceeași permisiune.

Verificarea este făcută de către Activity Manager, care rezolvă intent-ul și verifică dacă componenta destinație are asociată o permisiune. Dacă da, atunci delegă verificarea permisiunii către Package Manager. Dacă aplicația apelantă are permisiunea necesară atunci se va porni componenta destinație. Altfel, se va genera o excepție de securitate (`SecurityException`).

2.3.8 Verificarea dinamică a permisiunilor

În mod dinamic, componentele pot de asemenea să verifice dacă procesul apelant are permisiunile necesare, fără a le declara în Manifest. Deci în modul dinamic verificările sunt făcute de fiecare componentă în loc să fie făcute de către runtime environment.

Avantajul este că avem un control al accesului mai granular iar dezavantajul este că facem mai multe operații în componente.

Android-ul oferă o serie de metode pentru a verifica permisiunile în clasa `android.content.Context`. Două exemple de astfel de metode sunt `checkPermission` și `enforcePermission`.

`checkPermission(String permission, int pid, int uid)` va returna `PERMISSION_GRANTED` dacă procesul cu acel uid are permisiunea necesară și `PERMISSION_DENIED` altfel.

Dacă procesul apelant este root sau system, atunci permisiunea este acordată automat. Dacă permisiunea este declarată de către aplicația apelantă atunci permisiunea este acordată fără alte verificări. Dacă componenta destinație este privată atunci, accesul

este interzis. Altfel, se va apela Package Manager pentru a afla dacă procesul apelant are permisiunea necesară sau nu.

O altă metodă este `enforcePermission(String permission, int pid, int uid, String message)`, care efectuează operații asemănătoare cu cea precedentă, dar va genera o excepție de securitate cu mesajul specificat dacă permisiunea nu este acordată.

2.3.9 Verificarea permisiunilor pentru componentele aplicației

Verificarea permisiunilor pentru activități se face atunci când un intent este dat ca parametru metodei `startActivity()` sau `startActivityForResult()` care se va rezolva într-o activitate ce declară o permisiune.

Verificarea permisiunilor pentru servicii se face atunci când un intent este dat ca parametru metodei `startService()` sau `stopService()` sau `bindService()` care se va rezolva într-un serviciu care declară o permisiune. Dacă procesul apelant nu are acea permisiune, se va genera o excepție de securitate.

Permisiunile content provider-ilor protejează întreaga componentă sau un anumit URI exportat.

Se folosesc permisiuni diferite pentru citire si pentru scriere. Mai exact, permisiunea de citire controlează cine are voie să apeleze `ContentResolver.query()` pe un provider sau URI. Permisiunea de scriere controlează cine are voie să apeleze `ContentResolver.insert()`, `ContentResolver.update()` și `ContentResolver.delete()` pe un provider sau URI.

Verificările pentru content providers au loc în mod sincron atunci când una dintre metodele respective este apelată.

Atunci când se trimite un broadcast, poate fi necesar ca receiver-ii să aibă o anumită permisiune pentru a primi broadcast-ul. Sender-ul specifică permisiunea prin `Context.sendBroadcast(Intent intent, String receiverPermission)`. Verificarea se face atunci când se livrează broadcast-ul. Dacă receiver-ul nu deține permisiunea, el nu va primi mesajul, dar nu va fi aruncată o excepție de securitate.

De asemenea, ar putea fi necesar ca sender-ul să aibă o anumită permisiune pentru a trimite un mesaj de broadcast. Această permisiune va fi verificată tot atunci când se face livrarea, dar nu va arunca o excepție de securitate în cazul în care permisiunea lipsește.

Prin urmare, se pot face două verificări de permisiuni pentru fiecare livrare de broadcast: una pentru sender și una pentru receiver.

2.3.10 Declararea permisiunilor în fișierul Manifest

Pentru toate versiunile de Android, aplicația trebuie să declare permisiunile în fișierul Manifest. În continuare este prezentat un exemplu de declarație în Manifest: aplicația solicită permisiunea de a trimite SMS-uri.

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
        package="com.example.smd">
```

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

    <application ...>
        ...
    </application>
</manifest>
```

2.3.11 Solicitarea permisiunilor în timpul rulării

În timp ce permisiunile normale sunt oferite în mod automat de către sistem, permisiunile dangerous trebuie oferite în mod explicit de către utilizator. Atunci când facem o anumită operație trebuie verificat dacă aplicația are permisiunea de tip dangerous. Acest lucru trebuie făcut de fiecare dată deoarece permisiunile pot fi revocate începând cu Android 6, și este posibil ca între timp permisiunea să fie revocată de către utilizator.

Verificarea se face folosind metoda `checkSelfPermission()` care va returna `PERMISSION_GRANTED` dacă aplicația a primit acea permisiune, și atunci poate să facă operația. Dacă returnează `PERMISSION_DENIED` însemnă că aplicația nu a primit acea permisiune și trebuie cerută de la utilizator în mod explicit.

Cererea permisiunii se face atunci când `checkSelfPermission()` întoarce `PERMISSION_DENIED`. Cererea efectivă se face folosind metoda `requestPermissions()` care primește ca parametri un vector de permisiuni și un cod de cerere.

Chiar dacă utilizatorul acceptă întregul grup de permisiuni, aplicația trebuie să ceară fiecare permisiune în parte (prima oară va fi întrebat utilizatorul și următoarele sunt oferite automat de către sistem).

Atunci când este făcută cererea, se va afișa un dialog box utilizatorului, pentru a cere întregul grup de permisiuni (de exemplu: Contacts). Acest dialog box nu poate fi modificat de către aplicație, deoarece este generat de către sistem. De aceea, este indicat ca aplicația să dea o explicație separată înainte de a cere permisiunea.

Cererea este asincronă, răspunsul nu este primit pe loc, ci va fi primit într-un callback al activității.

Urmează un exemplu de verificare și cerere a permisiunii.

```
if (ContextCompat.checkSelfPermission(thisActivity,
        Manifest.permission.READ_CONTACTS)
    != PackageManager.PERMISSION_GRANTED) {

    // Permisiunea nu a fost oferită aplicaţiei

    ActivityCompat.requestPermissions(thisActivity,
                new String[]{Manifest.permission.READ_CONTACTS},
                MY_PERMISSIONS_REQUEST_READ_CONTACTS);

    // Permisiunea READ_CONTACTS este solicitată utilizatorului.
    // Rezultatele sunt primite în callback

} else {
    // Permisiunea a fost deja oferită aplicaţiei
}
```

Atunci când utilizatorul răspunde (acceptă sau nu permisiunea), sistemul va apela callback-ul `onRequestPermissionsResult()`. Aplicația trebuie să suprascrie acest callback pentru a afla răspunsul.

Metoda primește ca argumente codul cererii, permisiunile și răspunsurile pentru fiecare permisiune. În metodă trebuie verificat întâi codul permisiunii. Și apoi trebuie verificat dacă permisiunea a fost oferită sau nu de utilizator.

Dacă permisiunea a fost oferită atunci aplicația poate efectua operația asociată cu permisiunea respectivă. Dacă permisiunea a fost respinsă, atunci aplicația trebuie să dezactiveze funcționalitatea asociată și eventual să anunțe utilizatorul că nu poate să efectueze operația.

În continuare este prezentat un exemplu de implementare a metodei callback.

```
@Override
public void onRequestPermissionsResult(int requestCode,
        String permissions[], int[] grantResults) {
    switch (requestCode) {
        case MY_PERMISSIONS_REQUEST_READ_CONTACTS: {
            if (grantResults.length > 0
                && grantResults[0] == PackageManager.PERMISSION_GRANTED)
                    {
                // permisiunea a fost oferită
                // se poate face operaţia
            } else {
                // permisiunea a fost respinsă
                // trebuie dezactivată funcţionalitatea
            }
            return;
        }
    }
}
```

2.3.12 Permisiuni custom

Permisiunile custom sunt cele declarate de către aplicațiile third-party. După ce sunt declarate, pot fi adăugate în componentele aplicațiilor pentru a fi verificate în mod static de către sistem, sau aplicația însăși poate verifica dacă aplicația apelantă are permisiunea respectivă.

O permisiune custom este declarată în fișierul Manifest. În continuare este prezentat un exemplu.

```
<permission-tree
        android:name="com.example.app.permission"
        android:label="@string/example_permission_tree_label" />

<permission-group
        android:name="com.example.app.permission-group.TEST_GROUP"
        android:label="@string/test_permission_group_label"
        android:description="@string/test_permission_group_desc" />

<permission
        android:name="com.example.app.permission.PERMISSION1"
        android:label="@string/permission1_label"
        android:description="@string/permission1_desc"
```

```
android:permissionGroup="com.example.app.permission-group.
   TEST_GROUP"
android:protectionLevel="signature" />
```

Se crează întâi un arbore de permisiuni. Apoi se crează un grup de permisiuni, și în final, permisiunea care va aparține acelui grup. Pentru permisiune se setează și nivelul de protecție, care în acest caz este signature. Asta înseamnă ca doar aplicațiile semnate cu aceeași cheie ca aplicația care declară permisiunea pot primi această permisiune.

2.4 Semnarea aplicațiilor

Semnarea aplicațiilor în Android are rolul de a identifica dezvoltatorul. Toate aplicațiile ce rulează pe Android trebuie să fie semnate. Aplicațiile ne-semnate vor fi respinse de către Google Play și de către package installer pe dispozitivele mobile.

Pe Google Play, semnătura este o punte de legătură între încrederea pe care Google o are în dezvoltator și încrederea pe care dezvoltatorul o are în aplicație. Dezvoltatorul cunoaște bine aplicația deci este responsabil de comportamentul acesteia.

2.4.1 Certificat

Fiecare pachet aplicație (APK) trebuie să fie semnat cu un certificat care este generat folosind cheia privată a dezvoltatorului. Acest certificat identifică în mod unic dezvoltatorul aplicației și are rolul de a distinge între doi dezvoltatori de aplicații diferiți. Certificatul nu trebuie să fie semnat de către o autoritate de certificare, ci poate fi self-signed.

Actualizarea unei aplicații este permisă doar dacă noua versiune este semnată folosind aceeași cheie. Astfel, un atacator nu va putea forța instalarea unei noi versiuni deoarece nu are cheia originală.

Atunci când un APK este instalat pe dispozitiv, Package Manager va verifica semnătura. Va face asta folosind cheia publică din certificat care se găsește în apk. Astfel este garantat că aplicația nu a fost modificată între timp.

Aplicațiile de sistem sunt semnate folosind o cheie de platformă, care este cheia utilizată pentru a semna AOSP-ul (Android Open Source Project) la compilare.

2.4.2 Semnarea aplicațiilor cu aceeași cheie

Putem avea aplicații semnate cu aceeași cheie. Atunci când se întâmplă asta, e posibil ca cele două aplicații să fie configurate în fișierul Manifest să partajeze același UID, folosind mecanismul SharedUID. Totuși, acest mecanism este deprecated începând cu Android 10 (API 29).

Mai există și permisiunile cu nivelul de protecție Signature, care sunt garantate unei aplicații doar dacă aceasta este semnată cu aceeași cheie ca aplicația care a creat aceea permisiune. În acest caz, cele două aplicații vor rula în sandbox-uri diferite și vor avea UID-uri diferite.

2.4.3 Scheme de semnare

Android-ul folosește două tipuri de scheme de semnare, una bazată pe semnăturile arhivelor JAR (versiunea 1), și alta numită APK Signature Scheme v2. Schema v2 a fost introdusă în Android 7.

Pentru ca aplicația să aibă compatibilitate maximă, trebuie semnată folosind ambele scheme. Atunci când aplicația este instalată pe un dispozitiv cu Android 7 (sau versiune mai mare), se va verifica semnătura creată cu schema v2. În caz contrar, va ignora semnătura v2 și va verifica doar folosind schema v1.

Semnăturile v1 nu protejează anumite părți din APK, cum ar fi metadatele ZIP. Verificatorul APK trebuie să proceseze multe date și trebuie să le facă discard la cele care nu sunt acoperite de semnătură. Asta oferă atacatorilor o suprafață considerabilă de atac. În plus, verificatorul trebuie să decomprime toate intrările comprimate și asta consumă timp și memorie.

Din Android 7 a fost introdusă schema v2. Această schemă este mai rapidă și detectează mai multe tipuri de modificări neautorizate decât schema v1.

Întregul conținut al arhivei APK este hash-uită și semnată, rezultând APK Signing Block. Acesta este inserat în APK. În timpul verificării semnăturii, APK-ul este tratat ca un blob, deci se face verificarea semnăturii pe întreg fișierul. Astfel, orice modificare, incluzând metadatele ZIP, vor invalida semnătura.

2.5 Bibliografie

- Nikolay Elenkov. 2014. Android Security Internals: An In-Depth Guide to Android's Security Architecture. No Starch Press.
- https://source.android.com/security/ (Accesat: Aprilie 2021)
- https://source.android.com/security/apksigning/ (Accesat: Aprilie 2021)
- https://developer.android.com/guide/topics/permissions/overview.html (Accesat: Aprilie 2021)

Capitolul 3

Securitatea rețelei

3.1 Provideri Criptografici

3.1.1 Arhitectura JCA

Java Cryptography Architecture (JCA) oferă un framework extensibil pentru providerii criptografici și un set de API-uri pentru accesarea diferitelor primitive criptografice.

Aplicațiile care folosesc JCA trebuie doar să ceară un anumit algoritm, fără a fi nevoie să folosească direct un provider specific. Framework-ul se va ocupa să găsească providerul care oferă acel algoritm.

Cryptographic Service Provider (CSP) este un pachet ce include implementarea unui set de servicii sau algoritmi criptografici. Fiecare provider va anunța către JCA ce servicii și algoritmi implementează.

JCA va gestiona un registru cu providerii și algoritmii implementați de aceștia. În acest registru, providerii vor fi ordonați în ordinea preferinței. Dacă doi provideri implementează același algoritm, va fi selectat cel cu preferința cea mai mare (număr de ordine cel mai mic).

Service Provider Interface (SPI) este o interfață comună care trebuie respectată de toți providerii care implementează un anumit algoritm. Mai exact este o clasă abstractă care este implementată de către providerii ce oferă acel algoritm.

3.1.2 Clasele JCA Engine

JCA Engines oferă unul dintre următoarele servicii:

- Operații criptografice (criptare, decriptare, semnare, verificare semnătura, hash)
- Generare sau conversie a materialelor criptografice (chei sau parametrii algoritmilor)
- Gestionare și stocare a obiectelor criptografice (chei, semnături digitale)

Clasele engine decuplează codul clientului de implementarea algoritmilor, de aceea nu pot fi instanțiate în mod direct. În schimb ele oferă o metodă statică de tip factory numită

`getInstance()`. Prin această metodă se va cere implementarea unui serviciu în mod indirect.

În continuare sunt prezentate 3 formate ale lui `getInstance()`.

```
static EngineClassName getInstance(String algorithm)
        throws NoSuchAlgorithmException
static EngineClassName getInstance(String algorithm, String provider)
        throws NoSuchAlgorithmException, NoSuchProviderException
static EngineClassName getInstance(String algorithm, Provider provider)
        throws NoSuchAlgorithmException
```

Cel mai utilizat este primul, în care se specifică doar numele algoritmului iar JCA va identifica providerul cu cea mai mare prioritate care oferă acel algoritm. În al doilea format, se cere și un anumit provider specificandu-se numele providerului în format string. Al treilea are ca argument o instanță a provider-ului dorit.

Toate pot arunca `NoSuchAlgorithmException` dacă nu se găsește algoritmul, iar a doua mai poate arunca `NoSuchProviderException`, dacă nu găsește providerul respectiv.

3.1.3 Clasa MessageDigest

Clasa `MessageDigest` este folosită pentru a obține o funcție hash. Urmează un exemplu de folosire:

```
MessageDigest md = MessageDigest.getInstance("SHA-256");
byte[] data = getMessage();
byte[] hash = md.digest(data);
```

Întâi se obține o instanță a clasei `MessageDigest` specificând algoritmul SHA-256 atunci când apelăm metoda factory `getInstance()`. Apoi pentru datele reprezentate ca un array de bytes aplicăm metoda `digest()` și obținem hash-ul.

Există două metode prin care se pot da datele: 1) pentru date de dimensiuni mari, se pot da bucăți folosind metoda `update()`, iar la final se apelează `digest()` fără nici un argument; 2) pentru date de dimensiuni mici sau de dimensiune fixă se poate apela direct `digest()` cu datele ca argument.

3.1.4 Clasa Signature

Clasa `Signature` oferă o interfață pentru algoritmii de semnături digitale bazați pe criptare asimetrică. Numele unui algoritm este de obicei <digest>with<encryption>, unde digest este numele unui algoritm de hashing iar encryption este numele un algoritm de criptare asimetric.

În continuare este prezentat un exemplu de creare a unei semnături digitale.

```
byte[] data = "message to be signed".getBytes("ASCII");

Signature s = Signature.getInstance("SHA256withRSA");
s.initSign(privKey);
s.update(data);
byte[] signature = s.sign();
```

Întâi se obține instanța cu `getInstance()` si numele algoritmului SHA256withRSA. Apoi se setează cheia privată cu care se va realiza semnătura prin metoda `initSign()`. Apoi se dau datele prin metoda `update()`. În final se realizează semnătura prin metoda `sign()`.

Urmează un exemplu de verificare a semnăturii digitale.

```
Signature s = Signature.getInstance("SHA256withRSA");
s.initVerify(pubKey);
s.update(data);
boolean valid = s.verify(signature);
```

Întâi se obține instanța, apoi se configurează cheia publică cu care se va verifica semnătura prin metoda `initVerify()`. Apoi se dau datele cu metoda `update()`. În final se verifică semnătura cu metoda `verify()`.

3.1.5 Clasa Cipher

Clasa `Cipher` oferă o interfață comună pentru criptare și decriptare. Pentru criptare este prezentat următorul exemplu.

```
Secret key = getSecretKey();

Cipher c = Cipher.getInstance("AES/CBC/PKCS5Padding");

byte[] iv = new byte[c.getBlockSize()];
SecureRandom sr = new SecureRandom();
sr.nextBytes(iv);
IvParameterSpec ivp = new IvParameterSpec(iv);
c.init(Cipher.ENCRYPT_MODE, key, ivp);

byte[] data = "Message to encrypt".getBytes("UTF-8");
byte[] ciphertext = c.doFinal(data);
```

În primul rând se obține o instanță `Cipher` care folosește algoritmul de criptare AES, modul de operare CBC și padding de tipul PKCS#5. Apoi se generează un Initialization Vector (IV) random și se pune într-un obiect de tipul IvParameterSpec. Apoi se inițializează Cipher pentru criptare folosind metoda `init()` cu flag-ul `ENCRYPT_MODE`, o cheie de criptare și un IV.

Ciphertext-ul se obține prin apelarea metodei `doFinal()` asupra plaintext-ului.

