AYI NEDJIMI Consultants

Malwares Mobiles & IA - Rétro-Ingénierie Cross-Platform

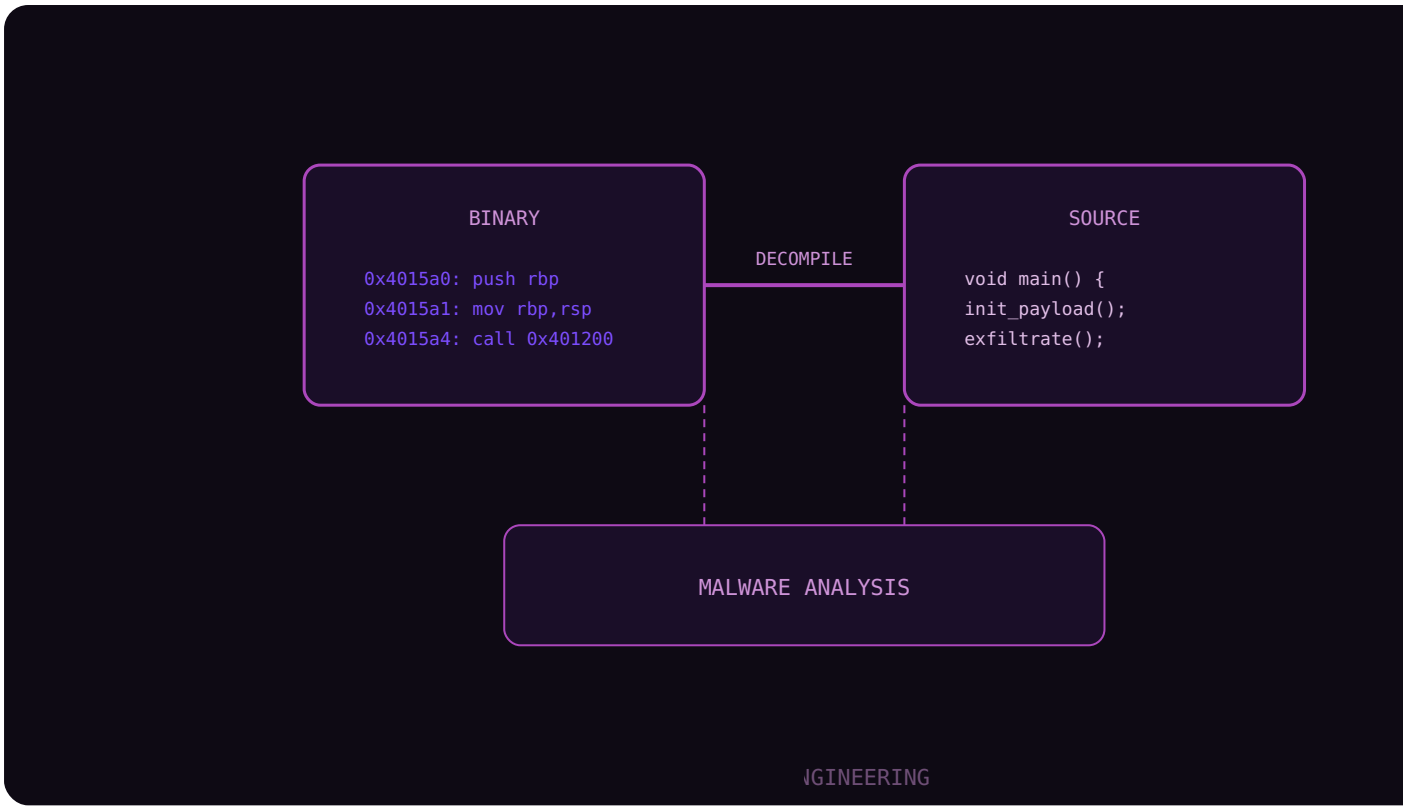
Catégorie : Retro-Ingenierie Lecture : 5 min Publié le : 08/03/2026 Auteur : Ayi NEDJIMI

Rétro-ingénierie des menaces mobiles Android/iOS : Pegasus, Anubis, FluBot. Analyse avec Jadx, Frida, extraction de modèles IA embarqués TFLite.

Malwares Mobiles & IA - Rétro-Ingénierie Cross-Platform constitue un enjeu majeur pour les professionnels de la sécurité informatique et les équipes techniques. Rétro-ingénierie des menaces mobiles Android/iOS : Pegasus, Anubis, FluBot. Analyse avec Jadx, Frida, extraction de modèles IA embarqués TFLite. Ce guide détaillé sur malwares mobiles ia retro ingenierie propose une méthodologie structurée, des outils éprouvés et des recommandations opérationnelles directement applicables. L'objectif est de fournir aux praticiens — consultants, ingénieurs sécurité, administrateurs systèmes — les connaissances et les techniques nécessaires pour aborder ce sujet avec rigueur. Chaque section s'appuie sur des retours d'expérience terrain et intègre les évolutions les plus récentes du domaine. Les recommandations présentées sont adaptées aux environnements d'entreprise et tiennent compte des contraintes opérationnelles réelles.

Avertissement : Les techniques présentées dans cet article sont destinées exclusivement à des fins éducatives et de tests autorisés. Toute utilisation malveillante est illégale et contraire à l'éthique professionnelle.

1. Introduction



Le paysage des menaces mobiles a radicalement évolué. En 2025, **3,9 milliards** de smartphones sont actifs dans le monde, et les malwares mobiles représentent désormais **33% de toutes les infections** détectées (source : Kaspersky Mobile Report 2025). La convergence entre les plateformes desktop et mobile, l'essor du BYOD (Bring Your Own Device), et l'intégration croissante de modèles d'IA embarqués créent une surface d'attaque majeure. Ce guide approfondi examine en detail les aspects fondamentaux et avances de Malwares Mobiles & IA, en proposant une analyse structuree et documentee des enjeux actuels. Les professionnels y trouveront des recommandations concretes, des methodologies eprouvees et des retours d'experience terrain directement applicables en environnement de production. L'analyse integre les dernieres evolutions technologiques, les tendances emergentes du secteur et les meilleures pratiques recommandees par les experts du domaine.

