

Sicherheit von Betriebssystemen – VO 09: Mobile Security – Vertiefung

Paul Kalauner, Rafael Vrecar

Research Group for Industrial Software (INSO)

<https://www.inso-world.com>



**FACHHOCHSCHULE
WIENER NEUSTADT**
University of Applied Sciences - Austria





Agenda

- Code-Analyse
- Debugging von Anwendungen
- Dynamische Instrumentation
- Code Obfuscation & Encryption
- Environment Checks

Code-Analyse

Grundlagen zur Code-Analyse

- Analyse dient als Grundlage, um Angriffe auf Applikationen durchzuführen
 - es existieren verschiedene Methoden
- zwei verschiedene Arten von Analyse, die sich zumeist untereinander ergänzen
 - statisch (die Applikation wird analysiert, ohne die Applikation laufen zu lassen)
 - dynamisch (die Applikation wird zur Laufzeit analysiert)

Statische Analyse

- dient oftmals als Grundlage für die dynamische Analyse
 - erlaubt die Identifikation von gesuchten Eintrittspunkten
 - gibt oftmals einen Ausblick darauf, was zu erwarten ist

- Zugang zu Source-Code erforderlich
 - Direkt: Source-Code verfügbar (z.B. Open-Source)
 - Indirekt: Dekompilierung

Dekompilierung von Anwendungen

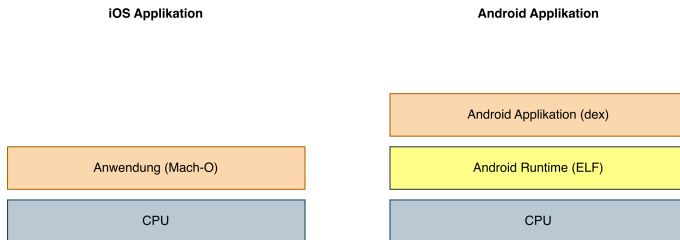
- mobile Applikationen sind ZIP-Bündel
 - Android-Applikationen werden in einer *.apk Datei gebündelt
 - iOS-Applikationen werden in einer *.ipa Datei gebündelt
- die Bündel enthalten Ressourcen (Mediendateien, Bibliotheken, etc.) & die eigentliche Anwendung
 - Android-Anwendungen sind im *.o)dex Dateiformat (Dalvik Executable) gespeichert
 - iOS-Anwendungen sind im Mach-O Dateiformat gespeichert

Dekompilierung von .dex Objekten

- .dex Objekte entstehen durch das Kompilieren von Sprachen wie Kotlin und Java zu Bytecode
- vergleichsweise einfach zu dekompilieren
 - der Bytecode ist sehr „ähnlich“ dem Originalcode
- gängige Disassembler & Decompiler sind
 - apktool (deassembliert Ressourcen aber dekompiliert sie nicht)
 - jadx (deassembliert und dekompiliert Ressourcen)
 - dex2jar (deassembliert .dex Objekte in .class Objekte)
 - ▶ kann zum Beispiel mit JD-GUI angesehen werden

Maschinencode vs. Bytecode

- Bytecode muss von der VM in Maschinencode umgewandelt werden, bevor er ausgeführt werden kann



Dekompilierung von Maschinencode

- Mach-O, geteilte Bibliotheken (*.dylib, *.so) sind Maschinencode
- im Gegensatz zu Bytecode plattformabhängig
 - spezielle Instruktionen
 - unterschiedliche Hardware
- meistens hochoptimiert und daher keine Meta-Informationen wie beim Bytecode

Gängige Disassembler & Decompiler

- ghidra (deassembliert & dekompileert)
- radare (deassembliert & kann mit Erweiterungen auch dekompileieren)
 - iaito ist eine GUI für radare
- IDA Pro (deassembliert & dekompileiert)
- Hopper (deassembliert)

Forensik als statische Analyse 1/2

- oftmals ist nicht nur spannend wie, sondern auch welche Daten verarbeitet werden
- die Daten werden entweder persistent im Dateisystem gespeichert oder temporär im Arbeitsspeicher hinterlegt
- auch sensible Daten müssen zumindest temporär gespeichert werden, um sie zu verarbeiten
 - Speicherabbilder (müssen zur Laufzeit gemacht werden) erlauben eine Momentaufnahme der verarbeiteten Daten zu einem gewissen Zeitpunkt
 - Speicherabbilder können mit fridump (auf allen Plattformen mit **frida** Unterstützung) oder LiME (nur auf Linux) erstellt werden