Pentru decriptare este prezentat următorul exemplu.

```
Cipher c = Cipher.getInstance("AES/CBC/PKCS5Padding");
c.init(Cipher.DECRYPT_MODE, key, ivp);

byte[] data = c.doFinal(ciphertext);
```

Pentru decriptare, se obține instanța `Cipher` și se inițializează Cipher-ul folosind `init()` cu flag-ul `DECRYPT_MODE`, cheia și IV-ul primit.

Plaintext-ul se obține prin apelarea metodei `doFinal()` asupra ciphertext-ului.

3.1.6 Clasa Mac

Clasa `Mac` oferă o interfață comună pentru algoritmii de tip Message Authentication Code. Urmează un exemplu de utilizare a clasei `Mac`.

```
SecretKey key = getSecretKey();
Mac m = Mac.getInstance("HmacSha256");
m.init(key);
byte[] data = "Message".getBytes("UTF-8");
byte[] hmac = m.doFinal(data);
```

În primul rând se obține o instanță `Mac` folosind metoda `getInstance()` în care se cere implementarea algoritmului HMAC care folosește SHA-256 ca funcție hash. Apoi este inițializată cu metoda `init()` și cheia secretă.

Se apelează metoda `doFinal()` pentru a obține valoarea MAC-ului pe baza datelor. De asemenea se pot da bucăți de date prin metoda `update()` și în final se apelează `doFinal()`.

3.1.7 Clasa KeyGenerator

Clasa `KeyGenerator` este folosită pentru a genera chei simetrice (folosite pentru algoritmi de criptare simetrici sau algoritmi MAC). Este mai bună decât `SecureRandom` pentru că se realizează verificări adiționale pentru chei slabe din punct de vedere criptografic și poate seta biți de paritate atunci când este necesar (de exemplu pentru algoritmul DES). De asemenea, poate folosi hardware criptografic dacă este disponibil.

În continuare sunt prezentate două exemple de utilizare.

```
KeyGenerator kg = KeyGenerator.getInstance("HmacSha256");
SecretKey key = kg.generateKey();
```

În primul exemplu se obține o instanță `KeyGenerator` pentru algoritmul HmacSha256 și apoi se generează cheia cu metoda `generateKey()`. Se va obține direct o cheie pe 256 biți pentru acel algoritm.

```
KeyGenerator kg = KeyGenerator.getInstance("AES");
kg.init(256);
SecretKey key = kg.generateKey();
```

În al doilea exemplu, se obține o instanță `KeyGenerator` pentru algoritmul AES. Apoi se dă dimensiunea cheii - 256, deoarece AES poate suporta 3 dimensiuni de chei. În final se generează cheia folosind metoda `generateKey()`.

3.1.8 Clasa KeyPairGenerator

Clasa `KeyPairGenerator` este folosită pentru a genera perechi de chei publice și private (pentru algoritmi de criptare asimetrici).

Urmează un exemplu de utilizare.

```
KeyPairGenerator kpg = KeyPairGenerator.getInstance("RSA");
kpg.initialize(1024);
KeyPair pair = kpg.generateKeyPair();
```

```
PrivateKey priv = pair.getPrivate();
PublicKey pub = pair.getPublic();
```

Se obține instanța `KeyPairGenerator` pentru algoritmul RSA. Se dă lungimea cheilor prin metoda `initialize()`. Apoi se obține un `KeyPair` folosind metoda `generateKeyPair()`. În final se obțin cheile cu metodele `getPrivate()` (cheia privată) și `getPublic()` (cheia publică).

3.1.9 Provideri JCA în Android

În această secțiune sunt prezentați câțiva provideri care se pot folosi pe Android.

Harmony's Crypto Provider este un provider JCA cu un set restrâns de funcționalități care este inclus în biblioteca de runtime Java. Include doar 4 algoritmi, pentru SecureRandom, KeyFactory, MessageDigest și Signature.

Android's Bouncy Castle Provider este un provider JCA cu un set foarte mare de algoritmi și servicii, care face parte din Bouncy Castle Crypto API. Avem mulți algoritmi pentru fiecare clasă în parte: Cipher, KeyGenerator, Mac, MessageDigest, SecretKeyFactory, Signature, CerticateFactory, etc.

AndroidOpenSSL Provider este implementat în cod nativ din motive de performanță. Are destul de multe funcționalitătii, acoperind marea majoritate a algoritmilor și serviciilor oferite de Bouncy Castle. Practic implementarea folosește JNI pentru a accesa biblioteca OpenSSL. Acest provider este cel preferat în mod implicit, are prioritatea cea mai mare, numărul 1.

3.2 SSL/TLS

Android-ul oferă provideri criptografici pentru calculul hash-urilor, MAC-urilor, pentru criptare, etc. Aceștia pot fi folosiți pentru asigurarea comunicației securizate.

Totuși, pentru a evita introducerea unor vulnerabilități, este mai bine să fie folosite protocoale de securitate standardizate, care au fost specificate și testate îndelung. Cele mai folosite protocoale pentru asigurarea comunicației securizate prin rețea sunt TLS și predecesorul său SSL.

Acestea sunt protocoale pentru comunicația securizată punct-la-punct, care oferă autentificare, confidențialitatea și integritatea mesajelor trimise între două entități care comunică peste TCP/IP. Ele folosesc o combinație între criptarea simetrică și asimetrică pentru a asigura confidențialitate și integritate și se bazează pe certificate pentru autentificare.

3.2.1 Cipher Suite

De obicei clientul care dorește să comunice cu un server prin SSL, inițiază comunicația prin trimiterea versiunii de SSL suportate și a unei liste de cipher suites.

Un cipher suite este un set de algoritmi folosiți pentru calculul cheii, autentificare, protecția integrității și criptare. Clientul și serverul vor negocia un cipher suite care este suportat de amândoi.

3.2.2 Autentificare și criptare

După aceea, se vor autentifica (vor verifica identitatea celuilalt) prin certificate. În general, doar serverul se autentifică la client. Totuși, SSL permite și autentificarea clienților.

Dacă autentificarea are loc cu succes, ei calculează o cheie simetrică partajată care va fi folosită pentru securizarea comunicației. În continuare, comunicația va fi securizată folosind algoritmul de criptare și cheia negociată.

3.2.3 Certificate bazate pe chei publice

Un certificat bazat pe cheie publică este folosit pentru asocierea unei identități cu acea cheie publică. SSL folosește certificate X.509 pentru autentificare.

Aceste certificate includ un număr mare de câmpuri: algoritmul de semnare, entitatea care l-a generat (issuer), validitatea, subiectul, etc. Un subiect este reprezentat de un set de atribute incluzând common name (CN), locația și organizația. Issuer-ul este descris prin niște atribute similare.

În Figura 3.1 se poate observa o parte din certificatul Google.

3.2.4 Încredere directă

Dacă clientul SSL comunică cu un număr mic de servere, poate fi configurat cu un set de certificate de încredere, numite și trust anchors. Aceste certificate pot fi self-signed. Un server este de încredere dacă certificatul lui face parte din acest set.

Avantajul este că putem avea control strict asupra serverelor de încredere. Dezavantajul este că e mai greu de actualizat cheia serverului și certificatul, trebuie modificat/reconfigurat clientul.

3.2.5 CA-uri private

O altă opțiune este folosirea unei autorități de certificare (CA) private pentru a semna certificatul serverului. Acest CA privat este folosit ca trust anchor. Clientul va avea încredere în orice certificat care este generat de acest CA.

Avantajul este că se poate actualiza ușor cheia serverului și certificatul, fără actualizarea clientului. Dezavantajul este faptul că acest CA devine un sigle point of failure. Dacă CA-ul este compromis, atunci atacatorul poate genera certificate în care clienții vor avea încredere în mod automat.

3.2.6 CA-uri publice

Un client SSL (browser web, client de mail) care nu știe în avans care vor fi serverele cu care va comunica, este de obicei configurat cu un set de trust anchors, care sunt CA-uri bine cunoscute, publice.

Majoritatea browser-elor web includ un set de mai mult de 100 de CA-uri (certificatele lor) ca trust anchors.

Figura 3.1: Certificat Google

3.3 JSSE

Android-ul suportă SSL/TLS prin implementarea Java Secure Sockets Extension (JSSE). API-ul JSSE este disponibil prin pachetele `javax.net` și `javax.net.ssl`. Acesta oferă:

- `SSLSocket` și `SSLServerSocket` - sockets SSL de tip client și server
- `SSLSocketFactory` și `SSLServerSocketFactory` - pentru generarea socket-ilor SSL
- `SSLEngine` - care produce și consumă stream-uri SSL
- `SSLContext` - un context de sockets secure, care este folosit pentru a crea factories și engines.
- `KeyManager` și `KeyManagerFactory` - Manageri de chei și factories pentru a-i crea
- `TrustManager` și `TrustManagerFactory` - Manageri de încredere și factories pentru a-i crea

- `HttpsURLConnection` - pentru conexiuni HTTPS

În Figura 3.2 sunt reprezentate clasele JSSE și interacțiunea între ele.

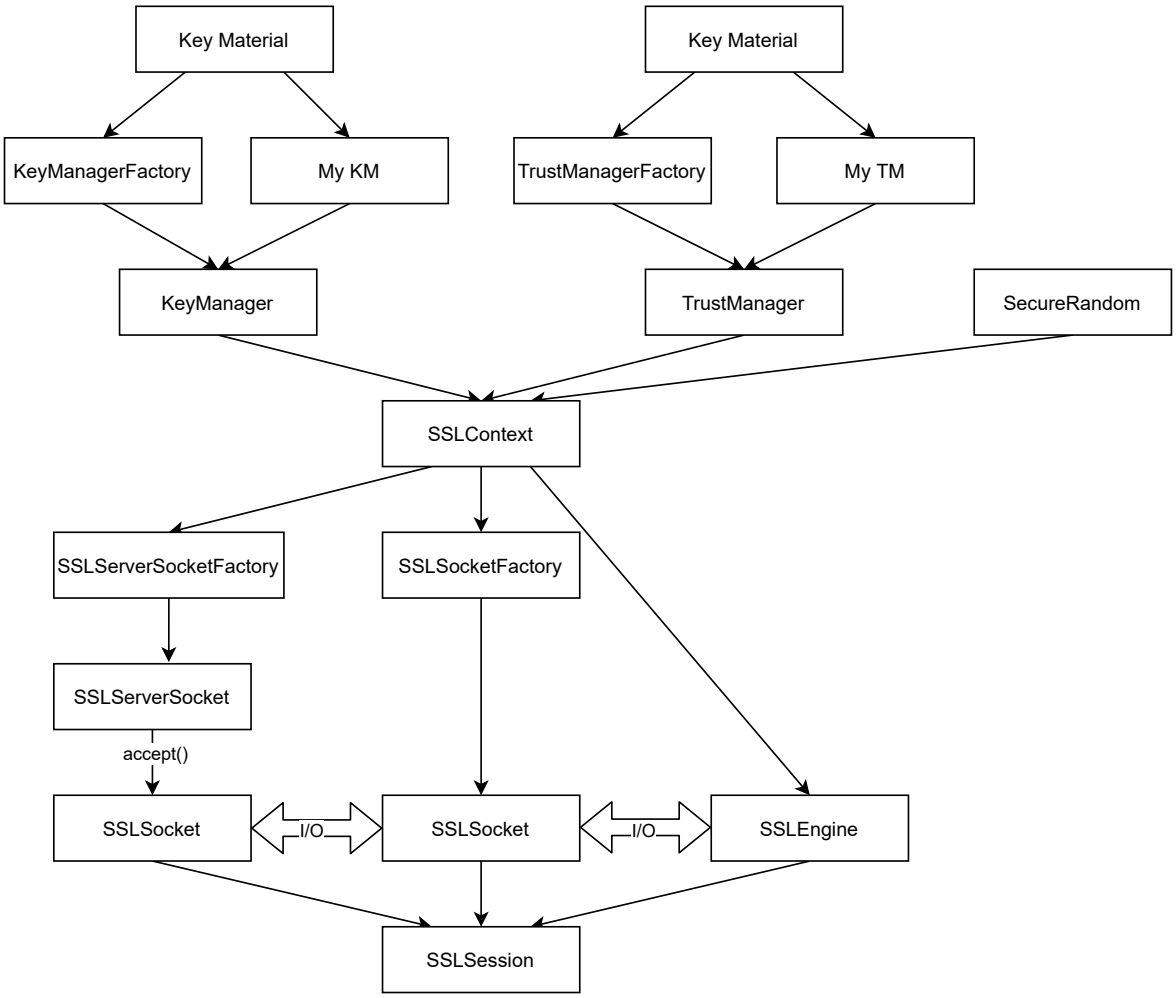

Figura 3.2: Clasele JSSE

În JSSE, clasele care reprezintă capetele unei conexiuni sunt `SSLSocket` și `SSLEngine`. Figura arată care sunt clasele principale folosite pentru crearea `SSLSocket` și `SSLEngine`.

3.3.1 SSLSocket

Un `SSLSocket` este creat prin `SSLSocketFactory`, sau prin acceptarea unei conexiuni pe un `SSLServerSocket`. Un `SSLServerSocket` este creat prin `SSLServerSocketFactory`.

Un `SSLEngine` este creat direct prin `SSLContext` și se bazează pe faptul că aplicația poate gestiona operațiile I/O.

3.3.2 SSLContext

Un `SSLContext` este obținut în două moduri: Prin apelarea metodei `getDefault()` a `SSLServerSocketFactory` sau `SSLSocketFactory` - acesta oferă un context implicit inițializat cu `KeyManager`-ul implicit, `TrustManager`-ul implicit și un generator

de numere aleatoare. Key material din keystore-ul și truststore-ul implicit sunt obținute din proprietățile de sistem.

Se poate obține o instață de `SSLContext` și prin apelarea metodei statice `getInstance()` a `SSLContext`. Apoi contextul este inițializat prin următorii parametri: un vector de obiecte `KeyManager`, un vector de obiecte `TrustManager` și un `SecureRandom`. Obiectele `KeyManager` și `TrustManager` pot fi obținute prin `KeyManagerFactory` și `TrustManagerFactory`. Aceste factories pot fi inițializate cu un `KeyStore` care conține key material.

3.3.3 SSLSession

După ce o conexiune a fost creată între un client și un server SSL, se crează un obiect `SSLSession`. `SSLSession` include informații de genul: identitățile capetelor conexiunii, cipher suite-ul negociat, etc.

Fiecare conexiune SSL are asociat un `SSLSession`, iar un `SSLSession` poate fi folosit pentru mai multe conexiuni între aceleași entități.

3.3.4 TrustManager și KeyManager

JSSE va delega deciziile legate de încrederea în certificate clasei `TrustManager` iar selecția cheilor pentru autentificare clasei `KeyManager`. Fiecare instanță `SSLSocket` creată prin JSSE va avea acces la aceste clase prin instanța `SSLContext` asociată.

`TrustManager` are un set de certificate de încredere generate de autorități de certificare, și va lua deciziile pe baza lui. Dacă un certificat este emis de către o autoritate de certificare de încredere atunci acel certificat va fi considerat de încredere.

3.3.5 System Trust Store

`TrustManager`-ul implicit din JSSE folosește trust store-ul de sistem, care include un set de certificate de CA comerciale și guvernamentale. Trust store-ul de sistem se găsește pe Android în `/system/etc/security/cacerts.bks`.

Urmează un exemplu în care sunt obținute toate certificatele din trust store-ul de sistem.

```
TrustManagerFactory tmf = TrustManagerFactory
        .getInstance(TrustManagerFactory.getDefaultAlgorithm());
tmf.init((KeyStore) null);

X509TrustManager xtm = (X509TrustManager) tmf
        .getTrustManagers()[0];

for (X509Certificate cert : xtm.getAcceptedIssuers()) {
    String certStr = "S:" + cert.getSubjectDN().getName()
                + "\nI:" + cert.getIssuerDN().getName();
    Log.d(TAG, certStr);
}
```

Întâi se obține o instanță de `TrustManagerFactory` care se inițializează cu argumentul `null`, și astfel va lua automat trust store-ul implicit, de sistem. Apoi se obține primul `TrustManager`, cel implicit.

Apoi prin metoda `getAcceptedIssuers()` se obține lista certificatelor din trust store-ul implicit. Iar pentru fiecare certificat în parte sunt afișate informații.

Până la Android 4.0, trust store-ul de sistem se afla într-un singur fișier `/system/etc/security/cacerts.bks`. Totuși, partiția de sistem este read-only, deci fișierul nu putea fi modificat (nici de către aplicațiile de sistem).

De la Android 4.0, în plus față de acest fișier avem două directoare adiționale: `/data/misc/keychain/cacerts-added` și `/data/misc/keychain/cacerts-removed`. Primul include certificate CA are au fost adăugate trust store-ului de sistem, și a doua include certificate CA care au fost scoase din trust store-ul de sistem.

Doar utilizatorul system poate adăuga sau scoate certificate CA din trust store-ul de sistem. Se pot adăuga trust anchors prin clasa `TrustCertificateStore`, care este accesibilă prin JCA KeyStore API.

3.3.6 Validarea certificatului server-ului

În continuare este prezentat un exemplu de validare manuală a unui certificat de server folosind trust store-ul de sistem.

```
TrustManagerFactory tmf = TrustManagerFactory
                                .getInstance("X509");
tmf.init((KeyStore) null);

TrustManager[] tms = tmf.getTrustManagers();
X509TrustManager xtm = (X509TrustManager) tms[0];

X509Certificate[] certChain = {serverCert};
xtm.checkServerTrusted(certChain, "RSA");
```

În primul rând se obține o instanță a lui `TrustManagerFactory`. Se inițializează cu trust store-ul de sistem prin folosirea metodei `init()` cu argumentul `null`.

Apoi se obține vectorul de `TrustManagers`. Se obține primul `TrustManager` și se face cast la `X509TrustManager`.

Se construiește un lanț de certificate incluzând certificatul severului și alți issuers intermediari. În final, se validează lanțul de certificate folosind metoda `checkServerTrusted()` a obiectului `X509TrustManager`.

`SSLSocket` și `HttpsURLConnection` efectuează această validare în mod automat.

3.3.7 HttpsURLConnection

`HttpsURLConnection` este metoda recomandată pentru conectarea la un server HTTPS. Folosește `SSLSocketFactory`-ul implicit pentru a crea socketi SSL.

Atunci când este nevoie de un trust store custom sau chei de autentificare, `SSLSocketFactory`-ul implicit poate fi înlocuit prin metoda statică `setDefaultSSLSocketFactory()` sau `setSSLSocketFactory()` a clasei `HttpsURLConnection`.

3.3.8 Folosirea unui Trust Store propriu

Dacă se dorește folosirea unui trust store propriu, trebuie parcurși următorii pași: întâi se încarcă trust store-ul (ce conține trust anchors) într-un obiect `KeyStore`.