> **Points cles :**
>
> - 1. Introduction
> - 2. Architecture Android Internals pour la RE
> - 3. Décompilation et Analyse Statique Android
> - 4. Instrumentation Dynamique avec Frida
> - 5. Analyse iOS et Jailbreak

Les malwares mobiles modernes ne se limitent plus aux simples trojans bancaires. Des outils comme **Pegasus** (NSO Group) exploitent des chaînes de vulnérabilités zero-click pour compromettre un appareil sans aucune interaction utilisateur. Les banking trojans comme

Anubis utilisent les services d'accessibilité Android pour voler des identifiants en temps réel. Et une nouvelle génération de malwares embarque des modèles **TensorFlow Lite** pour l'évasion et la classification comportementale sur l'appareil. Pour approfondir, consultez notre article sur Ghidra Reverse Engineering Guide Debutant. Pour plus d'informations, consultez les ressources de ANSSI.

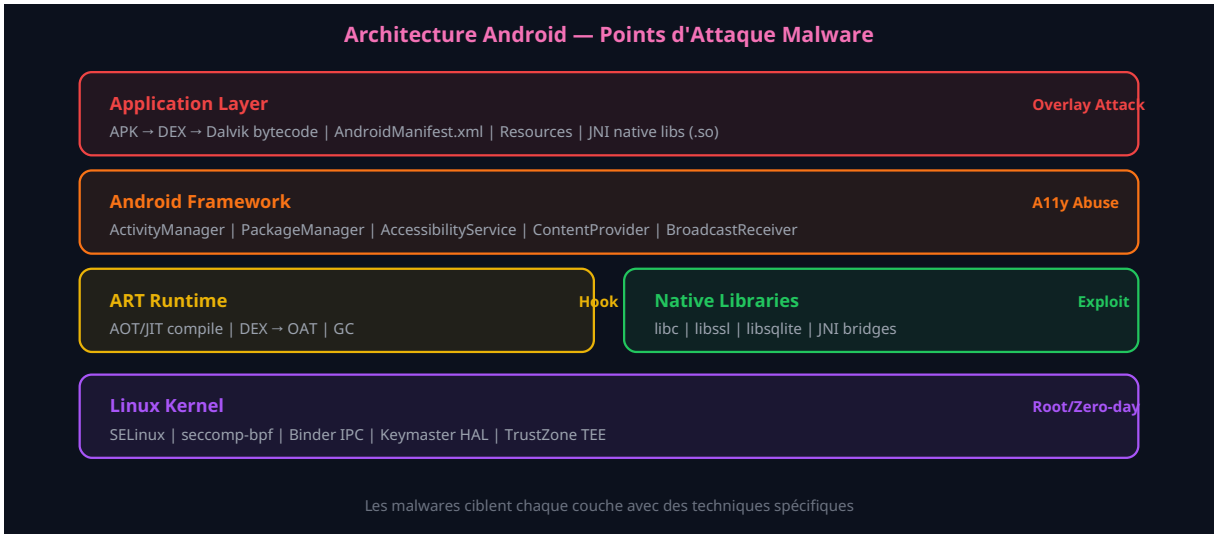
Cet article fournit un guide technique complet pour la rétro-ingénierie de ces menaces mobiles, avec un focus sur les outils, les méthodologies, et les techniques d'analyse statique et dynamique adaptées aux écosystèmes Android et iOS. Pour approfondir, consultez notre article sur Fileless Malware Analyse Detection Memoire. Pour plus d'informations, consultez les ressources de MITRE ATT&CK.

Notre avis d'expert
L'analyse de malware est un art qui requiert patience et méthodologie. Chaque échantillon raconte une histoire — les techniques d'obfuscation utilisées, les C2 contactés, les mécanismes de persistance déployés. Décoder cette histoire est essentiel pour construire des défenses efficaces.

Disposez-vous en interne des compétences de rétro-ingénierie nécessaires pour analyser un malware ciblant votre organisation ?

2. Architecture Android Internals pour la RE

La rétro-ingénierie de malwares Android nécessite une compréhension approfondie de l'architecture interne de la plateforme.



2.1 Structure d'un APK

Un fichier APK est une archive ZIP contenant :

```
# Analyse de la structure d'un APK suspect
unzip -l suspect.apk

# Structure typique :
# AndroidManifest.xml    → Permissions, composants, activités
# classes.dex            → Bytecode Dalvik (code Java/Kotlin)
# classes2.dex           → Multidex overflow
# lib/                   → Bibliothèques natives par architecture
#   ├── armeabi-v7a/     →   libpayload.so (ARM 32-bit)
#   ├── arm64-v8a/       →   libpayload.so (ARM 64-bit)
#   └── x86_64/          →   libpayload.so (émulateur)
# res/                   → Ressources (layouts, images)
# assets/                → Fichiers arbitraires (configs, modèles ML)
# META-INF/              → Signatures et certificat

# Extraction des permissions (aapt)
aapt dump permissions suspect.apk

# Vérification du certificat de signature
apksigner verify --print-certs suspect.apk
keytool -printcert -jarfile suspect.apk

# Hash du certificat (fingerprinting)
apksigner verify --print-certs suspect.apk 2>/dev/null | \
    grep SHA-256 | head -1
```

2.2 Format DEX et Bytecode Dalvik

```
"""
Analyseur de fichier DEX minimal
Extraction des classes et méthodes pour triage rapide
"""
import struct

class DexParser:
    def __init__(self, data):
        self.data = data
        self.header = self._parse_header()

    def _parse_header(self):
        """Parse le header DEX (112 octets)"""
        h = {}
        h['magic'] = self.data[:8]  # dex\n035\0 ou dex\n039\0
        h['checksum'] = struct.unpack('
```