Forensik als statische Analyse 2/2

- auch kryptographische Schlüssel müssen – teilweise persistent – abgelegt werden
 - bei unsicherer Speicherung einfach zu finden

Dynamische Analyse

- bei der dynamischen Analyse werden die Applikationen zur Laufzeit analysiert
- kein Source-Code erforderlich
- oftmals laufen gewisse Applikationen nicht auf Plattformen, die zum Reverse Engineering verwendet werden
 - Mechanismen zur Erkennung von Geräten, die gerootet oder gejailbreakt wurden
 - Mechanismen zur Erkennung von Emulatoren
 - Mechanismen zur Erkennung von Tools zur dynamischen Analyse

Diagnostische Anwendungen zur dynamischen Analyse

- einfachste Form der dynamischen Analyse
- keine direkte Manipulation von Daten
- Verwendung von existierender “Debugging Infrastruktur”
 - durchsuchen von Logs
 - Mitverfolgung von abgesetzten Systemcalls
 - ▶ iostace
 - ▶ strace
 - ▶ jtrace

Debugging von Anwendungen

Debugging als dynamische Analyse – Grundlagen

- Debugging ermöglicht normalerweise das Nachvollziehen von Fehlern im Programmablauf
- gängige Debugger sind
 - gdb (Linux und macOS)
 - lldb (plattformunabhängig)

Debugging als dynamische Analyse – Breakpoints

- Breakpoints erlauben, das Programm an einer gewissen Stelle zu stoppen
- Werte können ausgelesen werden
- Werte können auch neu gesetzt werden
- erlaubt das Hinzufügen von fremden Code
 - eine einfache Form der dynamischen Instrumentation
- das Setzen von Breakpoints verlangsamt den Ablauf eines Programmes meist deutlich

Einschränkungen des Debuggings

- das Debugging ist meist nur während der Entwicklung relevant
 - beim Entwickeln enthalten die Anwendung meistens zusätzliche Debug Informationen
 - ▶ Namen von Funktionen und Variablen
 - ▶ die zusätzlichen Informationen ermöglichen dem Debugger, die Codestelle zu referenzieren
- Debugger sind oft auf bestimmte Sprachen eingeschränkt:
 - gdb und lldb für Maschinencode
 - Java Debugger nur für JVM-basierte Sprachen (Groovy, Kotlin, Java, etc.)

Dynamische Instrumentation

Dynamische Instrumentation von Applikationen

- basiert auf der Idee von Debuggern
- das Werkzeug für die dynamische Instrumentation hängt sich zu einer App zur Laufzeit dazu
- modifiziert die Instruktionen eines Programms während der Laufzeit
- erlaubt so die Ausführung von zusätzlichem Code ohne die zugrundeliegende Applikationen persistent zu ändern

frida – das Universal Tool

- die Entwicklung erfolgt seit 2010 (siehe GitHub)
 - von Ole André V. Ravnås und Håvard Sørbø geschaffen, um ihnen beim Reverse Engineering zu helfen
- zu einem **DER** Tools für das Reverse Engineering geworden
- ermöglicht die dynamische Instrumentation von Applikationen auf Android, iOS, Linux, macOS, Windows
- in verschiedenen Sprachen geschrieben
- (offizielle) APIs
 - JavaScript/TypeScript API (am besten dokumentiert)
 - C API
 - etc.

Reverse Engineering mit frida 1/2

- frida erlaubt die Veränderung von so ziemlich allen Bestandteilen einer Applikation zur Laufzeit
 - die Werte von Variablen können geändert werden
 - Funktionen können komplett umgeschrieben werden
 - Rückgabewerte können manipuliert werden
 - etc.
- wird dadurch ermöglicht, dass sich das frida Framework bei einer Applikation „dazuhängt“
 - genaue Details können der offiziellen Dokumentation & den Präsentationen von Ole André V. Ravnås entnommen werden

Reverse Engineering mit frida 2/2

- frida ermöglicht u.a. das Mitverfolgen von allen Instruktionen (Stalker) aber auch einzelnen Funktionen (frida-trace)
- das Mitverfolgen aller Instruktionen ist selten aufschlussreich
- die statische Analyse liefert hier meistens Informationen zu wichtigen (Eintritts)punkten

Code Obfuscation & Encryption

Charakteristiken von Code Obfuscation/Encryption

- **Code Obfuscation:** Veränderung von Code ohne Beeinflussung des vorgesehenen Verhaltens
- **Code Encryption:** Verschlüsselung von Code, welcher während Laufzeit entschlüsselt und nachgeladen wird
- Lesbarkeit des Codes (nach Dekompilierung) wird dadurch deutlich verringert
- **Kein** 100%-iger Schutz gegen Dekompilierung, erhöht lediglich die dafür notwendigen Ressourcen