Apoi se obține o instanță a lui `TrustManagerFactory` care se inițializează cu acel trust store.

Dacă este nevoie de autentificarea clientului, se încarcă key material în obiectul `KeyStore`. Se obține o instanță de `KeyManagerFactory` și se inițializează cu acel `KeyStore`.

Apoi se obține o instanță `SSLContext` pentru TLS și se inițializează cu `TrustManager` și `KeyManager`.

Se crează un obiect `URL` și `HttpsURLConnection` bazat pe acel URL. Se asociază `SSLSocketFactory` (a lui `SSLContext`) la `HttpsURLConnection`.

```
KeyStore trustStore = loadTrustStore();
KeyStore keyStore = loadKeyStore();

TrustManagerFactory tmf = TrustManagerFactory
     .getInstance(TrustManagerFactory.getDefaultAlgorithm());
tmf.init(trustStore);

KeyManagerFactory kmf = KeyManagerFactory
     .getInstance(KeyManagerFactory.getDefaultAlgorithm());
kmf.init(keyStore, KEYSTORE_PASSWORD.toCharArray());

SSLContext sslCtx = SSLContext.getInstance("TLS");
sslCtx.init(kmf.getKeyManagers(), tmf.getTrustManagers(), null);

URL url = new URL("https://myserver.com");
HttpsURLConnection urlConnection = (HttpsURLConnection) url
urlConnection.setSSLSocketFactory(sslCtx.getSocketFactory());
```

În continuare este prezentat un exemplu mai specific, în care se încarcă propriul trust store.

```
KeyStore localTrustStore = KeyStore.getInstance("BKS");
InputStream in = getResources().openRawResource(
                R.raw.mytruststore);
localTrustStore.load(in, TRUSTSTORE_PASSWORD.toCharArray());

TrustManagerFactory tmf = TrustManagerFactory
        .getInstance(TrustManagerFactory.getDefaultAlgorithm());
tmf.init(localTrustStore);

SSLContext sslCtx = SSLContext.getInstance("TLS");
sslCtx.init(null, tmf.getTrustManagers(), null);

URL url = new URL("https://myserver.com");
HttpsURLConnection urlConnection =
                (HttpsURLConnection) url.openConnection();
urlConnection.setSSLSocketFactory(sslCtx.getSocketFactory());
```

Se poate genera trust store-ul în linia de comandă folosind Bouncy Castle și openSSL. Apoi se plasează trust store-ul în `res/raw`.

În primul rând se obține o instanță de `KeyStore` în format Bouncy Castle, în care se încărcă trust store-ul (va trebui specificată parola). Apoi se obține o instanță de `TrustManagerFactory` și se inițializează cu trust store-ul.

Se obține o instanță de `SSLContext` și se inițializează cu `TrustManager`-ul obținut din factory. Se observă că `KeyManager` este null deoarece nu se dorește autentificarea clientului în acest exemplu.

Apoi se obține obiectul `URL`, `HtttpsURLConnection` și se asociază `SSLSocketFactory` din context la conexiune.

3.3.9 Provideri JSSE în Android

În mod similar cu providerii criptografici JCA, există provideri JSSE care implementează funcționalitatea pentru clase engine definite de API-ul descris anterior.

Funcționalitatea implementată de provideri include: secure sockets, trust managers, key managers, etc. Aplicațiile nu vor lucra în mod direct cu implementarea claselor ci cu clasele engine.

În Android, există doi provideri JSSE: Harmony JSSE care implementat în Java, și AndroidOpenSSL, care este implementat în cod nativ, și accesat prin JNI.

3.3.9.1 HarmonyJSSE

HarmonyJSSE este bazat pe socketi Java și folosește clasele criptografice JCA pentru implementarea SSL. Oferă suport doar pentru SSLv3 și TLSv1. Este considerat deprecated și nu este dezvoltat în mod activ.

3.3.9.2 AndroidOpenSSL

AndroidOpenSSL implementează majoritatea funcționalității prin apeluri către biblioteca nativă OpenSSL (prin JNI). Oferă suport pentru TLSv1.1 și TLSv1.2.

De asemenea oferă suport pentru extensia de TLS numită Server Name Indication (SNI) - ceea ce însemnă că clienții SSL pot specifica un hostname, în cazul în care serverul are mai multe hostname-uri virtuale. SNI este folosit în mod implicit atunci când se face o conexiune cu `HttpsURLConnection`.

Amândoi providerii partajează același cod pentru `TrustManager` și `KeyManager`, dar implementarea de sockets este diferită.

3.4 Bibliografie

- Nikolay Elenkov. 2014. Android Security Internals: An In-Depth Guide to Android's Security Architecture. No Starch Press.
- http://nelenkov.blogspot.ro/2011/12/using-custom-certificate-trust-store-on.html (Accesat: Mai 2021)
- https://github.com/nelenkov/custom-cert-https (Accesat: Mai 2021)

- http://docs.oracle.com/javase/7/docs/technotes/guides/security/jsse/JSSERefGuide.html (Accesat: Mai 2021)

Capitolul 4

Bootloader, verified boot, root access

4.1 Bootloader

Un bootloader este un program specializat, proprietar și specific pentru hardware-ul pe care este executat (pentru un anumit System on Chip - SoC).

Acesta rulează atunci când dispozitivul este pornit (pe dispozitivele ARM asta înseamnă atunci când dispozitivul iese din reset). Are scopul de a inițializa hardware-ul, apoi de a găsi și a porni sistemul de operare.

Boot-area include de obicei mai multe etape și există un bootloader pentru fiecare etapă. Dar în această carte ne vom referi la un singur bootloader care include toate etapele.

4.1.1 Mod Fastboot/Download

Majoritatea bootload-erelor oferă un mod numit Fastboot sau Download care permite scrierea (flashing) partițiilor raw în spațiul de stocare permanent al dispozitivului, dar și bootarea unor imagini de sistem temporare (fără scrierea lor pe dispozitiv).

Modul fastboot este activat de o combinație specială de taste hardware în timpul bootării, dar și prin trimiterea comenzii `adb reboot bootloader`.

4.1.2 Bootloader blocat

Dispozitivele de pe piață vin cu un bootloader blocat (locked) pentru a asigura integritatea imaginilor ce rulează pe ele.

Atunci când bootloader-ul este blocat, nu se va putea scrie sau boota alte imagini de sistem. În cel mai bun caz se vor putea scrie imagini de sistem care au fost semnate de către producătorul dispozitivului.

Cele mai multe dispozitive vor permite deblocarea bootloader-ului, ceea ce dezactivează restricțiile fastboot și verificările de semnătura a imaginilor.

De obicei, deblocarea bootloader-ului necesită formatarea partiției userdata, pentru a nu permite unui sistem de operare malițios să aibă acces la datele curente ale utilizatorului.

4.1.3 Deblocarea bootloader-ului folosind Fastboot

Multe dispozitive vor permite deblocarea bootloader-ului folosind fastboot. Conectarea la dispozitiv se face folosind un cablu USB, ceea ce va permite trimiterea comenzilor prin intermediul utilitarelor `adb` și `fastboot`.

Primul pas este repornirea dispozitivului în modul fastboot, ori folosind comanda `adb reboot bootloader`, ori apăsând o combinație de taste în timpul boot-ării.

Pentru deblocarea bootloader-ului se va da comanda `fastboot oem unlock`, care va afișa o cerere de confirmare pe ecranul dispozitivului.

Mesajul afișat va anunța că deblocarea bootloader-ului va permite instalarea unor versiuni de Android netestate/nesigure, și faptul că vor fi șterse toate datele utilizatorului (va fi formatată partiția userdata, din motive de securitate).

Bootloader-ul poate fi blocat din nou prin comanda `fastboot oem lock`. Ceea ce va duce la revenirea bootloader-ului la starea inițială, în care scrierea sau boot-area imaginilor custom nu este permisă.

Pe lângă flag-ul de locked/unlocked, unele bootloader-e mai au un flag adițional numit “tampered” care va fi setat atunci când bootloader-ul a fost deblocat prima oară. Astfel vor putea fi interzise anumite operații după ce acest flag a fost setat, sau va fi afișat un warning.

4.1.4 Deblocarea bootloader-ului din setări

Pe unele dispozitive este suficientă activarea unei opțiuni din setări pentru a debloca bootloader-ul. Întâi se va activa Developer Options prin apăsarea unui număr de ori pe Build number, apoi, din Developer Options, se va activa “OEM unlocking”.

4.2 Recovery OS

O metodă mai flexibilă de a actualiza un dispozitiv este prin Recovery OS. Acesta este un sistem de operare minimal bazat pe Linux, care include un kernel, un RAM disk cu diferite utilitare de nivel scăzut și un UI (user interface) minimal. Este stocat pe partiția de recovery și de obicei folosit pentru a aplica pachete Over-the-air (OTA).

Pachetele OTA includ de obicei noi versiuni (patch-uri binare) ale unor fișiere de sistem și un script care aplică modificările. Fișierele din OTA sunt semnate folosind cheia privată a producătorului.

Imaginea de recovery include și cheia publică pentru verificarea fișierelor OTA înainte de a face modificările. Acest lucru ne garantează faptul că fișierele OTA provin dintr-o sursă de încredere.

Se poate intra în modul recovery prin folosirea comenzii `adb reboot recovery` sau prin apăsarea unei combinații de taste în timpul boot-ării.

Recovery OS poate fi stock sau custom. Recovery stock este oferit de producător și vine deja instalat pe dispozitivul mobil. Un recovery custom este dezvoltat de către o entitate third-party și poate fi instalat pe telefon doar dacă se deblochează bootloader-ul.

4.2.1 Recovery stock

Un recovery stock (de la producător) oferă o funcționalitate minimală, cu scopul de a actualiza software-ul de sistem, fără a șterge datele utilizatorului.

Recovery-ul stock oferă un UI minimal, simplu, sub forma unui meniu, care este folosit prin intermediul butoanelor fizice (de obicei butoanele Power, Volume Up și Down).

Meniul include opțiunile: reboot, apply update from ADB, factory reset și wipe cache partition. Opțiunea apply update from ADB va permite actualizarea prin comanda `adb sideload` (tethered).

4.2.2 Recovery custom

Imaginea de recovery este scrisă pe partiția de recovery și poate fi suprascrisă atunci când dispozitivul este în modul fastboot/download.

Atunci când se scrie o imagine de recovery custom, este posibilă dezactivarea verificării semnăturilor OTA. Acest lucru va putea permite modificarea sistemului de operare principal. De asemenea, un recovery custom poate permite accesul la root prin ADB. Mai poate permite citirea și salvarea datelor de pe partițiile existente.

Un recovery custom este creat de o entitate third party (nu de către producătorul dispozitivului). Acesta nu va fi semnat cu cheia producătorului. De aceea, bootloader-ul va trebui deblocat pentru a scrie sau boota acest recovery custom.

Pentru a boota un recovery custom putem folosi comanda `fastboot boot recovery.img`. Iar pentru a scrie pe dispozitiv, comanda `fastboot flash recovery recovery.img`.

Un recovery custom oferă funcționalități avansate, care nu sunt disponibile într-un recovery stock:

- Backup și restaurare a partițiilor întregi.
- Shell root cu acces la utilitare de gestiunea dispozitivului
- Suport pentru montarea dispozitivelor USB externe
- Dezactivarea verificării semnăturii pachetelor OTA (ceea ce permite modificarea sistemului de operare sau instalarea unor build-uri custom).

4.2.3 TWRP

Recovery-ul custom care are cele mai multe funcționalități și este activ dezvoltat este Team Win Recovery Project (TWRP).

Acesta este:

- Bazat pe recovery-ul stock AOSP și continuă să fie open source.
- Oferă o interacțiune pe bază de touch screen.
- Oferă suport pentru backup-ul partițiilor criptate
- Poate instala actualizări de sistem de pe dispozitive USB

- Poate face backup și restaurare pe/de pe dispozitive externe
- Include un manager de fișiere
- Oferă un limbaj de scripting pentru a descrie acțiunile ce trebuie executate la boot-area în modul recovery.

4.2.4 Atac folosind recovery custom

Dacă partiția de date este criptată, atunci nu se pot accesa datele în mod direct dintr-un custom recovery.

Dar se poate instala un rootkit (un program malițios) pe partiția system din custom recovery. Rootkit-ul va permite accesul remote la dispozitiv în timp ce acesta este în sistemul de operare principal. Astfel se vor putea accesa datele necriptate (ele sunt decriptate în mod transparent de către sistemul de operare la bootare).

Mecanismul verified boot poate preveni acest atac dacă este verificată partiția de boot folosind o cheie nealterabilă, stocată în hardware. Astfel, mecanismele de securitate verified boot și criptarea disk-ului vor putea limita daunele făcute de o imagine de sistem malițioasă (scrisă atunci când bootloader-ul este deblocat)

4.3 Verified Boot

4.3.1 Device Mapper

Mecanismul **verified boot** este bazat pe device mapper. Acesta este un framework din kernel-ul Linux, care oferă un mod generic de implementare a dispozitivelor bloc virtuale.

Acest framework este baza pentru Logical Volume Manager (LVM) în Linux și este folosit pentru a implementa criptarea întregului disc (dm-crypt), RAID arrays și storage replicat distribuit.

Device mapper funcționează prin maparea unui dispozitiv bloc virtual peste unul sau mai multe dispozitive fizice, și prin modificarea datelor în tranzit.

4.3.2 dm-verity

Mecanismul verified boot din Android este bazat pe dm-verity. Dm-verity este un target de device mapper pentru verificarea integrității blocurilor. Asta înseamnă că verifică integritatea fiecărui bloc atunci când este citit de pe disc.

Dacă verificarea se realizează cu succes atunci blocul de date este citit. Dacă nu, atunci citirea se va întoarce cu o eroare I/O (ca și cum blocul a fost corupt fizic).

Dm-verity este implementat folosind un arbore Merkle care include toate hash-urile blocurilor de pe dispozitiv.

Nodurile frunză ale arborelui includ hash-urile blocurilor fizice, iar nodurile intermediare sunt hash-uri ale nodurilor copil (hash-uri de hash-uri). Nodul rădăcină (numit și hash-ul rădăcină) este bazat pe toate hash-urile de la nivelurile inferioare.

O singură modificare asupra unui bloc va duce la modificarea hash-ului rădăcină. Prin urmare, pentru a verifica dacă un arbore hash este nemodificat, e nevoie doar de verificarea hash-ului rădăcină.

În timpul rulării, dm-verity calculează hash-ul fiecărui bloc atunci când este citit și îl verifică prin traversarea arborelui de hash-uri precalculat.

Citirea datelor de pe dispozitivul fizic este deja o operație ce consumă timp, iar latența introdusă de hashing și verificare este neglijabilă. Odată verificat, un bloc este cache-uit și la citirile ulterioare ale aceluiași bloc nu se va mai face nicio verificare de integritate.

Dm-verity depinde de arborele de hash-uri precalculat ale tuturor blocurilor unui dispozitiv, de aceea dispozitivul trebuie să fie montat read-only pentru ca verificarea să fie posibilă.

Montarea unui dispozitiv read-write va duce automat la eșuarea verificării de integritate. Chiar dacă fișierele nu sunt modificate la rulare, se vor modifica anumite metadate din superbloc.

De aceea mecanismul dm-verity funcționează bine cu partiția system, deoarece aceasta este modificată doar prin actualizarea sistemului de operare.

Orice altă modificare va însemna coruperea sistemului de operare sau a disk-ului, sau poate indica faptul că un program malițios încearcă să modifice sistemul de operare (un fișier de sistem).

Dm-verity se potrivește bine cu modelul de securitate al Android-ului, care folosește o partiție read-write doar pentru datele aplicațiilor, și stochează fișierele sistemului de operare pe partiția system care este montată read-only.

4.3.3 Android dm-verity

Din Android 4.4 este folosit target-ul dm-verity, dar mecanismul este implementat un pic diferit față de cel din kernel-ul de Linux. Mai exact: verificarea hash-ului rădăcină și montarea partițiilor verificate.

Cheia publică RSA folosită pentru verificare este stocată pe partiția de boot (fișierul `verity_key`), și este folosit pentru a verifica tabela de mapare dm-verity care include: locația dispozitivului, offset-ul tabelei de hash-uri, hash-ul rădăcină și salt-ul.

Verity metadata block include tabela de mapare și semnătura, și este scris pe disc imediat după ultimul bloc al sistemului de fișiere (Figura 4.1). Partiția este marcată verificabilă prin adăugarea unui flag `verify` în fișierul `fstab`.

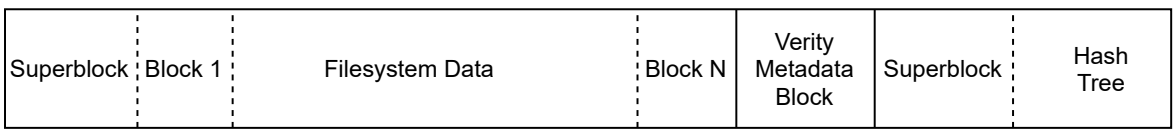

Figura 4.1: Conținutul partiției cu dm-verity activat

4.3.4 Procesul de verificare

Atunci când managerul sistemului de fișiere întâlnește acest flag `verify`, încarcă metadatele verity de pe dispozitiv și verifică semnătura folosind cheia verity.

Dacă verificarea semnăturii se face cu succes, managerul sistemului de fișiere parsează tabela de mapare dm-verity și o pasează mai departe la device mapper-ul din Linux. Device mapper-ul va folosi informațiile din tabela de mapare pentru a crea dispozitivul bloc virtual dm-verity. Acest dispozitiv bloc virtual este montat în punctul de montare specificat în fstab, în locul dispozitivului fizic.

Astfel, toate citirile de bloc de pe dispozitivul fizic vor fi verificate în mod transparent folosind arborele de hash-uri precalculat. Dacă sunt modificate sau adăugate fișiere, sau remontată partiția ca read-write, se va face o verificare de integritate care va eșua și va fi generată o eroare I/O.

4.3.5 Partiția boot de încredere

Pentru a asigura protecția integrității, trebuie să avem încredere în kernelul care conține dm-verity. Prin urmare partiția de boot trebuie să fie de încredere.

Pe Android, acest lucru necesită verificarea partiției de boot, care conține kernelul, RAM disk-ul și cheia verity.

Această verificare depinde de acel dispozitiv și este implementată de obicei în bootloader. Verificarea se face folosind o cheie nealterabilă de verificare a semnăturii care este stocată în hardware.

4.3.6 Activarea verified boot

Procedura pentru activarea verified boot pe Android include următorii pași:

- Generarea arborelui de hash-uri
- Crearea tabelului de mapare dm-verity
- Semnarea tabelului de mapare dm-verity
- Generarea și scrierea blocului de metadate verity pe dispozitiv

4.3.7 Generarea arborelui de hash-uri

Arborele de hash-uri dm-verity este generat folosind programul `veritysetup`, care este parte din cryptsetup. Acesta este un pachet de utilitare de gestiune a discurilor.