3. Décompilation et Analyse Statique Android

3.1 Pipeline d'analyse avec Jadx et apktool

```
#!/bin/bash
# Pipeline d'analyse statique complète pour APK Android
# Usage : ./analyze_apk.sh suspect.apk

APK=$1
WORKDIR="analysis_$(basename $APK .apk)"
mkdir -p "$WORKDIR"/{jadx,apktool,strings,yara}

echo "[*] Hashes"
sha256sum "$APK" | tee "$WORKDIR/hash.txt"
md5sum "$APK" >> "$WORKDIR/hash.txt"

echo "[*] Décompilation Jadx (Java source)"
jadx --deobf --show-bad-code \
     --output-dir "$WORKDIR/jadx" \
     "$APK" 2>&1 | tee "$WORKDIR/jadx.log"

echo "[*] Désassemblage apktool (Smali + resources)"
apktool d -f -o "$WORKDIR/apktool" "$APK" 2>&1 | \
    tee "$WORKDIR/apktool.log"

echo "[*] Extraction du Manifest"
cp "$WORKDIR/apktool/AndroidManifest.xml" "$WORKDIR/"

echo "[*] Permissions dangereuses"
grep -oP 'android:name="\K[^"]+' "$WORKDIR/AndroidManifest.xml" | \
    grep -iE "SMS|CALL|CAMERA|RECORD|LOCATION|CONTACTS|ACCESSIBILITY|ADMIN" | \
    tee "$WORKDIR/dangerous_permissions.txt"

echo "[*] Receivers et Services"
grep -E "(receiver|service)" "$WORKDIR/AndroidManifest.xml" | \
    grep -oP 'android:name="\K[^"]+' | \
    tee "$WORKDIR/components.txt"

echo "[*] Recherche de strings suspects"
grep -rn "http://\|https://\|DexClassLoader\|Runtime.exec\|getDeviceId" \
    "$WORKDIR/jadx/" 2>/dev/null | head -50 > "$WORKDIR/strings/urls_exec.txt"

echo "[*] Recherche de libs natives"
find "$WORKDIR" -name "*.so" -exec file {} \; | \
    tee "$WORKDIR/native_libs.txt"

echo "[*] Recherche de fichiers chiffrés/encodés dans assets"
find "$WORKDIR/apktool/assets" -type f -exec file {} \; 2>/dev/null | \
    tee "$WORKDIR/assets_analysis.txt"

echo "[*] Détection d'obfuscation"
TOTAL_CLASSES=$(find "$WORKDIR/jadx" -name "*.java" | wc -l)
SHORT_NAMES=$(find "$WORKDIR/jadx" -name "?.java" -o -name "??.java" | wc -l)
echo "Classes totales: $TOTAL_CLASSES"
echo "Noms courts (obfusqués): $SHORT_NAMES"
echo "Ratio obfuscation: $(( SHORT_NAMES * 100 / TOTAL_CLASSES ))%"

echo "[+] Analyse terminée → $WORKDIR/"
```

3.2 Détection et contournement de ProGuard/R8/DexGuard

```
"""
Script de détection et classification de l'obfuscation Android
Identifie ProGuard, R8, DexGuard, et obfuscateurs custom
"""
import os
import re
from collections import Counter

class ObfuscationDetector:
    def __init__(self, jadx_output_dir):
        self.src_dir = jadx_output_dir
        self.java_files = []
        self._scan_files()

    def _scan_files(self):
        for root, dirs, files in os.walk(self.src_dir):
            for f in files:
                if f.endswith('.java'):
                    self.java_files.append(os.path.join(root, f))

    def detect_obfuscator(self):
        """Identifier le type d'obfuscateur utilisé"""
        results = {
            'proguard': 0, 'r8': 0, 'dexguard': 0,
            'allatori': 0, 'custom': 0
        }

        class_names = [os.path.basename(f).replace('.java', '')
                       for f in self.java_files]

        # ProGuard/R8 : classes nommées a, b, c, aa, ab...
        short_names = [n for n in class_names if re.match(r'^[a-z]{1,3}$', n)]
        if len(short_names) > len(class_names) * 0.3:
            results['proguard'] = len(short_names)

        # DexGuard : strings chiffrées avec pattern spécifique
        for f in self.java_files[:100]:  # Sample 100 fichiers
            with open(f, 'r', errors='ignore') as fh:
                content = fh.read()
                # DexGuard string encryption pattern
                if re.search(r'new String\(new byte\[\]\{.*?\}', content):
                    results['dexguard'] += 1
                # Allatori string encryption
                if re.search(r'ALLATORIxDEMO', content):
                    results['allatori'] += 1

        # Déterminer l'obfuscateur principal
        max_score = max(results.values())
        if max_score == 0:
            return "Aucune obfuscation détectée", results

        detected = [k for k, v in results.items() if v == max_score]
        return detected[0].upper(), results

    def extract_encrypted_strings(self):
        """Extraire les strings chiffrées pour analyse"""
        encrypted = []
        for f in self.java_files:
            with open(f, 'r', errors='ignore') as fh:
                content = fh.read()

                # Pattern 1 : byte arrays (DexGuard)
```

```
                for m in re.finditer(
                    r'new byte\[\]\{([\d, -]+)\}', content):
                    bytes_str = m.group(1)
                    try:
                        byte_vals = [int(b.strip()) & 0xFF
                                     for b in bytes_str.split(',')]
                        encrypted.append({
                            'file': f,
                            'type': 'byte_array',
                            'data': bytes(byte_vals),
                            'raw': bytes_str[:80]
                        })
                    except ValueError:
                        pass

                # Pattern 2 : XOR décryptage inline
                for m in re.finditer(
                    r'(\w+)\s*\^\s*(0x[0-9a-fA-F]+|\d+)', content):
                    encrypted.append({
                        'file': f,
                        'type': 'xor',
                        'key': m.group(2),
                        'raw': m.group(0)
                    })

        return encrypted
```

Cas concret
L'analyse du malware Pegasus par le Citizen Lab et Amnesty International a révélé un arsenal d'exploitation zero-click ciblant iOS. La rétro-ingénierie des exploits FORCEDENTRY a montré une utilisation innovante de fichiers PDF malveillants traités par le moteur de rendu d'iMessage, sans aucune interaction de la victime.

4. Instrumentation Dynamique avec Frida

Frida est l'outil d'instrumentation dynamique par excellence pour l'analyse mobile. Il permet d'intercepter les appels de fonctions, de modifier les valeurs de retour, et d'inspecter la mémoire en temps réel, sur Android et iOS. Pour approfondir, consultez notre article sur Reverse Engineering Dotnet Decompilation Analyse.