Motivation

- Erschwerung von Reverse Engineering
- Schutz intellektuellem Eigentums
- Schutz vor Piraterie und Repackaging
- Insbesondere unter Android relevant, da Bytecode verhältnismäßig einfach dekompilierbar ist
- Allerdings: häufig auch von Malware-Entwickler:innen eingesetzt, um Erkennung und Analyse von schadhaften Programmen zu erschweren

Code Obfuscation Techniken Übersicht

- Identifier Renaming
- Control Flow Obfuscation
 - Dead Code Injection
 - Code Reordering
 - Erweiterungen durch Schleifen
 - Method Inlining
 - Control Flow Flattening
- Data Obfuscation
- Reflection-based Obfuscation

Identifizier Renaming

- Entwickler verwenden im Normalfall sinnvolle Bezeichnungen für Code-Elemente
⇒ auch für Reverse Engineering hilfreich
- Zuweisung bedeutungsloser Namen an Variablen, Klassen, Methoden
- Meist lexikalische Sequenzen oder Permutationen
- Standardmäßig von APK Optimizern wie ProGuard/R8 durchgeführt
 - Verkleinerung
 - Optimierung
 - Grobe Obfuskierung

Identifizier Renaming – Beispiel

```
1 public class ExampleClass {
2     private String someString = "abc";
3     private int someInt = 42;
4
5     public void doSomething() {
6         // ...
7     }
8 }
```

```
1 public class A {
2     private String aaa = "abc";
3     private int aab = 42;
4
5     public void a() {
6         // ...
7     }
8 }
```

Identifizier Renaming in Binärdateien

- Auch in Binärdateien sind oftmals Zusatzinformationen enthalten
 - Funktionsnamen
 - Variablennamen
 - Debug Informationen
- Das Programm **strip** entfernt für den Ablauf irrelevante Code Stellen
- Primär zur Verkleinerung der Dateien aber mit dem Nebeneffekt der Code Obfuscation

Control Flow Obfuscation (CFO)

- Veränderung des Control Flows
- Veränderte Ausführungspfade bei unverändertem Verhalten
- Vielzahl an Möglichkeiten

CFO: Dead Code Injection

- Einfügen von Code, welcher niemals ausgeführt wird oder keine Auswirkungen hat

```
1 for (int i = 0; i < 10; i++) {  
2     System.out.println(i);  
3 }
```

```
1 for (int i = 0; i < 10; i++) {  
2     if (i % 1337 > 10) {  
3         System.exit(0);  
4     }  
5     System.out.println(i);  
6 }
```

CFO: Code Reordering

- Veränderung der Ausführungsreihenfolge von Anweisungen

```
1 x = 0;
2 while (x < maxNum) {
3     i[x] += j[x];
4     x++;
5 }
```

```
1 x = maxNum;
2 while (x > 0) {
3     x--;
4     i[x] += j[x];
5 }
```

(Vergleiche Faruki et al. (2016))

CFO: Erweiterungen durch Schleifen

- Ersetzung von if-Anweisungen durch komplexere Schleifen
- Einführung zusätzlicher Konditionen mit keinerlei Auswirkungen

```
1 for (int i = 0; i < 15; i++) {
2     if (x < 10) {
3         x += 10;
4     }
5     System.out.println(x);
6 }
```

```
1 for (int i = 0; i < 15 i++) {
2     while (x < 10 || x % 20 == 0) {
3         x += 10;
4     }
5     System.out.println(x);
6 }
```

CFO: Method Inlining

- Ersetzung von Methodenaufrufen durch Methodenkörper
- Entfernt prozedurale Abstraktionen von Programmen
- Auch von Optimizern angewandt

```
1 int y = doSomething(42);
2 private int doSomething(int x) {
3     // ...
4     x += 1;
5     return x * 1337;
6 }
```

```
1 int x = 42;
2 // ...
3 x += 1;
4 int y = x * 1337;
```

CFO: Control Flow Flattening

- Verschiebung von Funktionskörpern, Schleifen und if-Anweisungen in einzige Schleife
- Programmablauf durch `switch` oder mehrere if-Anweisungen geregelt
- Blöcke des Control Flow Graphs werden damit auf selbe Ebene gebracht