Programul `veritysetup` poate acționa direct asupra dispozitivelor bloc și genera un arbore de hash-uri pe baza unei imagini de sistem de fișiere, și poate scrie tabela hash într-un fișier.

Arborele trebuie scris pe același dispozitiv, de aceea, atunci când se apelează `veritysetup`, trebuie specificat offset-ul - o locație după blocul de metadate verity.

Pașii parcurși pentru generarea arborelui sunt:

1. Alegerea unui salt random.
2. Împărțirea imaginii de sistem în blocuri de 4k.
3. Pentru fiecare bloc se calculează un hash SHA256 (se ia salt-ul în calcul).
4. Se formează nivelul din concatenarea hash-urilor în blocuri de 4k.

5. Se face padding cu 0 pentru a ajunge la un multiplu de 4k.
6. Se adaugă nivelul în arbore.

Se repetă pașii 2-6 pe baza nivelului tocmai generat, până când se obține un singur hash (rădăcină).

Figura 4.2 este o ilustrare a arborelui de hash-uri.

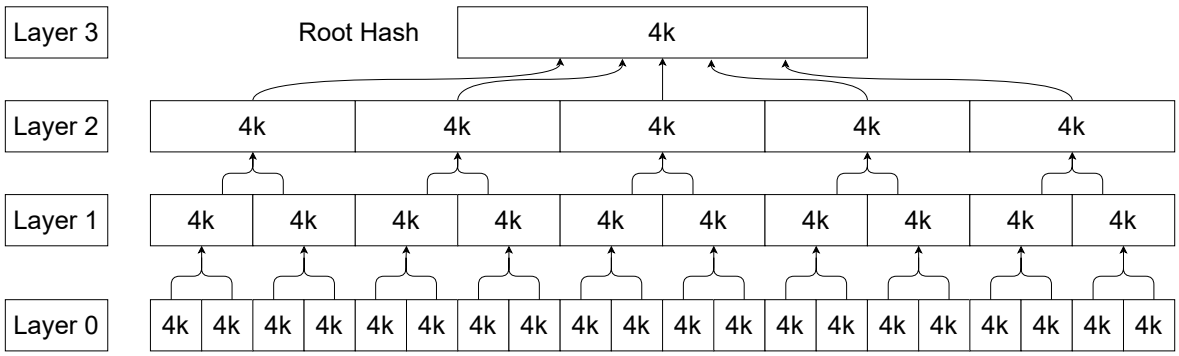

Figura 4.2: Hash Tree

4.3.8 Crearea tabelei de mapare

Atunci când se generează arborele de hash-uri, se generează hash-ul rădăcină. Acesta este folosit pentru a crea tabela de mapare dm-verity pentru acel dispozitiv.

Tabela de mapare conține versiunea de dm-verity, denumirea dispozitivului ce stochează datele și hash-urile, dimensiunile blocurilor de date și hash-uri, locația pe disk a datelor și a arborelui de hash-uri, algoritmul de hash-ing, hash-ul rădăcină și salt-ul.

4.3.9 Semnarea tabelei de mapare

Tabela de mapare este semnată folosind o cheie RSA pe 2048 biți, în format mincrypt. Aceasta este o bibliotecă criptografică minimalistă folosită și pentru verificarea semnăturilor OTA. Formatul mincrypt este o serializare a structurii RSAPublickey.

Cheia se găsește pe partiția de boot, în fișierul `/verity_key`. Semnătura rezultată este în format PKCS#1 v1.5. Blocul de metadate verity are 32 KB și include tabela de mapare plus semnătura.

4.3.10 Blocul de date verity

Mai exact blocul de metadate verity include:

- Magic number - folosit de managerul sistemului de fișiere pentru verificarea consistenței tabelei (4B)
- Versiunea - deoarece blocul se poate extinde pentru a permite diferite modificări (4B)
- Semnătura tabelei în format PKCS1.5 în forma padded (256B)
- Lungimea tabelului (4B)

- Tabelul în sine
- Padding de zerouri până la 32k

4.3.11 Fișierul fstab

Ultimul pas din procesul de activare al verified boot, este modificarea fișierului `fstab` pentru a activa verificarea integrității blocurilor pentru partiția system.

Pentru aceasta este nevoie doar de adăugarea flag-ului `verify`.

În continuare este prezentat un exemplu de fișier `fstab` de pe un dispozitiv Pixel XL.

```
marlin:/ $ cat /vendor/etc/fstab.marlin
# Android fstab file.
#<src> <mnt_point> <type> <mnt_flags and options> <fs_mgr_flags>
/dev/block/platform/soc/624000.ufshc/by-name/system /system ext4 ro,
    barrier=1 wait,slotselect,verify
```

Atunci când dispozitivul boot-ează, Android-ul crează automat dispozitivul virtual dm-verity pe baza intrării din `fstab` și a informației din tabela de mapare (conținută în blocul de metadate). Dispozitivul virtual va fi montat ca `/system`, în locul dispozitivului fizic.

4.3.12 Eroarea de verificare a integrității

Orice modificare ulterioară a partiției system va cauza o eroare de verificare a integrității. Aplicarea unui pachet OTA care modifică blocurile fără a modifica metadatele verity, va invalida arborele de hash-uri.

Un OTA compatibil cu verified boot trebuie să opereze la nivel bloc și să actualizeze atât blocurile de date cât și arborele de hash-uri și metadatele verity.

4.3.13 Verified boot

Verified boot verifică integritatea dispozitivului, de la rădăcina de încredere hardware până la partiția de sistem. La fiecare etapă a boot-ării, se verifică integritatea și autenticitatea următoarei etape, înainte de a se executa.

Starea boot-ării se referă la nivelul de protecție oferit utilizatorului atunci când dispozitivul boot-ează. Avem 4 stări: GREEN, YELLOW, ORANGE și RED.

Starea dispozitivului se referă la posibilitatea de a rescrie imaginile (flashing). Stările posibile sunt: LOCKED și UNLOCKED.

4.3.14 Cheile de verificare

Integritatea bootloader-ului este verificată folosind rădăcina de încredere hardware.

Partițiile boot și recovery sunt verificate folosind cheia OEM (de la producător). Întotdeauna încearcă să verifice partiția de boot cu această cheie înainte de a încerca alte chei (dacă verificarea nu are loc cu succes).

Dacă dispozitivul este LOCKED, se încearcă întâi cheia OEM, și apoi se încearcă verificarea folosind certificatul integrat în semnătura partiției. Dacă dispozitivul este UNLOCKED atunci este posibil ca utilizatorul să scrie imagini semnate cu alte chei.

4.3.15 Stări de boot-are

Un dispozitiv verificat va boota într-una dintre următoarele 4 stări:

- GREEN - a fost verificat întregul lanț de încredere, de la bootloader, la partiția de boot și alte partiții verificate (system).
- YELLOW - partiția boot a fost verificată folosind certificatul integrat și semnătura este validă. Bootloader-ul va afișa un warning și fingerprint-ul cheii publice înainte de a permite continuarea boot-ării.
- ORANGE - dispozitivul nu a fost verificat și poate fi modificat în mod liber. Bootloader-ul afișează un warning înainte de a continua boot-area.
- RED - verificarea dispozitivului a eșuat. Bootloader-ul va afișa un warning și va opri boot-area.

4.3.16 Stări ale dispozitivului

Există două stări posibile ale dispozitivului:

- LOCKED - dispozitivul nu poate fi rescris. Un dispozitiv LOCKED poate boota doar în stările GREEN, YELLOW, sau RED.
- UNLOCKED - Imaginile dispozitivului pot fi rescrise în mod liber. Dispozitivul nu va fi verificat. Va boota întotdeauna în starea ORANGE.

4.3.17 Procesul de verificare a boot-ării

Figura 4.3 reprezintă o schema cu verificarea boot-ării și cele 4 stări de boot-are.

4.4 Actualizări de sistem

Un recovery stock de obicei implementează atât actualizări OTA, cât și actualizări tethered.

În cazul actualizărilor OTA, sistemul de operare principal va downloada fișierul cu actualizarea și va instrui recovery-ul să o aplice.

În cazul actualizărilor tethered, utilizatorul va downloada fișierul cu actualizarea pe stația lui, și o va trimite la recovery folosind comanda `adb sideload`.

Procesul efectiv de actualizare este identic în ambele cazuri. Deci diferă doar modul de obținere a fișierului cu actualizarea (arhiva OTA).

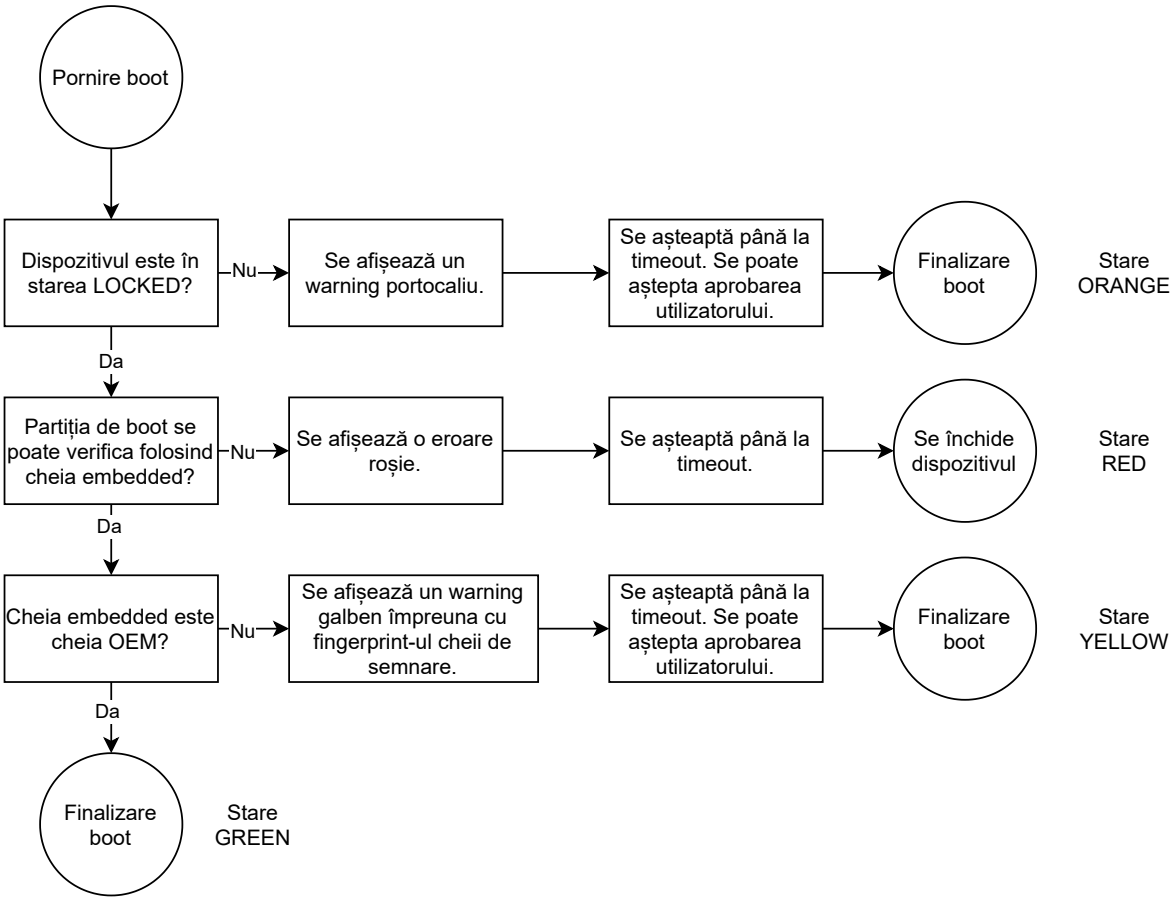

Figura 4.3: Verificarea boot-ării

4.4.1 Controlul operațiilor recovery

Sistemul de operare principal poate controla recovery-ul prin API-ul oferit de `android.os.RecoverySystem`. Acest API va comunica cu recovery-ul prin scrierea unor comenzi (string-uri) în fișierul `/cache/recovery/command`. Conținutul acestui fișier este citit automat de către procesul `/sbin/recovery` atunci când dispozitivul boot-ează în modul recovery.

Câteva opțiuni ce pot fi folosite:

- -update-package = verifică și instalează pachetul OTA
- -send-intent = după ce este finalizată acțiunea din recovery, la reboot-area în sistemul de operare principal, se va trimite acest Intent
- -wipe-data = șterge partițiile userdata și cache, apoi reboot-ează dispozitivul
- -wipe-cache = șterge patiția cache, apoi reboot-ează dispozitivul

4.4.2 Descărcarea pachetului OTA

Dispozitivul mobil va verifica periodic serverele OTA pentru actualizări. Atunci când o nouă actualizare este disponibilă, va obține URL-ul pachetului OTA și o descriere care va fi afișată utilizatorului.

Apoi va downloada pachetul pe partiția cache sau data. Va verifica semnătura și va

întreba utilizatorul dacă este de acord să instaleze acea actualizare.

4.4.3 Verificarea semnăturii OTA

Pachetele OTA sunt code-signed, cu o semnătură aplicată peste întreg fișierul (arhiva). Nu este semnat fiecare fișier în parte.

Atunci când procesul de actualizare este pornit din sistemul principal, după ce s-a download-at pachetul OTA, se va verifica folosind metoda `verifyPackage()` din clasa `RecoverySystem`. Metoda primește pachetul OTA și o arhivă cu certificate X.509.

Dacă la apelarea metodei de verificare nu este specificată o arhivă se va lua setul implicit de certificate din sistem, mai exact `/system/etc/security/otacerts.zip`. Dacă pachetul OTA este semnat cu o cheie privată corespunzătoare unuia dintre aceste certificate, atunci pachetul e considerat valid și sistemul va reboota în modul recovery pentru a aplica actualizarea.

Independent de verificarea anterioară, recovery OS va verifica pachetul OTA (ca să se asigure că nu a fost modificat între timp). Verificarea se face folosind un set de chei publice integrate în imaginea de recovery.

Atunci când se compilează o imagine de recovery, cheile publice sunt extrase dintr-un set de certificate de semnare OTA, convertite în format mincrypt și scrise în fișierul `/res/keys`.

Algoritmii de criptare folosiți sunt: 2048-bit RSA cu SHA-1 (versiunile 1 și 2 de chei) sau SHA-256 (versiunile 3 și 4 de chei), ECDSA cu SHA-256 (versiunea 5 de cheie), EC folosind chei de 256 biți.

4.4.4 Actualizarea partiției recovery

Datele din pachetul OTA sunt folosite pentru a actualiza partițiile boot, system și eventual vendor, după cum este necesar. Fișierul ce conține noua partiție recovery este salvat pe partiția system (deoarece nu poate fi actualizată în timp ce rulează).

Apoi dispozitivul boot-ează normal, partiția boot este încărcată, aceasta încarcă partiția system și rulează executabilele necesare. Este comparat conținutul partiției recovery cu conținutul fișierului salvat pe partiția system. Dacă sunt diferite, atunci conținutul fișierului este scris pe partiția system.

4.4.5 Metoda de actualizare A/B

Cea mai nouă metodă de actualizare a sistemului se numește A/B și folosește câte două seturi de partiții, numite slot-uri.

Această metodă va asigura un sistem funcțional și boot-abil în timpul actualizării OTA. Acest lucru va reduce șansa de a obține un sistem inutilizabil după aplicarea actualizării.

Actualizarea de tip A/B se aplică în timp ce sistemul rulează normal, și utilizatorul folosește dispozitivul. Nu este nevoie de restartare decât după aplicarea actualizării, pentru a boota pe partiția actualizată. Dar boot-area nu durează mai mult decât de obicei.

Metoda de actualizare A/B folosește două seturi de partiții numite sloturi (A și B). Sistemul rulează din slotul curent în timp ce partițiile din celălalt slot nu sunt folosite deloc.

Când se actualizează imaginea dintr-un slot, celălalt slot va conține un sistem funcțional și în caz de erori în urma actualizării se poate face rollback. Condiția este ca nicio partiție din slotul curent să nu fie actualizată.

4.4.6 Avantajele actualizării A/B

Dacă actualizarea eșuează, utilizatorul nu va fi afectat. El va folosi în continuare sistemul de operare vechi. Dacă actualizarea are loc cu succes dar boot-area în partiția actualizată eșuează, atunci dispozitivul va reboota pe partiția veche și utilizatorul va folosi vechiul sistem de operare.

dm-verity va garanta că partiția care boot-ează nu conține o imagine coruptă. Dacă imaginea nou actualizată nu boot-ează din cauza unei actualizări eronate sau unei probleme dm-verity, atunci dispozitivul va putea boota în partiția cu vechea imagine.

Există posibilitatea de a stream-ui actualizările către dispozitivele A/B, eliminând necesitatea de a downloada pachetul OTA înainte de instalare. Acest lucru este util mai ales dacă dispozitivul nu are suficient de mult spațiu liber pentru stocarea întregului pachet.

4.4.7 Actualizare A/B - Atribute

Un slot va avea atributul **bootable** dacă include un sistem funcțional care poate boota. Slotul curent ce conține sistemul care rulează este bootable, iar celălalt slot poate fi o versiune mai veche funcțională, o versiune mai nouă, sau poate conține date invalide.

Un singur slot este cel activ/preferat - cel care va fi folosit de către bootloader la următoarea boot-are.

Atributul **successful** este setat în userspace, la un slot care are deja atributul bootable. Acest slot poate boota cu succes, rula și se poate actualiza.

Un slot bootable care nu a fost marcat successful după mai multe încercări de boot-are, va fi marcat ca unbootable de către bootloader. De asemenea, se va face activ alt slot care este bootable (care a rulat cu succes imediat înainte de actualizare).

4.5 Acces root

4.5.1 Root

Android-ul are la bază un kernel de Linux, care implementează modelul de securitate DAC (Discretionary Access Control). Asta înseamnă că un anumit user, root (UID=0) are putere absolută în sistem.

Pe Android, user-ul root poate: să facă bypass la sandbox, să citească și să scrie fișierele private ale oricărei aplicații, să modifice partiții care ar trebui să fie read-only, să pornească și să oprească servicii de sistem, să elimine aplicații de sistem.

4.5.2 Măsuri de securitate

Folosirea utilizatorului root poate afecta stabilitatea dispozitivului, de aceea nu este permis în mod implicit pe dispozitivele din producție.

În plus, Android-ul încearcă să minimizeze numărul de procese de sistem cu permisiuni de root, deoarece vulnerabilitățile acestor procese pot permite atacuri de tip “privilege escalation” (în care aplicațiile third-party pot obține drepturi de root).

Dacă este activat SELinux în modul enforcing, procesele vor fi limitate de politica globală de securitate. Asta înseamnă că dacă un proces root este compromis nu are neapărat acces complet la dispozitiv. Dar tot se vor putea accesa datele private și se va putea modifica comportamentul sistemului.