Mise en pratique

4.1 Scripts Frida pour l'analyse de malwares Android

```
// frida_malware_analysis.js
// Script Frida complet pour l'analyse de malwares Android
// Usage : frida -U -l frida_malware_analysis.js -f com.malware.package

console.log("[*] Malware Analysis Script - Frida");

// === 1. SSL Pinning Bypass ===
Java.perform(function() {
    // Bypass OkHttp CertificatePinner
    try {
        var CertPinner = Java.use("okhttp3.CertificatePinner");
        CertPinner.check.overload("java.lang.String",
            "java.util.List").implementation = function(host, certs) {
            console.log("[SSL] Bypassed pin for: " + host);
            return;
        };
    } catch(e) {}

    // Bypass TrustManagerImpl
    try {
        var TrustManager = Java.use(
            "com.android.org.conscrypt.TrustManagerImpl");
        TrustManager.verifyChain.implementation = function() {
            console.log("[SSL] TrustManager bypassed");
            return arguments[0];
        };
    } catch(e) {}
});

// === 2. Interception des communications C2 ===
Java.perform(function() {
    // Hook HttpURLConnection
    var URL = Java.use("java.net.URL");
    URL.$init.overload("java.lang.String").implementation =
        function(url) {
        console.log("[HTTP] URL: " + url);
        return this.$init(url);
    };

    // Hook SharedPreferences (stockage config)
    var SPEditor = Java.use(
        "android.app.SharedPreferencesImpl$EditorImpl");
    SPEditor.putString.implementation = function(key, value) {
        console.log("[PREF] " + key + " = " + value);
        return this.putString(key, value);
    };
});

// === 3. Détection du vol de données ===
Java.perform(function() {
    // Hook TelephonyManager
    var TelMgr = Java.use("android.telephony.TelephonyManager");

    TelMgr.getDeviceId.overload().implementation = function() {
        var real = this.getDeviceId();
        console.log("[THEFT] getDeviceId() = " + real);
        return "000000000000000";  // Retourner un faux IMEI
    };

    TelMgr.getSubscriberId.overload().implementation = function() {
        var real = this.getSubscriberId();
        console.log("[THEFT] getSubscriberId() = " + real);
```

```
        return "000000000000000";
    };

    // Hook ContactsContract (vol de contacts)
    var ContentRes = Java.use("android.content.ContentResolver");
    ContentRes.query.overload(
        "android.net.Uri", "[Ljava.lang.String;",
        "java.lang.String", "[Ljava.lang.String;",
        "java.lang.String"
    ).implementation = function(uri, proj, sel, selArgs, sort) {
        console.log("[THEFT] ContentResolver.query: " + uri);
        return this.query(uri, proj, sel, selArgs, sort);
    };
});

// === 4. Hook AccessibilityService (overlay attacks) ===
Java.perform(function() {
    try {
        var AccService = Java.use(
            "android.accessibilityservice.AccessibilityService");
        AccService.onAccessibilityEvent.implementation =
            function(event) {
            console.log("[A11Y] Event: " + event.getEventType() +
                        " Package: " + event.getPackageName());
            this.onAccessibilityEvent(event);
        };
    } catch(e) {}
});

// === 5. Interception SMS ===
Java.perform(function() {
    var SmsManager = Java.use("android.telephony.SmsManager");
    SmsManager.sendTextMessage.overload(
        "java.lang.String", "java.lang.String",
        "java.lang.String", "android.app.PendingIntent",
        "android.app.PendingIntent"
    ).implementation = function(dest, sc, text, sent, deliv) {
        console.log("[SMS] TO: " + dest + " MSG: " + text);
        // Bloquer l'envoi vers des numéros premium
        if (dest && dest.length > 5) {
            console.log("[SMS] BLOCKED premium SMS!");
            return;
        }
        this.sendTextMessage(dest, sc, text, sent, deliv);
    };
});

// === 6. Anti-Root Detection Bypass ===
Java.perform(function() {
    var Runtime = Java.use("java.lang.Runtime");
    var origExec = Runtime.exec.overload("java.lang.String");
    origExec.implementation = function(cmd) {
        if (cmd.indexOf("su") !== -1 ||
            cmd.indexOf("which") !== -1) {
            console.log("[ROOT] Blocked exec: " + cmd);
            throw Java.use("java.io.IOException")
                .$new("Permission denied");
        }
        return origExec.call(this, cmd);
    };

    var File = Java.use("java.io.File");
```

```
    File.exists.implementation = function() {
        var path = this.getAbsolutePath();
        var rootPaths = ["/system/xbin/su", "/system/bin/su",
                         "/sbin/su", "/data/local/bin/su",
                         "/su/bin/su", "/system/app/Superuser.apk"];
        if (rootPaths.indexOf(path) !== -1) {
            console.log("[ROOT] Hidden: " + path);
            return false;
        }
        return this.exists();
    };
});
```

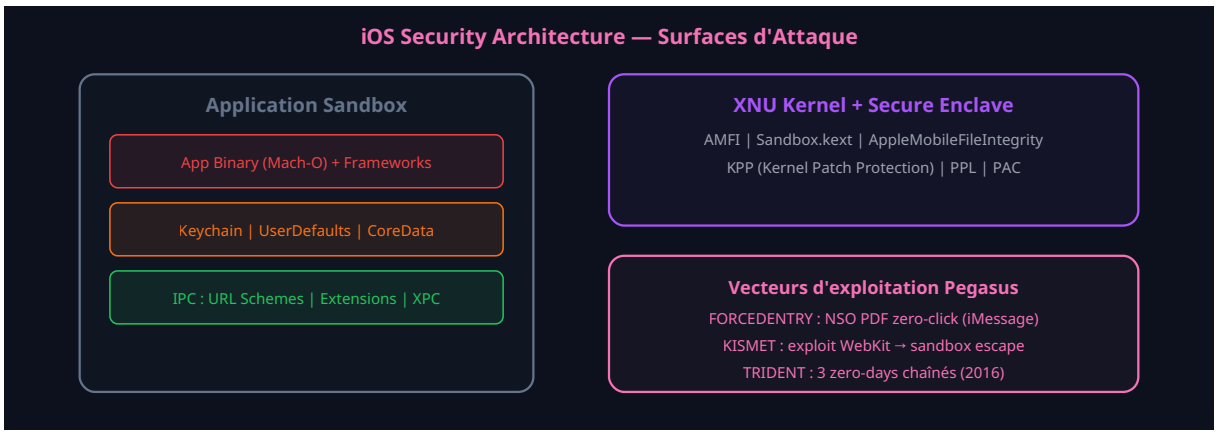
Savez-vous identifier les techniques d'anti-analyse utilisées par les malwares modernes ?