CFO: Control Flow Flattening – Beispiel

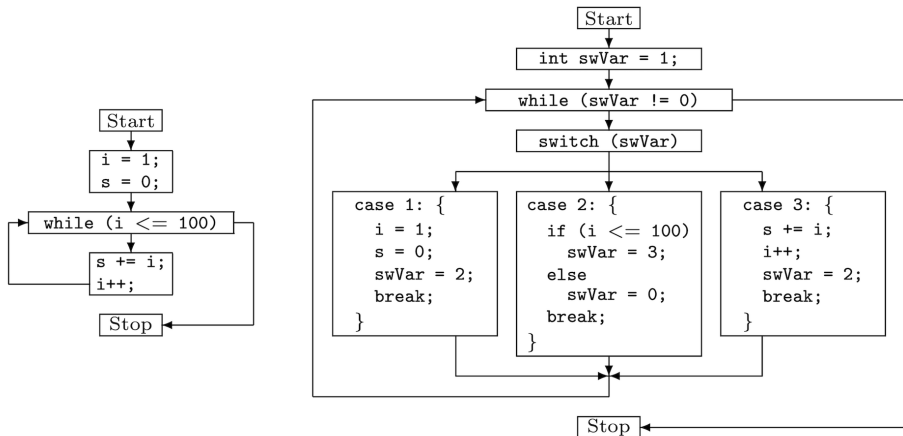
```
1 int i = 1;
2 int s = 0;
3 while (i <= 100) {
4     s += i;
5     i++;
6 }
```

```
1 int swVar = 1;
2 while (swVar != 0) {
3     switch (swVar) {
4         case 1:
5             i = 1; s = 0; swVar = 2;
6             break;
7         case 2:
8             if (i <= 100) swVar = 3;
9             else swVar = 0;
10            break;
11         case 3:
12            s += i; i++; swVar = 2;
13            break;
14     }
15 }
```

(Vergleiche László und Kiss (2009))

CFO: Control Flow Flattening – Beispiel

Control Flow Graph



(Vergleiche László und Kiss (2009))

Data Obfuscation

- Verstecken von Daten/Informationen in Applikationen
- Resource Encryption
 - Verschlüsselung von Ressourcen, die in App mitgepackt sind
 - Injektion von Entschlüsselungslogik in Methoden zum Öffnen von Ressourcen
- String Encryption
 - Ersetzung von Strings durch verschlüsselte Gegenstücke
 - Injektion von Methode(n), die den zugehörigen ursprünglichen String für den gegebenen verschlüsselten String zurückliefern
 - Entschlüsselung bei Zugriff auf String während Laufzeit
 - Invertierbare Ver- und Entschlüsselungsalgorithmen notwendig

String Encryption – Beispiel

```
1 public class ExampleClass {
2     private String secret = "flag{0123456789abcdef}";
3     private String password = "t0p_s3cr3t";
4
5     // ...
6 }
```

```
1 public class ExampleClass {
2     private String secret = InjectedClass.decrypt("vz325jas/E7ft76j$!");
3     private String password = InjectedClass.decrypt("T./$SWksgfzg$%u");
4
5     // ...
6 }
```

Reflection-based Obfuscation

- Java API
- Erlaubt es, Methoden dynamisch aufzurufen
- Verschleierung von Methodenaufrufen durch Entfernung direkter Referenzen
- Referenzen werden dynamisch während Laufzeit über API abgerufen
- Insbesondere in Kombination mit anderen Obfuskerungs-Techniken (z.B. String Encryption) hilfreich

```
1 // ...
2 SomeClass clazz = new SomeClass();
3 clazz.doSomething();
```

```
1 // ...
2 Object c = Class.forName("com.example.SomeClass").newInstance();
3 Method m = c.getClass().getMethod("doSomething");
4 m.invoke(c);
```

Code Encryption

- Code ist verschlüsselt und wird erst zur Laufzeit entschlüsselt und “lauffähig” gemacht
- Funktioniert nicht mit allen Programmiersprachen, da oft die Mechanismen dafür fehlen
 - Viele Sprachen können neuen Code starten, aber nicht anschließend damit interagieren
- Erfordert einen `ClassLoader`-Mechanismus (JVM-basierte Sprachen) bzw. die Möglichkeit den Speicher zu manipulieren (C, C++)

JVM-basierte Code Encryption

- JVM-basierte Sprachen können in der Regel auf den `ClassLoader` zugreifen
 - Ermöglicht es, Code während Laufzeit nachzuladen
- Ein eigens-implementierter `ClassLoader` ermöglicht, dass Objekte entschlüsselt werden bevor sie nachgeladen werden