Chiar și un proces constrâns de SELinux poate exploata o vulnerabilitate din kernel pentru a obține acces la root complet.

4.5.3 Avantajele accesului root

Accesul la root are anumite avantaje destul de mari:

- Se poate face ușor debugging și reverse engineering al aplicațiilor.
- Se pot realiza anumite personalizări ale sistemului.
- Se pot implementa anumite aplicații speciale de genul firewall, backup complet, partajarea rețelei, fără să fie nevoie de modificarea framework-ului sau de adăugarea unor servicii de sistem.

4.5.4 Build de tip user

Sistemul de build al Android-ului poate genera diferite variante de build pentru un anumit dispozitiv. Acestea au un număr diferit de aplicații, utilitare și valori ale proprietăților de sistem (care generează un comportament diferit al sistemului). Anumite variante de build permit accesul root în shell. Tipul build-ului curent se poate obține din proprietatea de sistem `ro.build.type`.

Build-ul **user** nu include utilitare de diagnostic și dezvoltare, daemon-ul adb (`adbd`) este dezactivat în mod implicit, nu permite debugging pentru aplicațiile care nu au debuggable true în fișierul Manifest, și nu permite accesul root prin shell.

4.5.5 Build de tip userdebug

Build-ul **userdebug** este similar cu user, doar că în plus permite debugging-ul tuturor aplicațiilor și daemnon-ul adb (`adbd`) este activ în mod implicit.

4.5.6 Build de tip engineering

Build-ul de dezvoltare (**engineering**) permite debugging-ul tuturor aplicațiilor, adb este activ în mod implicit, proprietatea `ro.secure` este 0. Asta înseamnă că daemon-ul `adbd` va continua să ruleze ca root și va deține întregul set de capabilități (nu face drop la capabilități).

4.5.7 Comanda su

Comanda `su` (substitute user) este folosită de obicei pentru a face switch de la un user normal la superuser. Are bitul de SUID setat, și permite procesului apelant să obțină un shell root și să ruleze o comandă ca alt user (UID=0).

În build-urile userdebug, accesul root poate fi obținut fără restartarea ADB ca root, prin folosirea comenzii `su`. Doar utilizatorii root (UID=0) și shell (UID=2000) au acces la comanda `su` implicită. Aceasta va seta UID-ul și GID-ul procesului la 0, și va porni un shell nou. Orice comandă executată în acest shell îi va moșteni privilegiile.

4.5.8 Accesul root pe dispozitivele comercializate

În producție (dispozitivele comercializate) vor exista doar build-uri de tip user. Daemon-ul adb va rula sub user-ul shell, și nu va exista nicio comandă `su` pe dispozitiv. Nu sunt permise operații care modifică configurația de bază a sistemului de operare. Nu este permis accesul la kernel-ul Linux. Astfel de operații sunt efectuate de comenzi care necesită privilegii de root.

Obținerea accesului de root pe un astfel de dispozitiv se numește root-are și acest lucru necesită deblocarea bootloader-ului. Dispozitivele care nu permit deblocarea bootloader-ului nu vor putea fi root-ate.

4.5.9 Root-are prin modificarea imaginii boot sau system

Există două metode mai vechi de a roota un dispozitiv: prin scrierea unei noi imagini de boot sau prin modificarea imaginii de sistem.

Pe unele dispozitive, un build user poate fi transformat într-un build engineering sau userdebug doar prin scrierea unei noi imagini de boot (ce conține un kernel custom). Acesta va schimba valorile proprietăților `ro.secure` și `ro.debuggable`. Astfel, se va rula daemonul adb ca root și va permite accesul root prin shell. Totuși, majoritatea build-urilor user curente, vor dezactiva acest comportament la compilare, și valorile proprietăților de sistem vor fi ignorate de către daemonul adb.

A doua modalitate de rootare a dispozitivelor este prin modificarea partiției system. Mai exact, se despachetează imaginea system, se adaugă utilitarul `su` și se suprascrie partiția system cu noua imagine. Acest lucru permite accesul root din shell dar și din aplicațiile third-party. Totuși, din Android 4.3 nu este permis aplicațiilor să execute programe cu SUID, prin faptul că Zygote face drop la capabilități și partiția system este montată cu flag-ul `nosetuid`.

În plus, versiunile mai noi au SELinux pe modul enforcing, ceea ce însemnă că execuția unui proces cu privilegii root nu va schimba contextul de securitate, procesul va fi încă limitat de politica MAC (Mandatory Access Control).

Astfel, SELinux și alte măsuri de securitate vor trebui dezactivate, pentru obținerea accesului root - prin înlocuirea imaginilor boot sau system. Totuși acest lucru va împiedica primirea actualizărilor de sistem de la producător.

De aceea este bine să păstrăm sistemul stock și să facem cât mai puține modificări pentru root-are.

4.5.10 Root-area printr-un pachet OTA

Metoda cea mai folosită în ziua de azi este utilizarea unui pachet OTA care adaugă și modifică fișiere de sistem, fără să fie nevoie de înlocuirea întregii imagini de sistem. Majoritatea aplicațiilor superuser includ un pachet OTA ce trebuie instalat (folosind un custom recovery) și o aplicație manager care se poate actualiza.

4.5.11 SuperSU

Cea mai populară soluție pentru root-are este SuperSU. Include un pachet OTA și o aplicație de management. Este dezvoltat în mod activ de către Jorrit "Chainfire" Jongma.

Pachetul OTA include câteva binare native compilate pentru: arm, arm64, armv7, mips, mips64, x86, x64. Include scripturi pentru instalarea și pornirea daemon-ului SuperSU. Include aplicația de management (apk-ul). De asemenea, include cele 2 scripturi (`update-binary` și `updater-script`) care aplică efectiv modificările din pachetul OTA.

4.5.12 Instalarea SuperSU

`Updater-script` întâi montează read-write sistemul de fișiere rootfs și partițiile system și data. Apoi copiază fișierele la destinațiile lor din sistemul de fișiere.

Binarele native `su` și `daemonsu` sunt copiate în `system/xbin/`. Apk-ul aplicației de management este copiat în `/system/app` și va fi automat instalat la reboot-area dispozitivului.

Apoi scriptul `install-recovery.sh` este copiat în `/system/etc`. Acest script va fi folosit pentru a porni anumite componente la boot-are.

Se vor seta permisiunile și labelurile de securitate SELinux pentru binarele nou instalate (se va seta label-ul `u:object_r:system_file:s0`).

Va apela `/system/xbin/su -install` pentru a efectua inițializări după instalare. În final, va demonta partițiile system și data.

Dacă scriptul se termină cu succes, dispozitivul va reboota în sistemul de operare principal.

4.5.13 Daemonsu

Serviciul `daemonsu` este folosit pentru a evita constrângerile de securitate discutate la începutul secțiunii de Root Access (Zygote face drop la capabilități, SELinux în modul enforcing).

Astfel, aplicațiile vor folosi binarul `su` pentru a executa comenzi sub utilizatorul root, care mai departe va trimite comenzile printr-un socket Unix către `daemonsu` care le execută fiind root și având contextul SELinux `u:r:supersu:s0`.

4.5.14 Aplicația SuperSU

Aplicația de management SuperSU are numele de pachet `eu.chainfire.supersu` (numele procesului). Această aplicație va întreba utilizatorul dacă permite accesul root

pentru aplicațiile care încearcă să folosească comanda `su`.

Accesul root poate fi acordat o singură dată, pentru o perioadă sau permanent, pentru o anumită aplicație. SuperSU are o listă internă de aplicații care au primit acces root de la utilizator și pentru care nu se va mai afișa nici un mesaj (întrebare) pentru utilizator.

4.5.15 Root-area folosind un custom ROM

În CyanogenMod - denumit mai nou LineageOS (cea mai populară versiune custom de Android), `su` este pornit direct ca un daemon din scriptul de inițializare `init.superuser.rc`.

Acest script definește serviciul `su_daemon`, care poate fi pornit și oprit folosind proprietatea de sistem `persist.sys.root_access`. Valoarea acestei proprietăți va determina dacă avem acces de root pentru aplicații, pentru shell-ul adb, pentru toate sau pentru nici una.

Accesul root este dezactivat în mod implicit, dar este ușor de activat din Development Options.

4.6 Bibliografie

- Nikolay Elenkov. 2014. Android Security Internals: An In-Depth Guide to Android's Security Architecture. No Starch Press.
- Joshua J. Drake, Zach Lanier, Collin Mulliner, Pau Oliva Fora, Stephen A. Ridley, and Georg Wicherski. 2014. Android Hacker's Handbook. Wiley Publishing.
- https://source.android.com/security/verifiedboot/ (Accesat: Mai 2021)
- https://source.android.com/devices/tech/ota/ (Accesat: Mai 2021)

Capitolul 5

Vulnerabilități și atacuri

5.1 Concepte generale

În domeniul de științei calculatoarelor, prin vulnerabilitate se înțelege o slăbiciune într-un sistem care ar putea fi exploatată de către o entitate malițioasă.

Exploatând vulnerabilitățile, se pot obține anumite “beneficii” atunci când este atacat un dispozitiv sau o rețea: acces la informații confidențiale, acces root pe dispozitiv, însă pot fi lansate și atacuri cu rol destructiv precum atacuri Denial of Service (DoS).

Vulnerabilitățile sunt, în general, cauzate de următorii factori: un bug în software/hardware (de exemplu, atacul HeartBleed care s-a folosit de faptul că nu se făcea un boundary check în biblioteca de OpenSSL de pe Linux) sau o greșeală de configurare a sistemului (de exemplu, serverul web e configurat să accepte conexiuni HTTPS care folosesc versiuni mai mici ca TLS v1.1).

5.1.1 Noțiuni de securitate

În această secțiune vor fi definiți anumiți termeni specifici zonei de securitate din știința calculatoarelor.

Prin suprafață de atac înțelegem punctele de intrare într-un sistem sau într-o componentă prin intermediul cărora se pot obține beneficii de pe urma vulnerabilităților odată ce s-a ajuns în sistem. Exemple de puncte de intrare în sistem sunt: port USB, interfața de rețea (Ethernet/WiFi), pagini web, e-mail-uri, sisteme de fișiere, etc.

Spre exemplu, cel mai comun și cel mai simplu mod prin care se poate ajunge în spațiul kernel al unui sistem dintr-un mediu remote este prin intermediul unui cadru de date (pachet IP). Acest cadru de date va ajunge la placa de rețea care îl va direcționa către driver-ul de rețea (care se află în spațiul kernel).

Prin vector de atac înțelegem calea sau mecanismul prin care un atacator reușește să obțină acces neautorizat în sistem. Altfel spus, vectorul de atac reprezintă mecanismul prin care atacatorul a reușit să străpungă o intrare din suprafața de atac.

5.1.2 Suprafața de atac a Android-ului

Oricare dintre componentele sistemului de operare (prezentate în Capitolul 1) sau componentele aplicațiilor Android pot avea vulnerabilități și sunt susceptibile la atacuri. Mai mult, sistemul de operare Android este unul foarte complex ceea ce face ca riscul de a avea vulnerabilități și riscul de atac să crească. O zicală renumită în domeniul securității este ca cel mai mare dușman al securității este complexitatea.

În funcție de locul de unde sunt lansați vectorii de atac asupra unui sistem, aceștia pot fi clasificați astfel:

- Remote (la distanță) - vectorul de atac poate fi lansat, în principiu, de oriunde din lume
- Local - este un subset al vectorului de atac remote, doar că acesta presupune ca atacatorul să se afle în vecinătatea sistemului pe care îl atacă
- Fizic - acest vector de atac presupune ca atacatorul să fie prezent fizic lângă sistemul pe care îl va ataca

5.2 Securitatea aplicațiilor

Suprafața de atac este reprezentată de către punctele de intrare într-un sistem sau într-o componentă. În cazul unei aplicații Android, punctele de intrare sunt reprezentate de către cele 4 componente de bază:

- Activitate
- Serviciu (în principiu doar cele exported și cele bound)
- Broadcast receiver
- Content provider

5.2.1 Permisiuni

Componentele aplicațiilor au deseori nevoie de acces la anumite resurse precum Internet, Bluetooth, storage, resurse pentru care nu au acces în mod implicit. Pentru a accesa aceste resurse, sunt necesare permisiuni. O potențială problemă legată de permisiuni este reprezentată de conceptele de undergranting și overgranting.

Prin undergranting se înțelege că o aplicație are mai puține permisiuni declarate în fișierul Manifest decât are nevoie de fapt. Acest lucru poate să ducă la aplicații care vor comporta într-un mod neașteptat din cauza absenței permisiunilor.

Prin overgranting se înțelege că o aplicație are mai multe permisiuni declarate în fișierul Manifest decât are nevoie de fapt. Acest lucru poate să ducă la probleme de securitate pentru că prin overgranting se oferă acces la resurse de care aplicația nu are nevoie de fapt.

Undergranting-ul, dar mai ales overgranting-ul de permisiuni pot fi cauzate atât prin exploit-uri combinate cu puțin social engineering (păcălești utilizatorul să acorde aplicației anumite permisiuni), cât și prin folosirea documentației Android învechite care poate să sugereze că pentru a accesa o resursă este nevoie de un set de permisiuni,

când de fapt, în realitate este nevoie de un set mai redus de permisiuni (middleware-ul are de fapt nevoie de un set redus de permisiuni).

5.2.2 Securizarea comunicației

O altă problemă de securitate la nivel de aplicație este legată de comunicația nesigură. Este indicat ca toate comunicațiile prin rețea cu o entitate din exterior să fie criptate.

În cazul HTTP este indicat să fie folosit întotdeauna HTTPS cu versiunile cele mai recente de TLS (cel puțin TLS 1.1, preferabil TLS 1.2). Este indicat ca și algoritmii criptografici folosiți de certificatele digitale să fie recenți: SHA-256, RSA cu o dimensiune a cheilor de cel puțin 2048 de biți.

5.2.3 Securizarea datelor

Un lucru care trebuie luat de asemenea în considerare este securizarea spațiului de stocare și a informațiilor scrise pe disc. În mod implicit, informațiile scrise pe spațiul de stocare sunt în format plaintext. De aceea soluția este reprezentată de criptarea datelor pe disc.

În Android versiunile 3-4, aplicația Skype deținea fișiere de log care puteau fi accesate de către orice alt proces prin intermediul permisiunilor deprecated world-readable/writable.

O problemă poate să fie și aceea a logging-ului excesiv efectuat de către anumite aplicații. De exemplu, Firefox în versiunile mai vechi de Android, loga informații legate de identificatorii de sesiune și cookies care puteau fi folosite ulterior pentru a face hijack la sesiune.

5.2.4 Securizarea activităților

Un alt pericol este reprezentat de accesarea componentelor unei aplicații Android. Întotdeauna trebuie pusă întrebarea: cine pe cine are voie să acceseze? Cine are voie să acceseze serviciul X? Cine poate emite broadcast-uri? Controlul accesului este realizat prin asocierea unei componente cu o permisiune custom.

Este important să controlăm cine anume poate să acceseze activitățile secundare. Fiind o componentă vizuală, aceasta poate fi folosită pentru a păcăli utilizatorul să execute anumite operații care să ofere informații sau să faciliteze exploit-ul unui atacator.

Un exemplu de atac folosind activități este renumitul atac Cloak and Dagger, de tip UI redressing attack care se folosește de tehnica de clickjacking pentru a păcăli utilizatorul să ofere mai multe permisiuni aplicației malițioase.

Pe scurt, acest atac funcționează în felul următor: aplicația malițioasă creează un overlay (o porțiune de ecran transparentă) care poate fi pusă peste conținutul vizual al unei alte aplicații. Acest overlay poate să conțină elemente vizuale cum ar fi butoane. Overlay-ul construit de aplicația malițioasă conține un buton care pare să fie integrat cu aplicația normală a utilizatorului.

Când utilizatorul apasă pe acel buton, el va acorda aplicației malițioase permisiunea de a accesa serviciul de accesibilitate. Acest serviciu este folosit, în general, de către aplicații menite să fie utile celor cu dizabilități (de exemplu, dizabilități de vedere). Prin

acordarea acestei permisiuni, aplicația malițioasă va intercepta evenimente care sunt trimise atunci când utilizatorul scrie la tastatura (de exemplu, evenimente generate la apăsarea anumitor taste). Folosindu-se de aceste evenimente, atacatorul poate să-și dea seama ce pin este folosit pentru protejarea dispozitivului mobil.

5.2.5 Securizarea serviciilor

Serviciile de tip bound sunt echivalentul unei interfețe de tip server care oferă anumite funcționalități. În mod implicit, serviciile pe Android sunt publice (flag-ul `exported` este setat pe True în mod implicit). Lipsa de control al accesului la acest serviciu poate conduce la o breșă de securitate în cazul în care acel serviciu oferă acces la informații confidențiale.

Pentru controlul accesului, este indicat ca serviciul să fie asociat cu o permisiune în cazul în care acesta trebuie să fie public. Dacă nu este necesar ca el să fie public, serviciul poate fi făcut privat și, astfel, va putea fi accesat doar de componentele din cadrul aceleiași aplicații din care face parte serviciul.

5.2.6 Securizarea broadcast receiver-ilor

Broadcast receiver este o componentă atipică din următorul punct de vedere: avem o sursă care generează mesajul de broadcast și o destinație (receiver-ul) care primește mesajul de broadcast. Din acest motiv există 2 permisiuni asociate cu un broadcast receiver: o permisiune pentru sender și una pentru receiver.

Permisiunea declarată la nivel de receiver este, în esență, permisiunea la nivel de componentă (ca la activități sau servicii) și specifică faptul că doar sender-ii care au declarat în Manifest acea permisiune au voie să trimită un broadcast.

Permisiunea declarată la nivel de sender specifică receiver-ii care pot să primească mesajul de broadcast. În acest caz, receiver-ii trebuie să declare în Manifest permisiunea cerută de sender.

5.2.7 Securizarea content provider-ilor

Un content provider reprezintă, în general, o interfață către date structurate, altfel spus, o interfață către o bază de date SQLite. O bază de date poate să expună date confidențiale. Din fericire, în Android, un content provider nu poate fi accesat în mod implicit de către cineva din exterior.

Controlul accesului pentru o componentă din exterior se face prin intermediul a 2 permisiuni, una pentru citire (READ) și una pentru scriere (WRITE).

Content provider-ul mai oferă și acces granular la nivel de URI, adică permite accesarea doar anumitor intrări din baza de date ce corespund unui anumit pattern.

5.2.8 Aplicații malițioase

O categorie importantă din Potentially Harmful Applications (PHAs) este reprezentată de aplicațiile malițioase (malware). Principalele categorii de malware sunt: virus, spyware, botnet, trojan, rootkit.

Virusul este un malware activ care urmărește să se replice și să infecteze alte dispozitive din rețea. Poate avea efecte distructive: corupere de date, furt de informații confidențiale și denial of service pe sistemul respectiv.