5. Analyse iOS et Jailbreak

La rétro-ingénierie sur iOS est plus complexe en raison de l'écosystème fermé d'Apple. L'analyse nécessite souvent un appareil jailbreaké ou l'utilisation de Corellium (émulateur iOS cloud). Pour approfondir, consultez notre article sur Anti Retro Ingenierie Apt.

5.1 Analyse statique d'un binaire Mach-O

```
# Analyse d'un binaire iOS (Mach-O)
# Pré-requis : appareil jailbreaké avec OpenSSH

# 1. Récupérer le binaire depuis l'appareil
scp root@iphone:/var/containers/Bundle/Application/*/App.app/App ./

# 2. Vérifier le format
file App
# App: Mach-O universal binary with 2 architectures:
#   arm64, arm64e

# 3. Lister les segments et sections
otool -l App | grep -A5 "sectname\|segname"

# 4. Extraire les classes Objective-C
class-dump -H -o headers/ App

# 5. Vérifier le chiffrement FairPlay (App Store)
otool -l App | grep -A4 "LC_ENCRYPTION_INFO"
# cryptoff = 16384  cryptsize = 12345678  cryptid = 1
# cryptid = 1 signifie chiffré → il faut dumper depuis la mémoire

# 6. Dump depuis la mémoire (appareil jailbreaké)
# Utiliser frida-ios-dump ou flexdecrypt
frida-ios-dump -u -o decrypted.ipa com.app.target

# 7. Décompilation avec Hopper ou IDA
# Chercher les sélecteurs Objective-C suspects
grep -r "sendSMS\|recordAudio\|readContacts\|uploadFile" headers/

# 8. Chercher les entitlements
codesign -d --entitlements :- App 2>&1 | \
    grep -E "keychain|aps-environment|com.apple"
```



6. Étude de Cas : Pegasus (NSO Group)

Pegasus est le spyware commercial le plus abouti jamais documenté. Développé par la société israélienne NSO Group, il cible à la fois iOS et Android. L'analyse de Citizen Lab, Amnesty International et Google Project Zero a révélé une ingénierie d'exploitation majeur.

6.1 Chaîne d'exploitation FORCEDENTRY (zero-click)

L'exploit **FORCEDENTRY** (CVE-2021-30860) exploite une vulnérabilité dans le parseur PDF d'iMessage (CoreGraphics/ImageIO). L'attaque est entièrement **zero-click** : la victime reçoit un message invisible contenant un PDF malformé qui déclenche l'exploitation sans aucune interaction.

```
"""
Détection de Pegasus avec MVT (Mobile Verification Toolkit)
Outil développé par Amnesty International
"""
import subprocess
import json
import os

class PegasusDetector:
    """Pipeline de détection Pegasus sur backup iOS"""

    # IOCs Pegasus connus (Amnesty International + Citizen Lab)
    PEGASUS_DOMAINS = [
        "*.amazonaws.com",       # Infrastructure AWS
        "*.cloudfront.net",      # CDN distribution
        "pc4ba2kq.get1tn0w.free247downloads.com",
        "php78mp5.opposedarrangement.net",
        "*.feb-allow.com",
        "*.bafrfrede.gq",
    ]

    PEGASUS_PROCESSES = [
        "bh",              # Bridgehead
        "roleaboutd",      # Persistence daemon
        "pcaborede",       # Exfiltration
        "RollingStone",    # Implant module
        "liaborede",       # Network module
    ]

    PEGASUS_PATHS = [
        "/private/var/db/com.apple.xpc.roleaccountd.staging/",
        "/private/var/tmp/cfurl_cache_snapshot/",
        "/private/var/tmp/BNRSpotlightCorrespondents/",
    ]

    def scan_backup(self, backup_path):
        """Analyser un backup iTunes pour traces Pegasus"""
        results = {
            'suspicious_processes': [],
            'suspicious_domains': [],
            'suspicious_files': [],
            'sms_exploitation': [],
        }

        # 1. Analyser les SMS/iMessage (DataUsage.sqlite)
        datausage = os.path.join(backup_path,
            "HomeDomain/Library/Databases/DataUsage.sqlite")
        if os.path.exists(datausage):
            # Chercher les process names suspects
            cmd = f"sqlite3 '{datausage}' " \
                  f"\"SELECT ZIDENTIFIER FROM ZPROCESS;\""
            output = subprocess.check_output(cmd, shell=True,
                                             text=True)
            for proc in self.PEGASUS_PROCESSES:
                if proc in output:
                    results['suspicious_processes'].append(proc)

        # 2. Chercher les domaines dans le cache Safari
        safari_history = os.path.join(backup_path,
            "HomeDomain/Library/Safari/History.db")
        if os.path.exists(safari_history):
            cmd = f"sqlite3 '{safari_history}' " \
```

```
                  f"\"SELECT url FROM history_items;\""
            output = subprocess.check_output(cmd, shell=True,
                                             text=True)
            for domain in self.PEGASUS_DOMAINS:
                pattern = domain.replace("*.", "")
                if pattern in output:
                    results['suspicious_domains'].append(domain)

        return results

    def run_mvt(self, backup_path, output_dir):
        """Exécuter MVT complet"""
        cmd = [
            "mvt-ios", "check-backup",
            "--output", output_dir,
            "--indicators", "pegasus_indicators.stix2",
            backup_path
        ]
        result = subprocess.run(cmd, capture_output=True, text=True)
        return result.stdout
```

6.2 IOCs Pegasus

```
# IOCs Pegasus (NSO Group) - mise à jour 2025
# Source : Amnesty International, Citizen Lab, Google TAG

# Installation MVT
pip install mvt

# Télécharger les indicateurs STIX2
wget https://raw.githubusercontent.com/AmnestyTech/\
investigations/master/2021-07-18_nso/pegasus.stix2

# Analyse d'un backup iOS
mvt-ios check-backup \
    --indicators pegasus.stix2 \
    --output results/ \
    ~/iTunes_Backup/

# Analyse d'un dump Android
mvt-android check-androidqf \
    --indicators pegasus.stix2 \
    --output results/ \
    android_dump/

# Vérifier les résultats
cat results/timeline.csv | head -20
cat results/sms_suspicious.json | python3 -m json.tool
```

7. Étude de Cas : Anubis / FluBot

Anubis (alias BankBot) est un banking trojan Android actif depuis 2017, distribué via le Google Play Store déguisé en applications utilitaires. **FluBot** (alias Cabassous) se propage principalement par SMS contenant des liens de faux suivi de colis.