JVM-basierte Code Encryption – Beispiel

- Als Grundlage hierfür dient ein Projekt von Sefik Serengil

```
1  ...
2  // decrypt the class
3  Cipher decryption = Cipher.getInstance(algorithm);
4  decryption.init(Cipher.DECRYPT_MODE, new SecretKeySpec(key, 0, key.length, algorithm));
5  byte[] decryptedContent = decryption.doFinal(encryptedContent);
6
7  // convert the byte array into a class
8  Class clazz = defineClass(name, decryptedContent, 0, decryptedContent.length);
9
10 // search for the wanted method and call it
11 Method m = clazz.getMethod("main", String[].class);
12 m.invoke(null, new Object[] {null});
13 ...
```

(Vergleiche <https://github.com/serengil/encrypted-class-loader>)

Beispiele für Tools

- Frei verfügbar/Open-Source
 - ProGuard
 - R8
- Kommerziell
 - DexGuard
 - DexProtector
 - Promon
 - PreEmptive DashO

Environment Checks

Einführung

- Besonders relevant in Apps mit medizinischem oder finanziellem Bezug
- „Environment Checks“ sind Tests, die zur Laufzeit sicherstellen, dass Gerät in „sicherem“ Zustand ist
 - Anwendungen erfordern teilweise, dass Gerät **nicht gerootet** ist, keine **Custom Firmware** einsetzt etc.
- Je nach Typ wird auf verschiedenste Eigenschaften getestet
 - Werte in Dateien
 - Vorhandensein verschiedenster Dateien & Ordner
 - etc.

Motivation

- Mechanismen, die bereits zur Compile-Zeit angewandt werden, können teils sehr einfach zur Laufzeit ausgehebelt/nachvollzogen werden
 - Beispiel Code Encryption
 - Die .dex-Datei liegt ursprünglich in verschlüsseltem Format vor
 - Spätestens vor Ausführung muss Datei entschlüsselt & in Speicher geladen werden
 - Sobald die .dex-Datei im Speicher landet, kann sie leicht aus Speicher extrahiert werden
- Environment Checks sorgen beispielsweise dafür, dass gerootetes Gerät oder Versuch, sich mit frida einzuklinken, erfolgreich erkannt wird

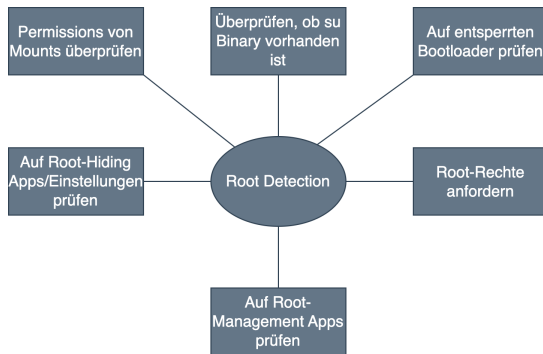
Detection Mechanismen

- Root Detection
- Hook Detection
- Emulator & Custom Firmware Detection
- Jailbreak Detection

Root Detection

- Dient dazu, dass gerootetes Gerät erkannt wird
- Zugriff auf **root** Benutzer:in ist Indiz für etwaige weitere Modifikationen

Root Detection



Tests der Root Detection

- Vorhandensein von verschiedenen Dateien
 - `/usr/bin/su`
 - `/sbin/su`
 - etc.
- Vorhandensein von verschiedenen Applikationen
 - Magisk (`com.topjohnwu.magisk`)
 - SuperSU (`eu.chainfire.supersu`)
 - etc.
- Möglichkeit, Befehle als `root` Benutzer:in auszuführen
 - Magisk bietet eine eigene Bibliothek an

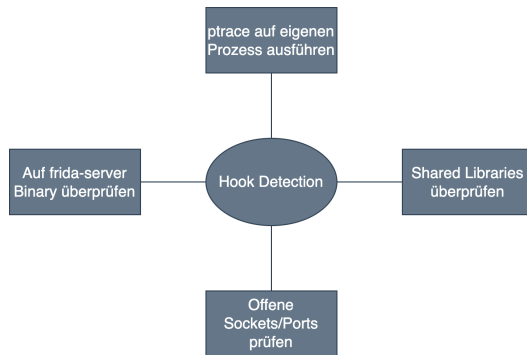
Detection Mechanismen

- Root Detection
- Hook Detection
- Emulator & Custom Firmware Detection
- Jailbreak Detection

Hook Detection

- Dient dazu, dass erkannt wird, ob Applikation gehookt wird
- Durch Hooken können andere Mechanismen & Hooking selbst verschleiert werden