Spyware este un malware cu un caracter pasiv, în sensul că el nu atacă în mod direct sistemul utilizatorului, ci monitorizează sistemul, culege informații despre acesta și le transmite către un server de comandă și control (C2) al atacatorului.

Botnet reprezintă o rețea de calculatoare care sunt coordonate pentru a lansa atacuri de tip Denial of Service sau pentru a obține informații confidențiale de la un sistem.

Trojan reprezintă un tip de malware care pare un utilitar/executabil inofensiv și legitim pentru utilizator (atașament la email), însă odată ce utilizatorul a executat utilitarul, trojan-ul poate să creeze un backdoor în sistem care să îi permită acces atacatorului sau pot fi citite informații confidențiale de pe sistem.

Rootkit reprezintă un executabil care poate să acceseze în mod neautorizat anumite informații din sistem. Verified boot și dm-verity reprezintă soluția pentru a preveni rularea unor rootkit-uri pe partiția system pe Android. Acel rootkit era pus de către un custom Recovery OS pe partiția system, și urma să fie executat când boota sistemul de operare principal.

5.2.9 Google Single Sign On

Urmează să vedem ce face Google pentru a combate apariția aplicațiilor malițioase în ecosistemul Android.

Un prim element este reprezentat de către Google Single Sign On (SSO) care permite unui utilizator să se autentifice cu același cont în serviciile și domeniile publice oferite de către Google (Gmail, Hangouts, Youtube, Google Play Store). Procesul SSO este reprezentat în Figura 5.1.

Primul pas (1) este cererea utilizatorului de acces la un anumit domeniu (de exemplu Gmail). Nefiind logat, utilizatorul va fi redirectat către domeniul SSO (2) unde trebuie să introducă adresa de email și parola. Odată ce acestea au fost introduse, sunt validate de către serviciul SSO (3). După ce au fost validate, serviciul SSO comunică cu un server de autentificare (4) căruia ii cere să genereze un token criptat care va fi transmis către domeniul inițial care trebuia să fie accesat și apoi către utilizator (5-6), mai precis către browser-ul utilizatorului. În acest moment, utilizatorul poate accesa orice serviciu public de la Google fără să se mai autentifice în modul clasic (cu email și parolă). Autentificarea este realizată prin tokenul criptat care a fost stocat în browser.

5.2.10 Aplicații third-party

Google Play Store reprezintă mediul principal de distribuție al aplicațiilor Android. Un alt mediu de distribuție este reprezentat de alte store-uri neoficiale, forumuri de unde pot fi download-ate fișiere apk. Aceste aplicații poartă numele de aplicații third party.

Un motiv principal pentru popularitatea aplicațiilor third party este reprezentat de aspectul financiar: există aplicații contra cost pe Google Play Store, iar dacă utilizatorul vrea să folosească aplicația în mod gratuit, are posibilitatea de a folosi o versiune

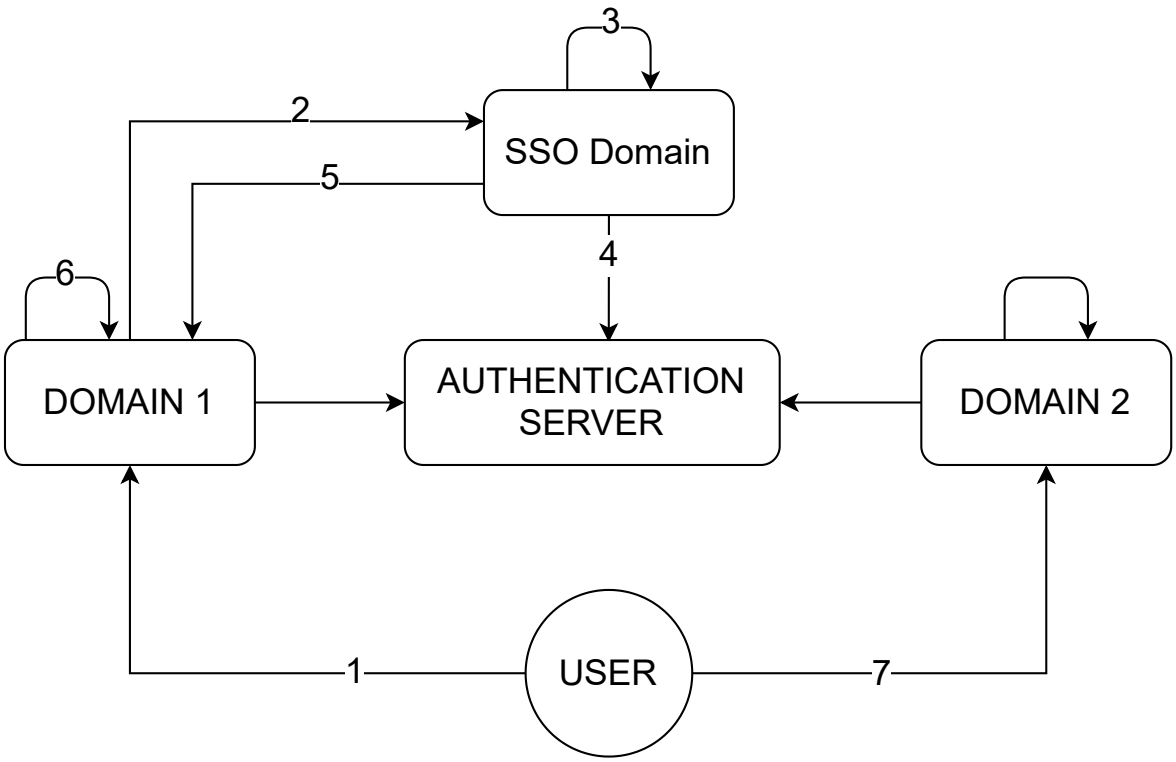

Figura 5.1: Google Single Sign On

crack-uită, third-party a aplicației. Totuși, aplicațiile third-party nu sunt de încredere și de aceea există un risc semnificativ de a instala aplicații malițioase în acest fel.

Un astfel de exemplu este reprezentat de trojan-ul `Android.troj.mdk` care a infectat peste un milion de dispozitive chinezești. Acest trojan era distribuit în versiunile third-party ale aplicațiilor de mobile gaming Temple Run și Fishing Joy.

5.2.11 Google Verify Apps

Pentru a combate apariția și instalarea aplicațiilor malițioase, infrastructura Google se folosește de 2 mecanisme: Verify Apps și Google Play Protect.

Verify Apps este un mecanism care are o componentă care rulează pe client (pe telefonul utilizatorului), cât și pe server (infrastructura Google). Clientul interoghează în mod periodic baza de date Google pentru a verifica cât de sigure sunt aplicațiile instalate pe dispozitiv.

5.2.12 Google Play Protect

Al doilea mecanism de protecție este reprezentat de Google Play Protect. Acesta încarcă o aplicație Android recent urcată pe Google Play Store într-o mașină virtuală izolată (sandbox) populată cu date dummy random (lista de contacte, date pe spațiul de stocare extern, baze de date prepopulate).

Aplicația va fi rulată în această mașină virtuală și va fi analizat comportamentul aplicației (ce resurse accesează, ce sockets deschide, din ce baze date citește, ce permisiuni solicită) folosind atât unelte de analiză statică, cât și de analiză dinamică a codului.

Scopul acestor unelte este de a detecta comportamente anormale sau malițioase din cadrul aplicației.

De-a lungul timpului dezvoltatorii de aplicații malițioase au investigat felul în care funcționează sandbox-ul în care sunt testate aplicațiile: au observat ce date dummy sunt folosite, își pot da seama pe baza fingerprint-ului mașinii virtuale dacă rulează într-un sandbox sau într-o mașină virtuală oarecare.

De asemenea, aplicațiile malițioase pot să nu conțină sau să nu activeze codul malițios. Un lucru des întâlnit pe care îl fac aplicațiile este că atunci când detectează că nu rulează într-un sandbox să deschidă o conexiune cu un server de comandă și control prin care să download-eze codul sursa/executabilul malițios.

Pentru a reduce apariția unor astfel de aplicații, echipa de la Google îmbunătățește în mod constant tehnicile de detecție a malware-ului.

5.2.13 Detecția aplicațiilor malițioase

O primă tehnică este reprezentată de detecția pe bază de semnături. Semnătura este, în esență, un hash introdus într-o bază de date cu semnături. Principiul de funcționare este similar cu cel al antivirusului: compară semnătura aplicației potențial malițioase cu semnăturile din baza de date. Dacă are loc o potrivire, atunci înseamnă ca am dat de un malware, altfel aplicația va fi considerată sigură.

Dezavantajul acestei metode este că fiind o tehnică statică, poate fi păcălită foarte ușor prin schimbarea unui singur bit din executabil. În acest fel, există șanse considerabile ca semnătura să nu mai corespundă cu nicio semnătură din baza de date și în acest fel aplicația va fi considerată sigură.

A doua tehnică este reprezentată de detecția pe baza algoritmilor de machine learning, mai precis rețele neurale și deep learning. Această tehnică este una dinamică și are un potential mai mare decât cea statică pentru a detecta aplicațiile malițioase care și-au schimbat semnătura. În general sunt necesare date în prealabil pentru a putea fi folosite pentru etapa de antrenare de către algoritmii de machine learning.

De multe ori, ambele tehnici sunt folosite, câteodată chiar într-un mod mixt: baza de date de semnături este folosită ca set de antrenare de către algoritmii de machine learning.

5.2.14 Măsuri de protecție minimale

Responsabilitatea reducerii riscului de a avea dispozitivul infectat nu ține doar de Google, ci și de utilizator. Câteva măsuri de protecție minimale sunt prezentate în continuare:

- Folosirea unui nivel minim de securitate pe dispozitive - parole greu de spart, lock screen pin.
- Instalarea aplicațiilor doar din sursele sigure, de exemplu: Google Play Store.
- Conectarea doar la rețele WiFi sigure care folosesc ca standard de securitate cel puțin WPA2, preferabil WPA3.
- Evitarea conectării la rețele WiFi open.
- Evitarea root-ării telefonului.

- Evitarea instalării partițiilor custom de recovery sau de boot.
- Actualizarea periodică a sistemului de operare (versiune, patch-uri de securitate).

Sunt 2 motive pentru care este importantă actualizarea sistemul de operare: În primul rând, cu cât versiunea sistemului de operare este mai recentă, cu atât ar trebui să fie mai sigură (în general). În al doilea rând, cu cât versiunea sistemului de operare este mai recentă, cu atât a avut mai puțin timp să fie expusă către atacatori și, astfel nu a trecut suficient timp pentru ca ei să cunoască foarte în detaliu noile actualizări.

Acest al doilea comportament este evidențiat și prin graficul[1] reprezentat în Figura 5.2 care arată evoluția numărului de PHA în funcție de versiunea de Android în intervalul ianuarie - martie 2021.

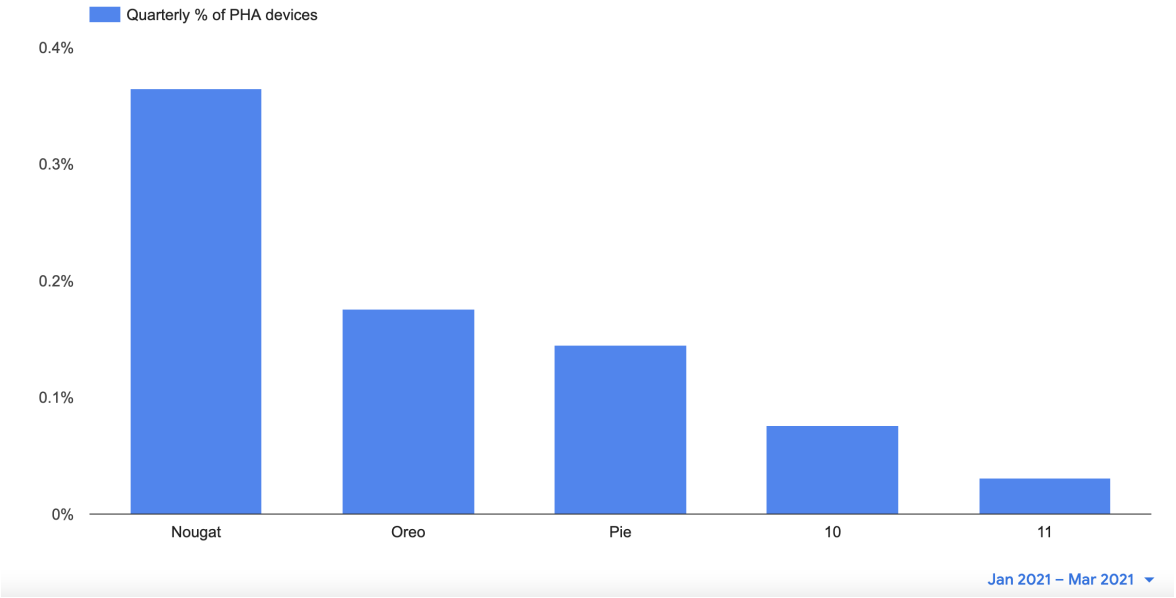

Figura 5.2: Evoluția numărului de PHA în funcție de versiunea de Android

5.3 Suprafețe de atac la distanță

În continuare vor fi prezentate în detaliu de suprafețele de atac de pe un dispozitiv mobil, începând cu suprafețele de atac la distanță (remote), îndeosebi atacurile prin rețea.

Un mare avantaj al dispozitivelor mobile de tip smartphone este că acestea, în general, nu au servicii de rețea active. Altfel spus, este foarte puțin probabil ca pe un smartphone să ruleze un server web sau un server SSH. Absența acestor servicii conferă un grad ridicat de securitate dispozitivului, eliminând astfel riscul accesului neautorizat la dispozitivul mobil prin acele servicii sau riscul unei scanări active în rețea folosind `nmap`.

Cu toate acestea, dispozitivul mobil este susceptibil la atacurile comune care se pot face într-o rețea WiFi sau celulară. Astfel, se poate lansa atacuri de tipul:

1) Spoofing pentru ARP, DNS, DHCP - în care atacatorul se dă drept o entitate legitimă: asociază IP-ul unei ținte (de exemplu, un gateway) cu propria adresă MAC, se dă drept

[1] https://transparencyreport.google.com/android-security/device-platform-safety

un server-ul DNS din rețea sau se dă drept server DNS sau gateway prin intermediul configurării DHCP.

2) Atacuri de tip în Man în the Middle (MitM) prin intermediul cărora poate să intercepteze traficul victimei.

3) Atacuri la nivel de protocol TCP - SYN flooding prin care inundă victima cu cereri TCP (half-open TCP connections) și, astfel, victima nu va mai putea iniția conexiuni legitime.

4) RST attack reprezintă un atac prin care atacatorul creează un pachet TCP cu flag-ul RST setat pe 1, cu IP sursa/destinație valide (sursa IP-ul victimei) și îl folosește pentru a termina conexiunea TCP a victimei. Marele firewall chinezesc (The great firewall of China) folosește RST attack pentru a preveni anumite conexiuni și, astfel controlează/cenzurează accesul la anumite resurse.

5) Denial of Service attacks precum TCP SYN flooding, ping of death sau DDOS prin care mai multe dispozitive atacă simultan victima (botnets).

5.3.1 ARP Spoofing

În continuare este prezentat un exemplu de ARP spoofing. ARP este un protocol de nivel 2.5 care este folosit pentru a determina adresa MAC corespunzătoare a unei adrese IP dintr-o rețea locală.

Protocolul folosește 2 mesaje: ARP Request în care sursa întreaba în rețea: stația IP-ul X ce adresa MAC are? Acest mesaj ajunge la toate stațiile din rețea, inclusiv la destinația legitimă și, inclusiv, la potențialii atacatori. Odată primit mesajul, destinația legitimă va răspunde cu un ARP Response în care spune ca IP-ul X este asociat cu adresa MAC Y. Odată ajuns acest mesaj la sursă, ea își actualizează tabela ARP.

O breșă de securitate legată de protocolul ARP este faptul că această tabelă este actualizată întotdeauna când este primit un ARP Response. În exemplul din Figura 5.3, se poate vedea cum atacatorul va trimite un ARP Response falsificat în care susține ca IP-ul gateway-ului este asociat de fapt cu adresa MAC a atacatorului. Odată primit acest mesaj, sursa își va actualiza tabela ARP și, atunci când va dori să trimită mesaje către gateway pentru a trimite pachete înafara rețelei, acestea vor fi de fapt redirectate către atacator.

5.3.2 Atacuri asupra rețelelor mobile

Spre deosebire de sistemele desktop, dispozitivele mobile prezintă o suprafața de atac nouă: conexiunea celulară prin GSM, 4G sau 5G. Această zonă nouă introduce și niște vectori de atac noi prin intermediul SMS-ului, MMS-ului sau a WAP push - un protocol prin care operatorul de telefonie mobilă poate să configureze anumite elemente pe telefon sau poate să taxeze utilizatorul.

Un prim vector de atac este reprezentat de "dialer attack" care se folosește de coduri USSD pentru a executa atacul. Codurile USSD sunt coduri care pot fi folosite de către client pentru a cere informații de la operatorul de telefonie mobilă sau pentru a instrui operatorul să execute anumite operații asupra telefonului (de exemplu, PUK reset).

Cel mai comun exemplu de USSD code este *123# prin intermediul căruia se pot afla informații despre abonamentul utilizatorului. Există coduri USSD care pot avea efecte

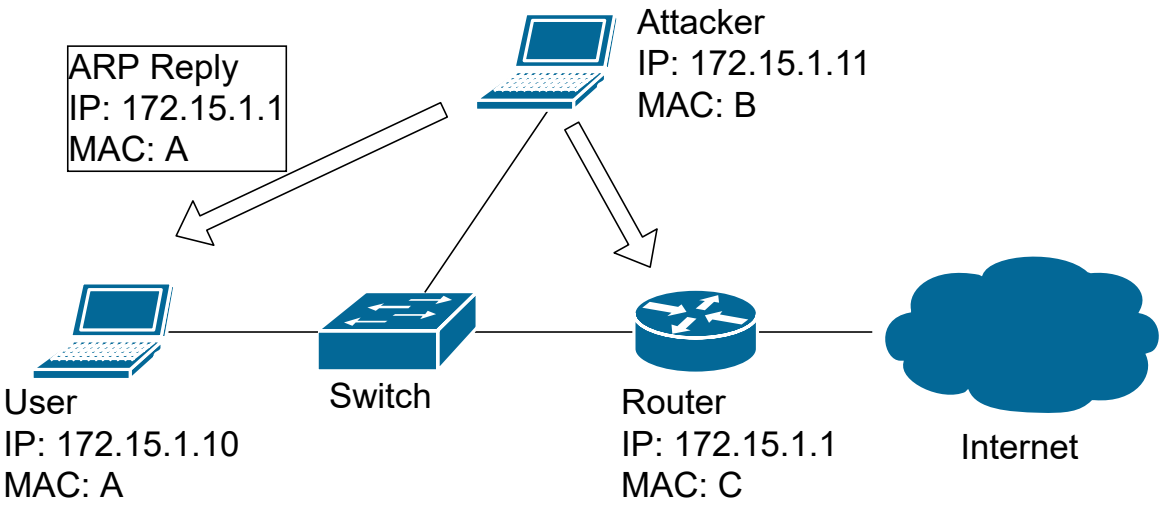

Figura 5.3: ARP Spoofing

distructive precum: factory reset, PUK reset care dacă va fi executat de 10 ori, va rezulta în distrugerea cardului SIM.