7.1 Mécanisme d'overlay attack Anubis

```
// Reconstruction du mécanisme d'overlay attack d'Anubis
// Basé sur la décompilation Jadx d'un sample réel

// L'overlay attack fonctionne ainsi :
// 1. Le malware surveille les apps bancaires lancées
// 2. Quand une app cible est détectée, il affiche un
//    overlay (fausse page de login) par-dessus
// 3. L'utilisateur saisit ses identifiants dans le faux formulaire
// 4. Les credentials sont exfiltrés vers le C2

// Service d'accessibilité malveillant (simplifié)
public class BotAccessibilityService
    extends AccessibilityService {

    // Liste des apps bancaires ciblées
    private static final String[] TARGET_APPS = {
        "com.bnpp.fr.mobilebanking",      // BNP Paribas
        "fr.creditagricole.androidapp",     // Crédit Agricole
        "com.caisseepargne.android",        // Caisse d'Épargne
        "fr.laposte.lapostemobile",         // La Banque Postale
        "com.boursorama.android.clients",   // Boursorama
        "com.paypal.android.p2pmobile",     // PayPal
    };

    @Override
    public void onAccessibilityEvent(AccessibilityEvent event) {
        String packageName = event.getPackageName().toString();

        // Vérifier si l'app en foreground est une cible
        for (String target : TARGET_APPS) {
            if (packageName.equals(target)) {
                // Lancer l'overlay (injection web)
                launchOverlay(target);
                break;
            }
        }

        // Keylogger via AccessibilityEvent
        if (event.getEventType() ==
            AccessibilityEvent.TYPE_VIEW_TEXT_CHANGED) {
            String text = event.getText().toString();
            sendToC2("keylog", packageName + ": " + text);
        }
    }

    private void launchOverlay(String targetApp) {
        // Charger l'injection HTML depuis le C2
        // Chaque banque a son propre template de phishing
        Intent intent = new Intent(this, OverlayActivity.class);
        intent.putExtra("target", targetApp);
        intent.addFlags(Intent.FLAG_ACTIVITY_NEW_TASK);
        startActivity(intent);
    }

    private void sendToC2(String type, String data) {
        // Communication C2 via HTTP POST
        // URL chiffrée en RC4 dans les SharedPreferences
        // Format : {"type":"keylog","data":"...","bot_id":"..."}
    }
}
```

7.2 Extraction de la config C2 Anubis

```
"""
Extracteur de configuration C2 pour Anubis/BankBot
Le C2 est souvent stocké chiffré dans les SharedPreferences
ou dans un fichier assets chiffré
"""
import base64
import json
from Crypto.Cipher import ARC4  # RC4

def extract_anubis_c2(apk_dir):
    """
    Anubis stocke ses C2 de plusieurs manières :
    1. Base64 dans strings.xml (anciennes versions)
    2. RC4 dans assets/config.dat
    3. Telegram bot API comme fallback C2
    4. Twitter/Pastebin comme dead drop resolver
    """
    c2_urls = []

    # Méthode 1 : strings.xml encodées
    strings_file = f"{apk_dir}/res/values/strings.xml"
    try:
        with open(strings_file, 'r') as f:
            content = f.read()
            # Chercher les base64 dans les strings
            import re
            for match in re.finditer(
                r'>([A-Za-z0-9+/=]{20,})<', content):
                try:
                    decoded = base64.b64decode(match.group(1))
                    if b'http' in decoded or b'.' in decoded:
                        c2_urls.append(('strings.xml',
                                        decoded.decode('utf-8',
                                                       errors='ignore')))
                except Exception:
                    pass
    except FileNotFoundError:
        pass

    # Méthode 2 : fichier assets chiffré
    config_file = f"{apk_dir}/assets/config.dat"
    try:
        with open(config_file, 'rb') as f:
            encrypted = f.read()

        # Clés RC4 communes dans Anubis
        common_keys = [
            b'jHGFDhkjh67',
            b'botnet2019',
            b'anubis_key',
            b'1234567890',
        ]
        for key in common_keys:
            cipher = ARC4.new(key)
            decrypted = cipher.decrypt(encrypted)
            if b'http' in decrypted:
                try:
                    config = json.loads(decrypted)
                    c2_urls.append(('config.dat', config))
                except json.JSONDecodeError:
                    # Extraire les URLs brutes
                    urls = re.findall(
```

```
                        rb'https?://[^\s"\']+', decrypted)
                    for url in urls:
                        c2_urls.append(('config.dat',
                                        url.decode('utf-8',
                                                   errors='ignore')))
                break
    except FileNotFoundError:
        pass

    # Méthode 3 : Telegram bot fallback
    for java_file_path in _find_java_files(apk_dir):
        with open(java_file_path, 'r', errors='ignore') as f:
            content = f.read()
            # Token Telegram bot
            tg_match = re.search(
                r'(\d{8,10}:[A-Za-z0-9_-]{35})', content)
            if tg_match:
                c2_urls.append(('telegram_bot',
                                tg_match.group(1)))
            # Chat ID
            chat_match = re.search(
                r'chat_id["\s:=]+(-?\d{6,})', content)
            if chat_match:
                c2_urls.append(('telegram_chat',
                                chat_match.group(1)))

    return c2_urls

def _find_java_files(directory):
    import os
    for root, dirs, files in os.walk(directory):
        for f in files:
            if f.endswith('.java'):
                yield os.path.join(root, f)
```

8. IA Embarquée dans les Malwares Mobiles

Une tendance émergente et inquiétante : l'intégration de modèles de **machine learning embarqués** (TensorFlow Lite, CoreML) directement dans les malwares mobiles. Ces modèles servent à l'évasion, à la classification des cibles, et à l'automatisation des attaques.