Hook Detection



Tests der Hook Detection 1/2

- Vorhandensein von Dateien in `/data/local/tmp`
 - `frida`
 - `frida-server`
 - `etc.`
- Verbindung zu Port 27042, welcher standardmäßig von **frida** aufgemacht wird
- **frida** verwendet Named Pipes zur Kommunikation
 - Auf Android (immer) unter dem Pfad `/data/local/tmp/pipe-[a-f0-9]{32}` zu finden

Tests der Hook Detection 2/2

- Überprüfung von Werten im virtuellen Verzeichnis `/proc`
 - `/proc/self/maps` beinhaltet die verwendeten geteilten Bibliotheken ⇒ **frida** taucht darin auf
 - `/proc/self/maps` kann benutzt werden, um Speicherregionen zu öffnen & z. B. nach Wort „frida“ zu durchsuchen
 - Wird `/proc/self/fd` mit `readlink` aufgelöst, beinhaltet bei Verwendung von **frida** retournierten Pfad **linjector**
 - `/proc/self/tasks` beinhaltet bei Verwendung von **frida** Werte **pool-frida** & **gmain**

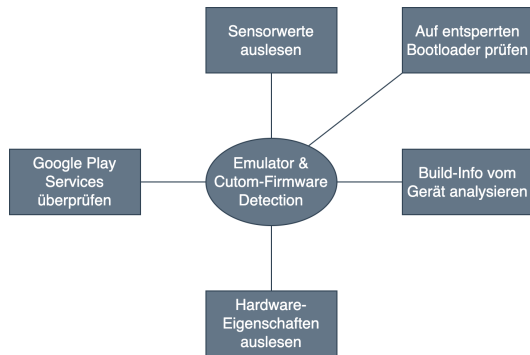
Detection Mechanismen

- Root Detection
- Hook Detection
- Emulator & Custom Firmware Detection
- Jailbreak Detection

Emulator & Custom Firmware Detection

- Dient dazu, dass erkannt wird, ob inoffizielles Betriebssystem auf Gerät läuft oder Gerät emuliert wird
- Custom Firmware ist oftmals Indiz für weitere Modifikationen

Emulator & Custom Firmware Detection



Tests der Emulator & Custom Firmware Detection 1/3

- Vorhandensein von gewissen Werten
 - In virtueller Datei (vom Kernel generiert) `/proc/version` sind oft Informationen zu Entwickler:innen des Kernels/der Custom ROM enthalten
- Auslesen der **System Properties**
 - Werden oftmals beim Bauen integriert & sind daher nicht einfach änderbar
 - Enthalten je nach ROM zusätzlich Einträge bzw. Werte, welche Rückschlüsse auf Custom Firmware zulassen

Tests der Emulator & Custom Firmware Detection 2/3

```
1 [ro.build.display.id]: [lineage_crosshatch-userdebug 11 RQ3A.211001.001 a384684437]
2 [ro.build.flavor]: [lineage_crosshatch-userdebug]
3 [ro.lineage.build.version]: [18.1]
4 [ro.lineage.build.version.plat.rev]: [0]
5 [ro.lineage.build.version.plat.sdk]: [9]
6 [ro.lineage.device]: [crosshatch]
7 [ro.lineage.display.version]: [18.1-20220421-NIGHTLY-crosshatch]
8 [ro.lineage.releasetype]: [NIGHTLY]
9 [ro.lineage.version]: [18.1-20220421-NIGHTLY-crosshatch]
10 [ro.lineagelegal.url]: [https://lineageos.org/legal]
```

Tests der Emulator & Custom Firmware Detection 3/3

- Vorhandensein von gewissen Pfaden
 - `/vendor/bin/qemu-props`
 - `/dev/qemu_pipe`
 - `/sys/qemu_trace`
 - `/vendor/bin/qemu-props`

Detection Mechanismen

- Root Detection
- Hook Detection
- Emulator & Custom Firmware Detection
- Jailbreak Detection

Jailbreak

- Ziel: iOS-Sicherheitsmechanismen durchbrechen (z. B. Sandbox)
- Nutzung einer Reihe von Schwachstellen im System
- Deaktivieren der „Chain of Trust“
- Kernel-Modifikationen: Installation von unsignierten/benutzer:innendefinierten Anwendungen
- **Jailbreak Detection:**
 - Exklusiv auf iOS Geräten zu finden
 - Erkennung von Jailbreak ist meistens Indiz dafür, dass noch mehr Modifikationen durchgeführt wurden