Acest atac poate fi lansat prin intermediul SMS-urilor sau a mesajelor de Twitter/Hangouts care să conțină un URI similar cu: tel://URI. Acest URI va conține și codul USSD.

5.3.3 Atacul Stagefright

O vulnerabilitate mai recentă poate fi găsită în biblioteca nativă multimedia Android Stagefright. Această bibliotecă este folosită pentru encoding/decoding de fișiere .mpeg sau .mp4.

Exploit-ul poate fi declanșat prin includerea clipurilor .mp4 special create pentru a declanșa un integer overflow în biblioteca Stagefright care va rezulta în cele din urmă într-un heap overflow.

Acest overflow poate fi folosit mai departe pentru a executa un shellcode care să creeze un reverse TCP connection callback. Altfel spus, îl notifică pe atacator că poate să inițieze o conexiune TCP către socket-ul deschis pe dispozitivul victimei.

5.3.4 Atacuri asupra browser-elor web

O altă suprafață de atac este reprezentată de către clienții web precum browser-ele. Tehnologiile folosite cu preponderență sunt HTTP(S), FTP(S), HTML, framework-uri de Javascript. Toată această suită de tehnologii, precum și interconectarea lor creează un mediu propice pentru atacuri asupra browser-elor web.

Un prim exemplu de atac este cel de rogue URL attack prin care victimei i se oferă un link aparent legitim (către contul bancar de exemplu), dar, în realitate, link-ul acela este unul ilegitim: atacatorul a construit un site web care să semene foarte bine cu site-ul bancar și, în acest fel, îl păcălește pe utilizator.

Alte două tipuri de atacuri sunt: cross-site scripting (XSS) și cross-site request forgery (XSRF).

XSS (Figura 5.4) reprezintă un atac în care atacatorul introduce într-o pagină web un script care poate fi folosit pentru a colecta anumite informații. Ca acest lucru să se întâmple, trebuie ca site-ul web să fie vulnerabil la script injection (1). Atacatorul poate să injecteze un script care să colecteze cookie-urile de sesiune ale celor care vizitează site-ul (2). Atunci când utilizatorul va vizita site-ul web (3), scriptul injectat va fi executat, va colecta cookie-urile de sesiune și le va transmite către atacator (4).

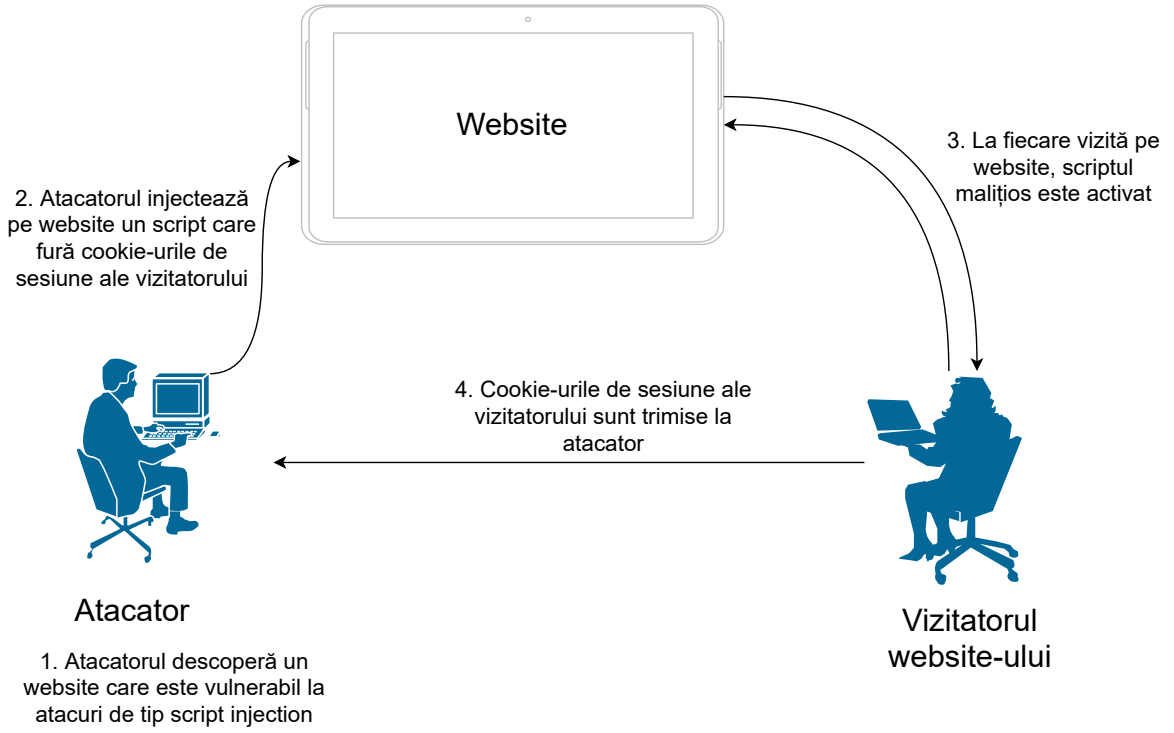

Figura 5.4: Atacul XSS

XSRF (Figura 5.5) reprezintă un atac în care atacatorul creează un request falsificat care să fie executat în mod automat de către site-ul web pentru care este destinat request-ul.

În acest exemplu, atacatorul creează un request falsificat pentru un transfer bancar către conturile lui (1). Atacatorul va include acest request într-un hyperlink care este trimis către utilizatorii acelui site web în speranța că unii dintre ei vor fi online în acel moment (2). În cazul în care utilizatorul dă click pe hyperlink (3), request-ul este trimis către site-ul web (în cazul de față către bancă). Site-ul web validează request-ul și trimite banii către contul atacatorului (4).

A se observa că în cazul XSS, ținta directă a vectorului de atac a fost site-ul web, iar în cazul XSRF este utilizatorul.

5.3.5 Atacuri asupra altor clienți web-based

Pe dispozitivele mobile există și clienți/aplicații web-based precum Twitter sau Dropbox. Un procent semnificativ din aceste aplicații sunt predispuse la atacuri de tip Man-in-the-Middle din cauza faptului că nu se validează certificatele digitale folosite.

De aceea este recomandat ca, întotdeauna, să fie validate certificatele digitale folosite de aplicații pentru autentificarea între client și server.

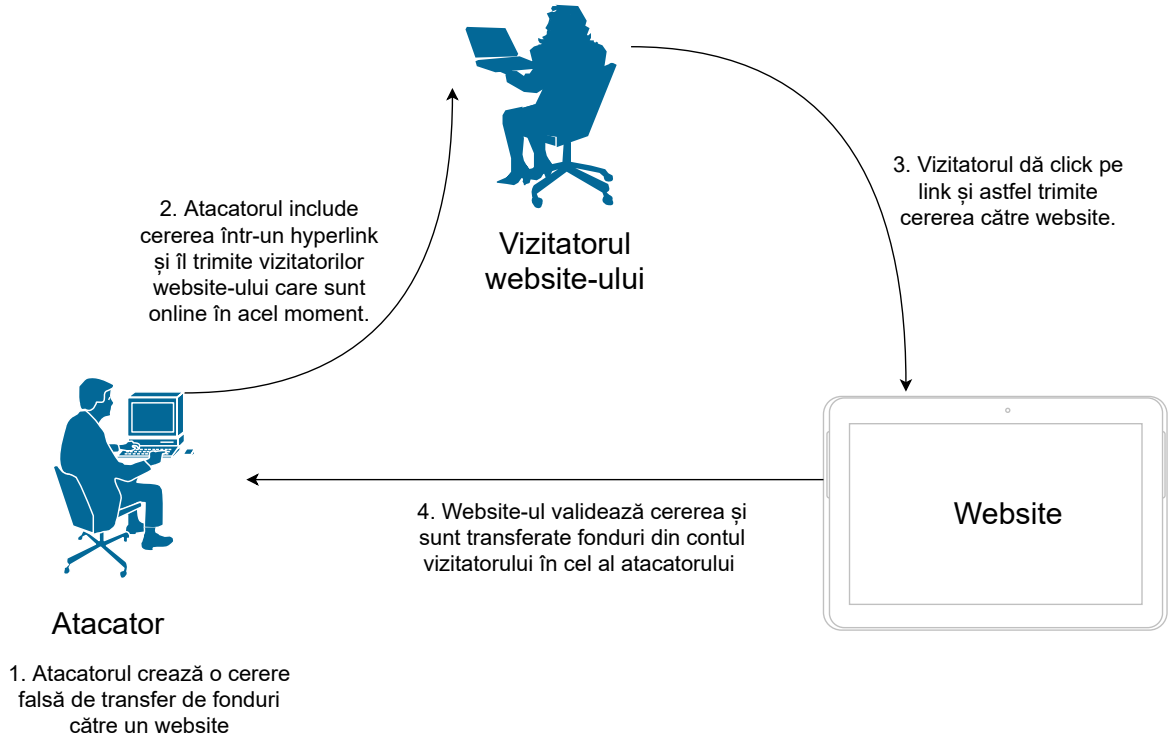

Figura 5.5: Atacul XSRF

5.3.6 Atacuri asupra GPS

O altă suprafață de atac este reprezentată de GPS. În clipa de față, nu există un atac prin GPS care să poată compromite un dispozitiv (de exemplu, să obțină acces root). Motivul este reprezentat de simplitatea datelor pe care le oferă GPS: doar latitudine, longitudine și altitudine. Intern, GPS-ul are mecanisme complexe, dar ceea ce oferă ca informații reprezintă doar niște numere care nu pot fi folosite pentru a compromite un dispozitiv mobil.

Cel mai comun atac prin GPS este acela de GPS spoofing - prin care se oferă informații false despre locație.

Un caz renumit de GPS spoofing este cel al unui cetățean german din Berlin care se plimba pe o stradă cu un căruț cu 99 de telefoane mobile cu Google Maps pornit și cu GPS-ul activ. Google Maps determină dacă într-o zonă este congestie în trafic pe baza numărului de dispozitive cu GPS activ și a vitezei de deplasare. Când serverele Google Maps au detectat 99 de entități care se deplasează cu viteza redusă pe stradă, a considerat ca este congestie în trafic, deși strada era liberă. Astfel, mașinile care urmau să intre pe acea stradă au fost redirecționate pe alte străzi.

Între timp Google a rezolvat această problema: cel mai probabil se face o corelare mai bună între numărul de participanți în trafic și viteza lor (dacă toți se deplasează cu o viteză absolut identică, e posibil să fie ceva în neregulă).

5.3.7 Atacuri asupra tehnologiilor celulare

În ceea ce privește tehnologiile celulare precum GSM, HSPA, LTE sau NR (2G, 3G, 4G și 5G), atacurile urmăresc de obicei să interacționeze cu driver-ul de modem baseband, altfel spus, driver-ul responsabil pentru comunicarea celulară.

Un exemplu de atac este emularea unei stații de bază (rogue base station). Stația de bază reprezintă antena la care se conectează telefonul atunci când se folosesc date mobile sau când este realizat un apel. Totuși, este destul de dificil de emulat o stație de bază pentru că de cele mai multe ori conține hardware și software proprietar dezvoltate de un anumit vendor (Huawei, Zitec, etc.).

De asemenea, în prezent, o astfel de emulare ar costa de la câteva zeci de mii până la sute de mii de dolari în funcție de capabilitățile stației de bază, deși există câteva inițiative open-source pentru a construi stații de bază mai ieftine.

RIL (radio interface layer) este o componentă a baseband-ului ce poate fi atacată prin intermediul comenzilor de atenție (AT commands). Acestea sunt comenzi mai vechi (de pe vremurile GSM și GPRS) puteau fi trimise de către operatorul de telefonie mobilă pentru a taxa utilizatorul, pentru a citi sau a scrie mesaje pe dispozitivul mobil sau pentru a face downgrade la sistemul de operare.

Din motive de backwards compatibility, modemul de baseband încă trebuie să mai suporte aceste comenzi de atenție. Interacțiunea cu modemul de baseband se poate face atât prin USB, cât și prin Bluetooth.

5.3.8 Atacuri asupra Bluetooth

O altă suprafață de atac care a devenit destul de folosită în ultimii ani, este reprezentată de Bluetooth. De-a lungul timpului, Bluetooth a prezentat mai multe slăbiciuni legate de Bluetooth pairing și de algoritmii de criptare folosiți în stiva Bluetooth.

Câteva exemple de atacuri care pot fi realizate prin Bluetooth:

- Bluejacking - trimiterea de mesaje nesolicitate către o țintă Bluetooth (un echivalent de Denial of Service pentru Bluetooth).
- Bluesnarfing - în această categorie intră atacurile care urmăresc să obțină acces la distanță pe un dispozitiv cu Bluetooth.
- Blueborne - este un exemplu de atac de tip Bluesnarfing care reușește să obțină acces la distanță printr-un heap overflow generat de trimiterea mai multor pachete de Bluetooth discovery special create.
- BlueFrag - este un atac care permite execuția codului la distanță prin intermediul unui pachet Bluetooth special creat. Nu necesită pairing, atacatorul trebuie doar să se afle în proximitatea victimei și poate să deducă adresa Bluetooth din adresa MAC pe anumite dispozitive.

5.3.9 Atacuri asupra WiFi

În continuare este descrisă o suprafața de atac destul de comuna și anume WiFi (standardul 802.11). Sunt prezentate câteva potențiale atacuri. Standardele criptografice folosite în WiFi de-a lungul timpului și în prezent: WEP și WPA, WPA2 și WPA3.

Rogue AP (access point) - în acest atac se poate instala un access point ilegitim într-o rețea de corporație, spre exemplu, fără ca acesta să fie detectat în prima fază. De obicei acest access point este unul software pentru că el trebuie să facă parte din rețeaua

organizației. AP-ul poate să fie și hardware, dar este mult mai greu de instalat în rețea pentru ca este nevoie de permisiunea administratorului de rețea pentru a face acest lucru.

Unul dintre cele mai grave atacuri asupra WiFi din ultimii ani este reprezentat de către Krack (Key Reinstallation Attack), un replay attack care urmărește o vulnerabilitate în 4-way handshake-ul de stabilire a cheii secrete în standardul WPA2. Prin acest replay attack, se retransmite încontinuu al 3-lea mesaj din acest handshake care va reseta cheia de criptare WPA2. Pe măsură ce numărul de reset-uri crește, cu atât cheia respectivă va fi mai expusă până când în cele din urmă se va putea afla întreaga cheie de criptare și se va putea decripta traficul dintre client și access point.

Atacul acesta este foarte grav prin prisma faptului că vulnerabilitatea se găsește la nivel de protocol și, din păcate, nu se pot face foarte multe în această privință pentru ca ar trebui actualizate atât firmware-ul, cât și hardware-ul din access point. Totuși, atacurile prin WiFi sunt atacuri care trebuie executate de la o distanță mică (cel mult 300 de metri), astfel că acest atac este mai limitat din acest punct de vedere.

5.3.10 Atacuri asupra NFC

O ultimă suprafață de atac folosită pentru transferul datelor este reprezentată de Near Field Communication (NFC). În primii ani de viață, NFC a suferit din cauza absenței autentificării și a criptării datelor. Câteva potențiale atacuri sunt browser attack și NFC relay attack.

Browser attack - prin NFC se pot transfera orice fel de date, inclusiv URL-uri. NFC reader-ul de pe dispozitivul destinație poate să decidă în cazul unui URL să îl deschidă în mod implicit cu browser-ul fără a mai întreba utilizatorul dacă dorește să-l deschidă. URL-ul respectiv poate fi un rogue URL care să conțină și cod injectat de Javascript care poate fi executat pe dispozitiv, iar mai departe să ofere informații atacatorului.

NFC relay attack este un atac folosit de obicei atunci când plătim cu cardul prin NFC, apropiindu-l de dispozitivul casierului. În cazul în care dispozitivul respectiv are acces la Internet și are o aplicație malițioasă instalată, aceasta ar putea să intercepteze plata cu cardul prin NFC și să redirecționeze tranzacția către atacator.

Redirecționarea tranzacției se face prin rețea către un server de comandă și control care, mai apoi, va redirecționa tranzacția către un POS (point of sale), de unde plata se va face în contul bancar al atacatorului.

5.4 Suprafețe de atac locale

Prin atacuri locale înțelegem atacuri prin care atacatorul trebuie să afle în vecinătatea victimei, precum și atacuri care sunt mai mult cauzate de vulnerabilități ale unor componente interne din sistem care nu reprezintă punctele de intrare ale suprafeței de atac.

Există vulnerabilități și la nivelul sistemului de fișiere precum F2FS care pot conduce la coruperi de memorie și la posibilitatea execuției de cod în spațiul kernel. Vulnerabilitatea apare atunci când se folosește sistemul de fișiere F2FS și se montează o imagine de

fișier sau o partiție malițioasă cu atributul `losetup`. Acest atribut face ca imaginea de fișier sau partiția să fie văzute ca un dispozitiv bloc.

Și stiva TCP/IP prezintă la rândul ei vulnerabilități cauzate în general de absența boundary checks sau de race conditions. Un astfel de race condition poate să apară în cazul pachetelor IP fragmentate care pot fi șterse înainte de a fi adăugate în lista LRU (Least Recently Utilised) de fragmente.

Accesul unei zone deja eliberate de memorie este un acces invalid și poartă numele use-after-free. În cazuri frecvente și repetate, acest comportament poate să ducă la un Denial of Service intern.

Binder-ul are probleme similare cu cele ale stivei TCP/IP: de exemplu, probleme legate de use-after-free în cazul apelurilor de ioctl.

Ashmem (memoria anonima partajată) a fost folosită în Android versiunile 3-4 în cadrul jailbreak-ului KillingInTheNameOf. Acest jailbreak creează o zonă de memorie partajată în care mapează zona de memorie corespunzătoare proprietăților de sistem. Această zonă de memorie era creată cu atributul `PROT_WRITE` ceea ce face aceasta zonă de memorie să fie writable. În consecință, proprietatea de sistem `ro.secure` poate fi setată pe 0 și în acest fel, vom avea acces root prin ADB.

5.5 Suprafețe de atac fizice

Există și atacuri pentru care este nevoie de acces fizic la dispozitivul mobil. Exemple de atacuri: dezmembrarea/distrugerea dispozitivului, atac prin USB.