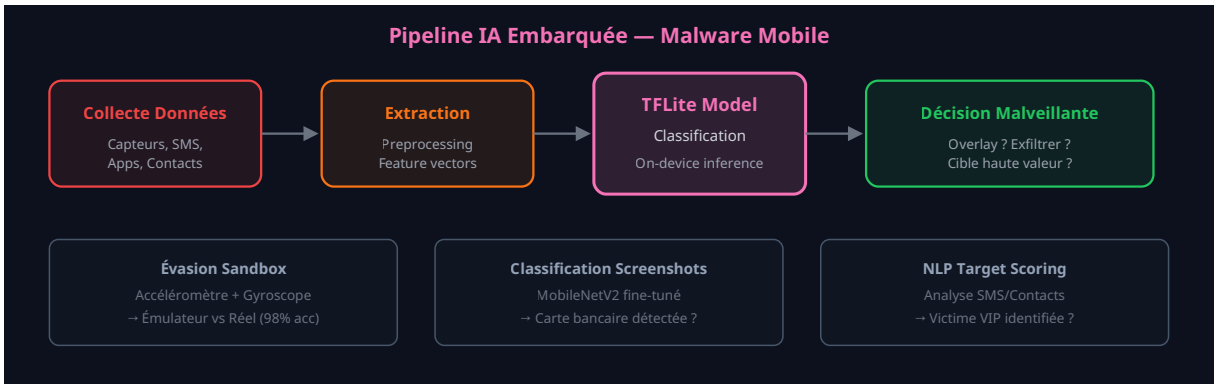
8.1 Cas d'usage malveillants de l'IA embarquée

- **Évasion dynamique** : un modèle TFLite classifie l'environnement (émulateur vs appareil réel) en analysant les patterns de capteurs (accéléromètre, gyroscope)
- **Ciblage intelligent** : un modèle NLP analyse les SMS/emails pour identifier les victimes à haute valeur (dirigeants, personnalités)
- **Phishing adaptatif** : un modèle génère des overlays personnalisés basés sur les apps installées et l'historique de navigation
- **Exfiltration sélective** : un modèle de classification d'images filtre les captures d'écran pour ne voler que celles contenant des données sensibles (cartes bancaires, documents confidentiels)

8.2 Extraction et analyse de modèles TFLite

```
"""
Extracteur et analyseur de modèles TensorFlow Lite
embarqués dans des APK malveillants
"""
import os
import zipfile
import struct
import json

class TFLiteExtractor:
    """Extraire et analyser les modèles TFLite d'un APK"""

    TFLITE_MAGIC = b'\x18\x00\x00\x00TFL3'

    def __init__(self, apk_path):
        self.apk_path = apk_path
        self.models = []

    def extract_models(self):
        """Chercher les fichiers .tflite dans l'APK"""
        with zipfile.ZipFile(self.apk_path, 'r') as z:
            for name in z.namelist():
                # Fichiers .tflite explicites
                if name.endswith('.tflite') or \
                   name.endswith('.lite'):
                    data = z.read(name)
                    self.models.append({
                        'path': name,
                        'size': len(data),
                        'data': data
                    })

                # Fichiers cachés (extension modifiée)
                elif name.endswith(('.dat', '.bin', '.model',
                                    '.enc', '.cfg')):
                    data = z.read(name)
                    if self._is_tflite(data):
                        self.models.append({
                            'path': name,
                            'size': len(data),
                            'data': data,
                            'hidden': True
                        })

        return self.models

    def _is_tflite(self, data):
        """Vérifier si les données sont un modèle TFLite"""
        if len(data) < 8:
            return False
        # FlatBuffer magic number pour TFLite
        return data[4:8] == b'TFL3'

    def analyze_model(self, model_data):
        """Analyser la structure d'un modèle TFLite"""
        try:
            import tensorflow as tf

            interpreter = tf.lite.Interpreter(
                model_content=model_data)
            interpreter.allocate_tensors()
```

```
            input_details = interpreter.get_input_details()
            output_details = interpreter.get_output_details()

            analysis = {
                'input_shape': [d['shape'].tolist()
                                for d in input_details],
                'input_dtype': [str(d['dtype'])
                                for d in input_details],
                'output_shape': [d['shape'].tolist()
                                 for d in output_details],
                'output_dtype': [str(d['dtype'])
                                 for d in output_details],
                'num_tensors': len(
                    interpreter.get_tensor_details()),
            }

            # Identifier le type de modèle par sa forme
            in_shape = input_details[0]['shape']
            if len(in_shape) == 4:  # [1, H, W, C]
                analysis['model_type'] = 'Image Classification'
                analysis['input_size'] = f"{in_shape[1]}x{in_shape[2]}"
            elif len(in_shape) == 2:  # [1, features]
                analysis['model_type'] = 'Tabular/Sensor'
                analysis['num_features'] = in_shape[1]
            elif len(in_shape) == 3:  # [1, seq_len, features]
                analysis['model_type'] = 'Sequence/NLP'
                analysis['seq_length'] = in_shape[1]

            return analysis

        except Exception as e:
            return {'error': str(e)}

# Usage
# extractor = TFLiteExtractor("suspect.apk")
# models = extractor.extract_models()
# for m in models:
#     print(f"Model: {m['path']} ({m['size']} bytes)")
#     analysis = extractor.analyze_model(m['data'])
#     print(f"  Type: {analysis.get('model_type')}")
#     print(f"  Input: {analysis.get('input_shape')}")
```