(Semi-)Tethered Jailbreaks

▪ Tethered Jailbreak

- Computer erforderlich
- Verändert Bootvorgang (iBoot-Stufe)
- Kann nur in einem einzigen Bootvorgang verwendet werden ⇒ temporär
- Nach Aus- & Einschalten ⇒ Recovery Mode

▪ Semi-Tethered Jailbreak

- Wie Tethered, jedoch landet man nicht im Recovery Mode

(Semi-)Untethered Jailbreaks

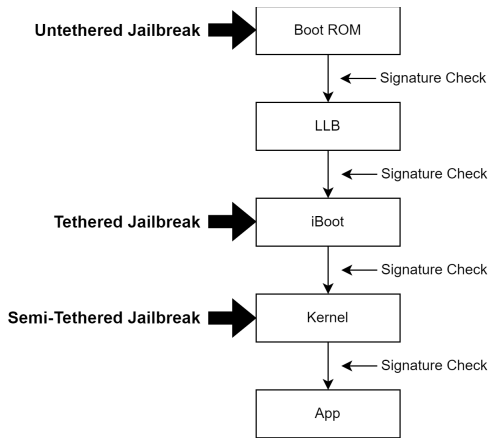
▪ Untethered Jailbreak

- Kein Computer erforderlich
- Während des Bootvorgangs wird Kernel automatisch exploited \Rightarrow permanent
- Schwierig zu implementieren

▪ Semi-Untethered Jailbreak

- Kein Computer erforderlich
- Verwendet eine App, Website oder andere Software

Jailbreak Arten in Chain of Trust



LLB = Low Level Bootloader

(Vergleiche Liu et al. (2016), Kellner et al. (2019), Ali et al. (2019))

Cydia

- Paketmanager für alternative Apps & Erweiterungen, welche nicht im offiziellen App Store erhältlich
- Jailbreak-Tools installieren Cydia oft zusätzlich
- Advanced Package Tool (APT) für iOS-Geräte
 - Grafische Benutzer:innenoberfläche
 - Unterstützt Installation von Apps & einigen Tools zur Manipulation von Apps
- Registriert ein URL Scheme (`cydia://`)

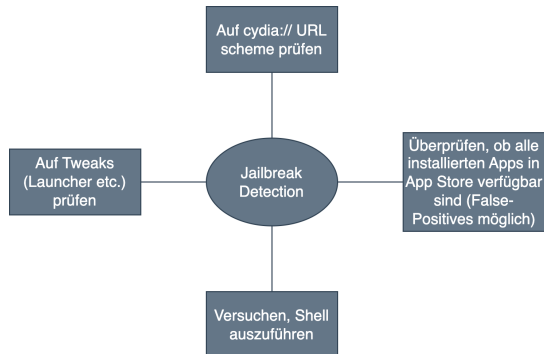


(Vergleiche <https://cydia.saurik.com>)

Beispiele für Jailbreak Tools & Exploits

- Tools
 - Checkra1n (14.0 - 14.8.1): Semi-Tethered
 - Taurine (14.0 - 14.3): Semi-Untethered
 - Unc0ver (14.0 - 14.8): Semi-Untethered
- Exploits
 - Checkm8 (CVE-2019-8900)
 - Ivac entry use-after-free (CVE-2021-1782)
 - Pattern-f's closed source exploit (CVE-2021-30883)
 - Cicutu_virosa (CVE-2021-1782)

Jailbreak Detection



Beispiele für Jailbreak Detection in Apps

- Dateien/Apps prüfen (z. B. Cydia in `/Applications`, Paths mit `isReadableFile` testen)
- Dateisystem-Prüfungen (z. B. versuchen, in `/privat`-Verzeichnis zu schreiben)
- Überprüfung der Plattformfunktionalität (z. B. URL schemes - `canOpenURL`, `bin/sh` Check mit `system`-Funktion)

Zusammenfassung 1/3

- Obfuscation: Veränderung von Code ohne Beeinflussung des Verhaltens
- Encryption: Verschlüsseln des Codes, Entschlüsselung während Laufzeit
- Erschwert Reverse Engineering
- Vielzahl an Varianten, beispielsweise
 - Identifier Renaming
 - Control Flow Obfuscation
 - Data Obfuscation

Zusammenfassung 2/3

- Code-Analyse: statisch vs. dynamisch
- statisch = Source-Code ansehen
- dynamisch = zur Laufzeit, kein Source-Code erforderlich
- Tools:
 - Debugger
 - frida
 - ...