Un atac prin USB este acela prin care se transmit comenzi de atenție (AT) prin USB. Un alt atac este acela care exploatează vulnerabilitatea din daemon-ul vold (volume daemon).

Daemon-ul vold este responsabil cu montarea partițiilor de pe USB pe dispozitiv. În versiunile mai vechi de Android, exista o vulnerabilitate care permitea suprascrierea partiției system de pe dispozitiv cu partiția system de pe USB.

5.6 Atacurile side channel

Atacurile de tip side channel reprezintă o categorie aparte de atacuri. Acestea atacă dispozitivul prin metode neconvenționale, prin alte canale și nu atacă folosind vulnerabilități directe. Ele atacă implementarea și anumite efecte indirecte/fizice ale implementării respective.

5.6.1 Clasificarea atacurilor side channel

Ele pot fi clasificate în felul următor:

- Active vs. pasive - sunt active în cazul în care atacatorul trebuie să intervină pentru a provoca acel atac; în cazul celor pasive atacatorul nu trebuie să intervină, el trebuie doar să monitorizeze sau să observe

- Fizice vs. logice - este echivalent cu hardware vs software. Există hardware side-channel attacks în care efectele se observă în hardware sau software side-channel attacks în care efectele se observă cu precădere în comportamentul software.
- Local vs. vicinity vs remote - similar cu suprafețele de atac fizice, locale și remote

5.6.2 Atacuri locale pasive

În această secțiune sunt descrise câteva exemple de atacuri locale pasive.

1) Power analysis attack. Se poate analiza consumul de putere al dispozitivului și pe baza lui se poate deduce când anume se fac operații criptografice. În cazul algoritmului criptografic simetric DES, se poate deduce cheia secretă folosită.

2) Electromagnetic analysis attack. Este similar cu atacul anterior, doar că se folosește energia electromagnetică emanată de dispozitiv. Este un atac mult mai periculos pentru că se pot deduce cheile folosite de algoritmii criptografici AES, RSA, ECDSA și ECC.

3) Smudge attack. Se aplică pentru telefoanele cu touchscreen pe care se adună praf și mizerie. Când utilizatorul face swipe pe ecran sau când introduce pattern-ul de unlock pe telefon, praful și mizeria de pe telefon pot să lase urme specifice care pot fi apoi utilizate pentru a deduce unlock pattern-ul.

4) Shoulder surfing and mirroring. Se referă la atacuri în care prin intermediul reflexiilor în geamuri, sau ochelari de soare, se poate deduce ce a scris utilizatorul.

5) Hand and device movement. Se referă la atacuri în care atacatorul nu vede în mod direct ecranul utilizatorului, dar poate să-și dea seama ce scrie sau ce face pe baza mișcării mâinilor.

5.6.3 Atacuri locale active

În această secțiune sunt descrise câteva exemple de atacuri locale active.

1) Clock and power glitching. Atacatorul are nevoie de o sursă de putere pentru a realiza acest atac. Dispozitivul mobil va fi conectat la această sursă de putere și va putea suferi modificări în ceea ce privește frecvența de procesare și puterea consumată. Astfel, se poate face underclocking pe dispozitiv, caz în care operațiile pe telefon se vor mișca foarte lent. Se poate face și overclocking, caz în care operațiile vor fi foarte rapide, dar consumul de putere/baterie va crește.

2) Electromagnetic fault injection. Este un atac în care celulele de memorie sunt supuse la un impuls electromagnetic suficient de puternic (EMP attack) în așa fel încât să se schimbe valorile celulelor de memorie.

4) Laser and optical faults. Este similar cu atacul anterior, doar că se poate folosi un laser pentru a modifica valorile celulelor de memorie.

5) Temperature variation este un atac în care se folosesc temperaturi foarte înalte sau foarte joase. Temperaturile foarte înalte pot să ducă la celule de memorie nefuncționale sau la bit errors. Temperaturile foarte joase pot produce the remanence effect pentru memoria RAM în care după reset, RAM-ul nu s-a golit încă, valorile mai sunt stocate pentru o perioadă scurtă de timp. Acest tip de atac mai poartă și numele de cold-boot attack.

5.6.4 Atacuri pasive din vecinătatea dispozitivului

În această secțiune sunt descrise câteva exemple de atacuri pasive din vecinătatea dispozitivului.

1) Network traffic analysis. Se pot folosi utilitare de monitorizare pasivă precum Wireshark sau tcpdump în modurile monitor sau promiscuous pentru a vedea ce pachete se transmit în aer.

2) USB power analysis. Folosind stațiile publice de încărcare a dispozitivelor prin USB, se poate deduce puterea consumată de dispozitiv și pe baza acestui aspect, se poate deduce (prin inferență) ce site-uri a vizitat utilizatorul în browser.

3) WiFi signal monitoring. Canalele de comunicare în WiFi au o proprietate denumită CSI (Channel State Information) care conține informații despre cum semnalul se propagă de la transmițător la receptor și cum este influențat de efecte precum shadowing, fading, scattering sau atenuarea cu distanța. Pe baza CSI, se poate deduce folosind modele de inferență ce anume scrie utilizatorul pe dispozitiv sau ce pin are pentru unlock screen. Prin apăsarea diferitelor "taste", canalul de WiFi va avea un CSI diferit.

5.6.5 Atacuri pasive la distanță

În această secțiune sunt descrise câteva exemple de atacuri pasive de la distanță.

1) Linux inherited procfs leaks. Informațiile despre un anumit proces (ce file descriptori folosește, cât RAM și CPU consumă) pot fi găsite în `/proc/[pid]/status`. Pe baza acestor informații se poate deduce ce comportament de browsing are un utilizator prin prisma memoriei folosite. De asemenea, dimensiunea memoriei partajate într-o aplicație poate să sugereze tranziția de la o activitate la alta pe Android. Numărul de context switch-uri și de întreruperi hardware poate să sugereze scrierea de la "tastatură" și pe baza acestora se pot deduce anumite keystroke patterns.

2) Data-usage statistics. Android colectează cât trafic incoming și outgoing există per aplicație. Pe baza acestui lucru, se poate deduce în cadrul browserelor web ce site-uri accesează utilizatorul.

3) Page deduplication. Acest concept se referă la faptul ca dacă 2 procese distincte accesează 2 pagini de memorie (în propriul spațiu de adrese) absolut identice, din motive de economie de memorie, este mult mai indicat ca cele două pagini să fie deduplicate, adică să fie unite într-o singură pagină de memorie.

Altfel spus, paginile de memorie identice din procese diferite vor deveni zone de memorie partajată pentru acele procese, până în momentul în care unul dintre procese va dori să scrie în zona respectivă (mecanismul Copy-on-Write). În această situație, se va genera un copy-on-write fault și se va face o copie a zonei de memorie.

Pe baza acestui comportament, se poate implementa o aplicație care strânge imaginile cele mai comune de pe Internet, iar atunci când utilizatorul va descărca și el una dintre aceste imagini și va dori să o modifice, în acel moment se va genera un copy-on-write fault care va indica atacatorului care imagine a fost descărcată de către utilizator.

4) Microarchitectural attacks. Arhitectura calculatoarelor moderne include multe componente menite să îmbunătățească performanța (de exemplu, memorii cache pe mai multe nivele). Atacurile microarhitecturale se bazează pe măsurarea timpilor de

execuție și de acces la memorie pentru cache, RAM, disk sau branch prediction units pentru a determina ce informații sunt transmise.

Un exemplu clasic de cache timing attack este acela aplicat algoritmilor criptografici. În mod special la algoritmul AES se poate determina cheia secretă.

5) Location inference. Pentru determinarea locației se pot folosi accelerometrul și giroscopul. Pe Android există un API capabil să detecteze dacă difuzorul telefonului este activ sau nu. Acest lucru înseamnă că se poate determina cât timp este folosit difuzorul pentru a transmite sunete.

Șoferii folosesc de obicei Google Maps sau Waze cu difuzorul pornit pentru a ajunge la o anumită locație. Botul care indică traseul folosește exprimări parțial hardcoded, precum și direcția de mers: la 300 de metri faceți prima la dreapta.

Pe baza celor 2 puncte de mai sus, se poate determina folosind modele de inferență care este traseul șoferului pe baza faptului că se determină cât durează fiecare cuvânt pe care îl spune bot-ul.

6) Speech recognition. Semnalele acustice pot influența măsurătorile făcute de giroscop și, astfel, folosind modele de inferență, se poate determina ce spune utilizatorul.

5.6.6 Atacuri active la distanță

Rowhammer este un atac foarte popular în ultimii ani și apare cu preponderență în memoriile SDRAM de tip DDR3 sau DDR4. Motivul pentru care apare în aceste modele este pentru că celulele de memorie sunt mai mici și pentru că acestea au o densitate mult mai mare, ceea ce înseamnă ca celulele de memorie sunt foarte apropiate una de alta.

Apropierea dintre celule face ca în anumite condiții, sarcina electrică a unei celule să se propage în celulele învecinate și, astfel, se face bypass la izolarea dintre celule. Acest comportament poate fi folosit pentru a lansa exploit-uri precum RAMpage cu ajutorul căruia se poate obține acces root.

Un exemplu de atac folosind Rowhammer (Figura 5.6) se face prin activarea repetată a două rânduri de memorie, N-1 și N+1 (rânduri agresor), cu scopul de a determina ca biții din rândul N (rând victimă) să fie modificați.

5.7 Obținerea accesului root

Acestă secțiune include o prezentare scurtă a jailbreak-ului **RageAgainstTheCage**.

În secțiunea de cod prezentată mai jos se observă 3 apeluri de funcții: o funcție executată ca utilizator privilegiat, un apel `setuid` care face drop la capabilitățile de root ale procesului și, astfel, va deveni proces neprivilegiat și, în final, o funcție executată ca utilizator neprivilegiat.

```
/* Cod care rulează cu privilegii ridicate */
do_stuff_as_privileged();

/* Renunță la privilegii pentru a rula ca utilizator non-privilegiat */
setuid(uid);
```

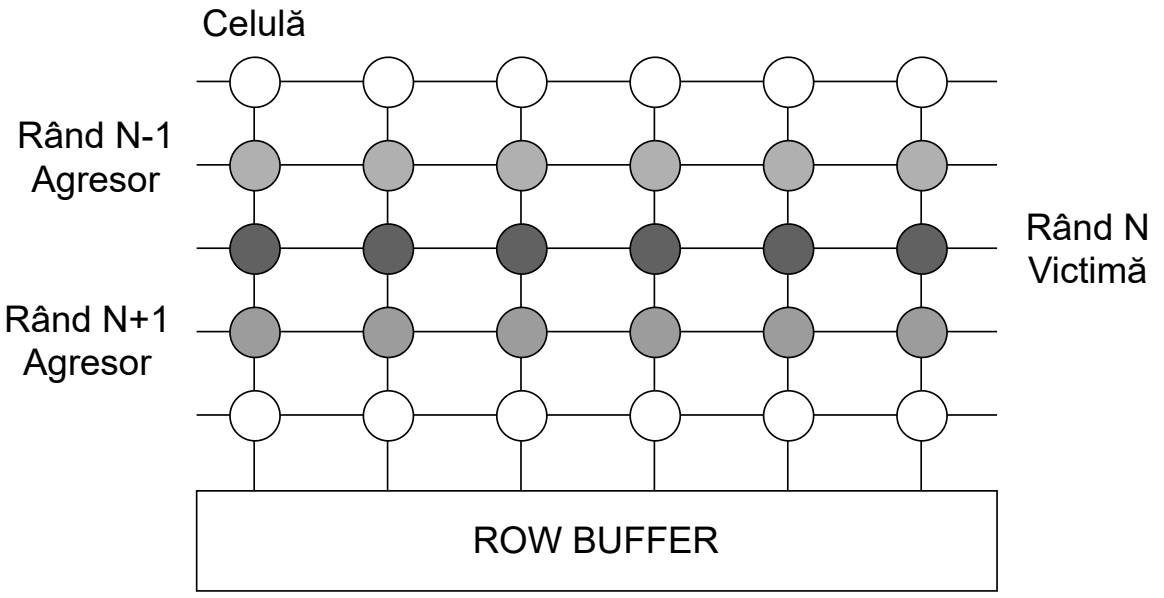

Figura 5.6: Atacul Rowhammer

```
/* Cod care rulează cu privilegii scăzute */
do_stuff_as_unprivileged();
```

Care dintre aceste funcții ar trebui să eșueze pentru a putea rămâne cu acces root? Răspuns: funcția `setuid`.

În pagina de manual[1], se poate observa că `setuid` o să genereze codul de eroare `EAGAIN` atunci când utilizatorul cu UID-ul `$uid` a atins limita `RLIMIT_NPROC`.

```
ERRORS
    EAGAIN The uid does not match the current
           uid and uid brings process over its
           RLIMIT_NPROC resource limit.
```

În pagina de manual[2] se observă că `RLIMIT_NPROC` reprezintă numărul maxim de procese care pot să ruleze sub un anumit utilizator (UID).

Dacă avem prea multe procese sub același UID, nu se mai poate crea un alt proces sub același UID, prin urmare `setuid` va eșua.

Revenind la secțiunea de cod originală, `setuid` va eșua dacă s-a atins limita `RLIMIT_NPROC` pentru utilizatorul neprivilegiat la care ar trebui să se facă switch-ul în mod obișnuit.

```
RLIMIT_NPROC
        This is a limit on the number of extant process (or, more
        precisely on Linux, threads) for the real user ID of the
        calling process.  So long as the current number of
        processes belonging to this process's real user ID is
        greater than or equal to this limit, fork(2) fails with
        the error EAGAIN.
```

[1] https://linux.die.net/man/2/setuid
[2] https://man7.org/linux/man-pages/man2/getrlimit.2.html

Care este procesul din Android care poate fi luat în considerare pentru un astfel de atac? Răspuns: `adb`.

Dacă este inspectat codul sursa al `adb`, se poate observa că și el apelează la un moment dat atât `setuid`, cât și `setgid`. Când este creat procesul `adb` prima oară, el rulează ca root, apoi se face drop la capabilități și apoi va rula ca utilizatorul shell. Se potrivește perfect cu scenariul original.

Soluția este următoarea: se crează procese sub utilizatorul shell până când se atinge limita `RLIMIT_NPROC`, se omoară procesul de `adb` și se repornește. În acest moment apelul `setuid` va eșua, și astfel se obține acces root prin `adb`.

5.8 Bibliografie

- Joshua J. Drake, Zach Lanier, Collin Mulliner, Pau Oliva Fora, Stephen A. Ridley, and Georg Wicherski. 2014. Android Hacker's Handbook. Wiley Publishing.
- Raphael Spreitzer, Veelasha Moonsamy, Thomas Korak and Stefan Mangard. 2018. Systematic Classification of Side-Channel Attacks: A Case Study for Mobile Devices. IEEE Communications Surveys & Tutorials, vol. 20, no. 1, pp. 465-488.
- Milad Taleby Ahvanooey, Qianmu Li, Mahdi Rabbani and Ahmed Raza Rajput. 2017. A Survery on Smartphones Security: Software Vulnerabilities, Malware and Attacks. International Journal of Advanced Computer Science and Applications, Vol. 8, No. 10, pp. 30-45.

Capitolul 6

Concluzii

Android-ul câștigă din ce în ce mai multă popularitate, în primul rând, datorită faptului că este ușor de folosit și se pot instala un număr mare de aplicații gratuite de pe Google Play Store. Fiind cel mai folosit sistem de operare pentru dispozitive mobile la nivel global face ca securizarea lui folosind mecanisme de protecție eficiente și de încredere să joace un rol extrem de important.

În această carte sunt prezentate mecanismele de protecție și framework-ul de securitate oferit de Android, precum și suprafețele de atac și potențialele vulnerabilități expuse pe un dispozitiv mobil. Pentru început este descris impactul pe care îl aduce mecanismul de sandboxing al proceselor preluat din lumea Linux de către Android. Acest mecanism conduce la izolarea unui proces atât la nivel de acces al componentelor unei aplicații Android, cât și la nivel de acces al datelor. Bineînțeles, comunicarea între procese este posibilă prin intermediul obiectelor de tip Intent și nu numai. Al doilea mecanism de protecție descris este cel reprezentat de permisiuni și de modul în care acestea sunt gestionate de framework-ul de securitate de pe Android. Cele mai relevante sunt permisiunile care au nivelul de protecție dangerous pentru ca acestea sunt, în principiu, singurele permisiuni care oferă acces la resurse/date cu un anumit risc.

În continuare, este descrisă securitatea la nivelul conexiunilor de rețea. Obiectivul principal este ca aceste conexiuni să satisfacă cei trei piloni ai unei conexiuni sigure: autentificare, confidențialitate și integritate a datelor. Autentificarea are rolul de a garanta că o entitate este cine spune ea ca este și este asigurată, de obicei, prin algoritmi de criptare asimetrică. Confidențialitatea garantează că un mesaj nu poate fi descifrat în cazul interceptării lui de către o entitate terță și aceasta este asigurată prin algoritmi de criptare simetrică. Ultimul pilon este reprezentat de integritatea datelor care validează ca datele nu au fost modificate în tranzit de către un atacator. Integritatea este asigurată prin algoritmi de hashing. Acest ansamblu de caracteristici este garantat prin folosirea protocolului HTTPS concomitent cu un certificat digital valid. Android oferă în acest sens un framework criptografic JCA care oferă toate aceste facilități prin intermediul provider-ului criptografic Android OpenSSL.

Android oferă utilizatorilor posibilitatea de a actualiza sistemul de operare folosind update-uri oficiale, precum și acela de a instala versiuni custom ale sistemului de operare numai dacă s-a făcut unlock la bootloader în prealabil. Aceasta ultimă facilitate este folosită de obicei pentru a obține acces de tip root pe dispozitivul mobil și vine la

pachet cu un set de riscuri: orice proces de tip root o sa aibă acces la orice tip de resurse și date. Pentru a minimiza impactul pe care o actualizare de sistem ar putea sa o aibă și pentru a garanta integritatea partițiilor, Android oferă mecanismul numit verified boot. Acesta intră în funcțiune doar pe dispozitivele mobile care au bootloader-ul locked și are rolul de a valida ca o partiție de tip read-only nu a fost modificată în mod neautorizat.

În cele din urmă, este efectuată o trecere în revistă a suprafețelor de atac de pe un dispozitiv mobil. Acestea sunt în mare parte identice cu suprafețele de atac din lumea desktop-urilor. Spre deosebire de acestea însă, dispozitivele mobile prezintă câteva suprafețe de atac adiționale precum cea reprezentată de interfața celulară de 3G/4G/5G sau anumite atacuri de tip side-channel. Sunt descrise și câteva dintre mecanismele de protecție pe care Google le folosește pentru a garanta că aplicațiile Android care ajung pe Play Store sunt aplicații sigure.