9. Détection et Protection

9.1 Analyse automatisée avec YARA pour Android

```
// Règles YARA pour détection de malwares Android
rule Android_BankingTrojan_Anubis {
    meta:
        author = "Ayi NEDJIMI"
        description = "Detects Anubis/BankBot Android malware"
        date = "2026-02-05"

    strings:
        // AccessibilityService abuse
        $a11y1 = "AccessibilityService" ascii
        $a11y2 = "onAccessibilityEvent" ascii
        $a11y3 = "BIND_ACCESSIBILITY_SERVICE" ascii

        // Overlay attack indicators
        $overlay1 = "TYPE_APPLICATION_OVERLAY" ascii
        $overlay2 = "WindowManager$LayoutParams" ascii

        // Banking app targets
        $bank1 = "com.bnpp" ascii
        $bank2 = "creditagricole" ascii
        $bank3 = "boursorama" ascii
        $bank4 = "paypal" ascii

        // C2 communication
        $c2_1 = "sendBot" ascii
        $c2_2 = "getInjects" ascii
        $c2_3 = "/o1o/a1.php" ascii

        // SMS interception
        $sms1 = "SmsReceiver" ascii
        $sms2 = "pdus" ascii
        $sms3 = "getMessageBody" ascii

    condition:
        // Must be a ZIP (APK) file
        uint32(0) == 0x04034B50 and
        2 of ($a11y*) and
        1 of ($overlay*) and
        2 of ($bank*) and
        1 of ($c2_*) and
        2 of ($sms*)
}

rule Android_TFLite_Suspicious {
    meta:
        author = "Ayi NEDJIMI"
        description = "Detects APK with embedded TFLite model"

    strings:
        $tflite_magic = { 18 00 00 00 54 46 4C 33 }
        $tflite_ext = ".tflite" ascii
        $sensor = "SensorManager" ascii
        $accel = "TYPE_ACCELEROMETER" ascii

    condition:
        uint32(0) == 0x04034B50 and
        ($tflite_magic or $tflite_ext) and
        ($sensor or $accel)
}
```

9.2 Analyse de trafic réseau

```
"""
Analyseur de trafic réseau pour malwares mobiles
Capture et analyse les communications C2
"""
from mitmproxy import http
import json
import re
from datetime import datetime

class MalwareTrafficAnalyzer:
    """Addon mitmproxy pour l'analyse de trafic malware"""

    SUSPICIOUS_PATTERNS = [
        r'/gate\.php',           # Panel de C2 classique
        r'/api/bot',             # Bot C2
        r'/o1o/',                # Anubis C2
        r'bot_id=',              # Bot identifier
        r'imei=',                # Device fingerprint
        r'model=',               # Device model
        r'apps=',                # Installed apps list
    ]

    def __init__(self):
        self.log_file = open('malware_traffic.jsonl', 'a')
        self.alerts = []

    def request(self, flow: http.HTTPFlow):
        url = flow.request.pretty_url
        body = flow.request.get_text() or ""

        # Vérifier les patterns suspects
        for pattern in self.SUSPICIOUS_PATTERNS:
            if re.search(pattern, url + body):
                alert = {
                    'timestamp': datetime.now().isoformat(),
                    'url': url,
                    'method': flow.request.method,
                    'pattern': pattern,
                    'body_preview': body[:500],
                    'headers': dict(flow.request.headers),
                }
                self.alerts.append(alert)
                self.log_file.write(json.dumps(alert) + '\n')
                self.log_file.flush()

                print(f"[ALERT] Suspicious: {pattern}")
                print(f"  URL: {url}")
                break

    def response(self, flow: http.HTTPFlow):
        # Capturer les réponses contenant des injections
        content_type = flow.response.headers.get(
            'content-type', '')
        if 'html' in content_type or 'json' in content_type:
            body = flow.response.get_text() or ""
            if any(kw in body.lower() for kw in [
                'inject', 'overlay', 'phishing', 'keylog']):
                print(f"[INJECT] Injection HTML reçue depuis "
                      f"{flow.request.pretty_url}")
```

```
# Usage : mitmproxy -s malware_analyzer.py
addons = [MalwareTrafficAnalyzer()]
```

1. Introduction	2. Architecture Android Internals pour la RE	3. Décompilation et Analyse Statique Android
Implementation	Renforcement de la securite globale	Complexite de mise en oeuvre
Monitoring	Detection proactive des menaces	Ressources necessaires
Conformite	Alignement aux referentiels	Cout de certification

Pour approfondir ce sujet, consultez notre outil open-source malware-analysis-toolkit qui facilite l'analyse automatisée de malwares.

Questions frequentes

Comment mettre en place Malwares Mobiles & IA dans un environnement de production ?

La mise en place de Malwares Mobiles & IA en production necessite une planification rigoureuse, incluant l'evaluation des prerequis techniques, la definition d'une architecture cible, des tests de validation approfondis et un plan de deploiement progressif avec des points de controle a chaque etape.

Pourquoi Malwares Mobiles & IA est-il essentiel pour la securite des systemes d'information ?

Malwares Mobiles & IA constitue un element fondamental de la securite des systemes d'information car il permet de reduire significativement la surface d'attaque, d'ameliorer la detection des menaces et de renforcer la posture globale de securite de l'organisation face aux cybermenaces actuelles.

Quelles sont les bonnes pratiques pour Malwares Mobiles & IA en 2026 ?

Les bonnes pratiques pour Malwares Mobiles & IA en 2026 incluent l'adoption d'une approche Zero Trust, l'automatisation des controles de securite, la mise en place d'une veille continue sur les vulnerabilites et l'integration des recommandations des organismes de reference comme l'ANSSI et le NIST.

Sources et références : MITRE ATT&CK · CERT-FR

10. Conclusion

La rétro-ingénierie des malwares mobiles est un domaine en expansion rapide, porté par trois tendances majeures :

- **Convergence des plateformes** : les malwares cross-platform (React Native, Flutter, Kotlin Multiplatform) permettent aux attaquants de cibler iOS et Android avec un seul codebase

- **IA embarquée malveillante** : les modèles TFLite/CoreML embarqués dans les APK offrent des capacités d'évasion et de ciblage majeur, le tout sans communication réseau
- **Zero-click exploitation** : Pegasus a démontré qu'un smartphone peut être compromis sans aucune interaction utilisateur, via des vulnérabilités dans les parseurs de médias (iMessage, WhatsApp)

Pour l'analyste, l'écosystème d'outils est mature : **Jadx** et **apktool** pour l'analyse statique, **Frida** pour l'instrumentation dynamique, **MVT** pour la détection de spywares commerciaux, et **mitmproxy** pour l'analyse de trafic. La clé reste la méthodologie : combiner analyse statique et dynamique, automatiser le triage, et maintenir une veille constante sur les nouvelles techniques d'évasion.

Ressources essentielles : OWASP Mobile Security Testing Guide (MSTG), Frida CodeShare (snippets communautaires), MobSF (analyse automatisée), et le dépôt GitHub d'Amnesty International pour les IOCs Pegasus.

Ayi NEDJIMI Consultants — Expert cybersécurité offensive & intelligence artificielle

ayinedjimi-consultants.fr · ayi@ayinedjimi-consultants.fr