Zusammenfassung 3/3

- Environment Checks um sicherzustellen, dass sich Gerät in “sicherem” Zustand befindet
- Beispiele für Environment Checks
 - Root Detection
 - Hook Detection
 - Jailbreak Detection
- Schutz von persönlichen Daten oder vor Angriffen auf Compile-Zeit Schutzmechanismen
- Umsetzung durch Prüfen verschiedenster Geräteeigenschaften

Literaturverzeichnis 1/7

- Android Runtime (ART) and Dalvik:
<https://source.android.com/devices/tech/dalvik>
- Dalvik Executable format:
<https://source.android.com/devices/tech/dalvik/dex-format>
- frida: The Engineering Behind the Reverse Engineering: <https://frida.re/slides/osdc-2015-the-engineering-behind-the-reverse-engineering.pdf>
- GitHub: frida: <https://github.com/frida>

Literaturverzeichnis 2/7

- Elenkov (2014): Android Security Internals
- Drake, et al. (2014): Android Hacker's Handbook
- Chell (2015): The Mobile Application Hacker's Handbook
- Vijay Kumar Velu. Mobile Application Penetration Testing. Birmingham: Packt Publishing Ltd, 2016. isbn: 978-1-78588-337-8.

Literaturverzeichnis 3/7

- Parvez Faruki, Hossein Fereidooni, Vijay Laxmi, Mauro Conti, und Manoj Gaur. [Android Code Protection via Obfuscation Techniques: Past, Present and Future Directions](#). 2016. doi: [10.48550/arXiv.1611.10231](#)
- Pierre Graux, Jean-Francois Lalande, und Valérie Viet Triem Tong. [Obfuscated Android Application Development](#). In *Proceedings of the Third Central European Cybersecurity Conference, CECC 2019*. Association for Computing Machinery, 2019. doi: [10.1145/3360664.3361144](#)

Literaturverzeichnis 4/7

- Shuaike Dong, Menghao Li, Wenrui Diao, Xiangyu Liu, Jian Liu, Zhou Li, Fenghao Xu, Kai Chen, Xiaofeng Wang, und Kehuan Zhang. [Understanding Android Obfuscation Techniques: A Large-Scale Investigation in the Wild](#). In *International Conference on Security and Privacy in Communication Systems*, Seiten 172–192. Springer, 2018.
[doi: 10.1007/978-3-030-01701-9_10](#)
- Timea László und Ákos Kiss. [Obfuscating C++ Programs via Control Flow Flattening](#). *Annales Universitatis Scientiarum Budapestinensis de Rolando Eötvös Nominatae, Sectio Computatorica*, 30(1):3–19, 2009

Literaturverzeichnis 5/7

- Kaiyuan Kuang, Zhanyong Tang, Xiaoqing Gong, Dingyi Fang, Xiaojiang Chen, und Zheng Wang. [Enhance virtual-machine-based code obfuscation security through dynamic bytecode scheduling.](#)
Computers & Security, 74:202–220, 2018
- PreEmptive DashO – <https://www.preemptive.com/products/dasho/>

Literaturverzeichnis 6/7

- Feng Liu, Ke-Sheng Liu, Chao Chang, und Yan Wang. [Research on the Technology of iOS Jailbreak](#).
In *2016 Sixth International Conference on Instrumentation Measurement, Computer, Communication and Control (IMCCC)*, Seiten 644–647, 2016.
doi: [10.1109/IMCCC.2016.178](https://doi.org/10.1109/IMCCC.2016.178)
- Ansgar Kellner, Micha Horlboge, Konrad Rieck, und Christian Wressnegger. [False Sense of Security: A Study on the Effectivity of Jailbreak Detection in Banking Apps](#).
In *2019 IEEE European Symposium on Security and Privacy (EuroS P)*, Seiten 1–14, 2019.
doi: [10.1109/EuroSP.2019.00011](https://doi.org/10.1109/EuroSP.2019.00011)

Literaturverzeichnis 7/7

- Amin Aenurahman Ali, Niken Cahyani, und Erwin Jadied. [Digital Forensic Analysis on iDevice: Jailbreak iOS 12.1.1 as a Case Study](#). *Indonesia Journal on Computing (Indo-JC)*, 4(2):205–218, Sep. 2019. doi: [10.34818/INDOJC.2019.4.2.349](https://doi.org/10.34818/INDOJC.2019.4.2.349)
- Bernd Prünster, Gerald Palfinger, und Christian Kollmann. [Fides: Unleashing the Full Potential of Remote Attestation](#). In *ICETE (2)*, Seiten 314–321, 2019

Vielen Dank!

`https://establishing-security.at/`