esse

ESSE Mobile Security – VO 3:
Statische und dynamische Analyse von Applikationen

Raphael Kiefmann, Paul Kalauner, Christian Schanes

INSO – Industrial Software
Institut für Information Systems Engineering | Fakultät für Informatik | Technische Universität Wien

Agenda

Grundlagen zur Code-Analyse

Statische Analyse

- Dekompilierung
- Deassemblierung
- Spezialfall Datenforensik

Dynamische Analyse

- Diagnostische Anwendungen
- Debugging von Anwendungen
- Dynamische Instrumentation

frida Demo

Grundlagen zur 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)
- siehe auch *Introduction to Security (183.594)*

Statische Analyse

Die 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

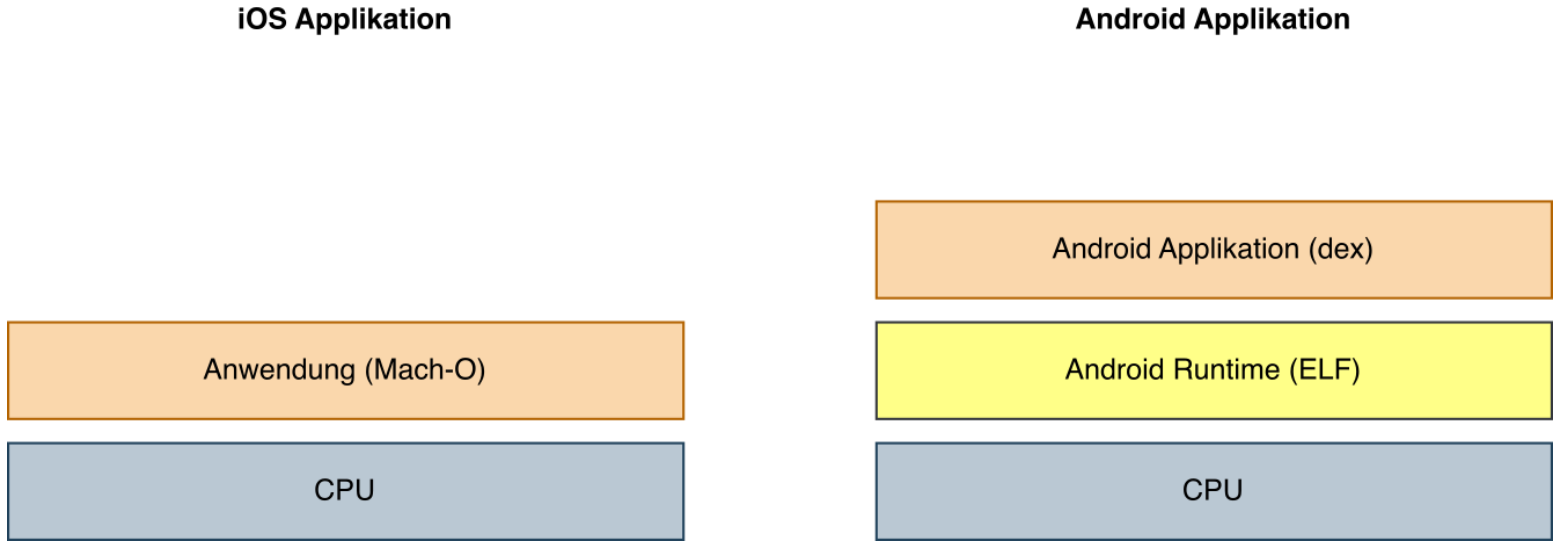
- mobile Applikationen sind ZIP-Bündel
  - Android-Applikationen werden in einer *.apk Datei gebündelt
  - iOS-Applikationen werden in einer *.ipa Datei gebündelt
- die Bündel enthalten Ressourcen (Mediendateien, Bibliotheken, etc.) & die eigentliche Anwendung
  - Android-Anwendungen sind im *.(o)dex 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 & dekompiliert)
- radare (deassembliert & kann mit Erweiterungen auch dekompilieren)
  - iaito ist eine GUI für radare
- IDA Pro (deassembliert & dekompiliert)
- 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** Unterstüzung) 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

$\Rightarrow$ *Selected Topics of Digital Forensics I (194.065)*

Dynamische Analyse

Dynamische Analyse von Applikationen

- 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
    - iostrace
    - strace
    - jtrace

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

Debugging als dynamische Analyse – Watchpoints

- Watchpoints sind nicht an einem fixen Ablauf des Programms, sondern konditional
  - Stellen im Speicher können überwacht werden und es kann darauf reagiert werden, falls sich der Wert ändert
- wie auch bei Breakpoints existieren zwei verschiedene Typen von Watchpoints
  - software-basiert: der Breakpoint muss in den Code injiziert werden, verlangsamt dadurch den Ablauf
  - hardware-basiert: die CPU ist dafür verantwortlich auf Änderungen zu reagieren, dadurch ist der Performance-Verlust nicht zu groß

Einschränkungen des Debuggings

- das Debugging ist meist nur während der Entwicklung relevant
  - beim Entwicklen 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 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

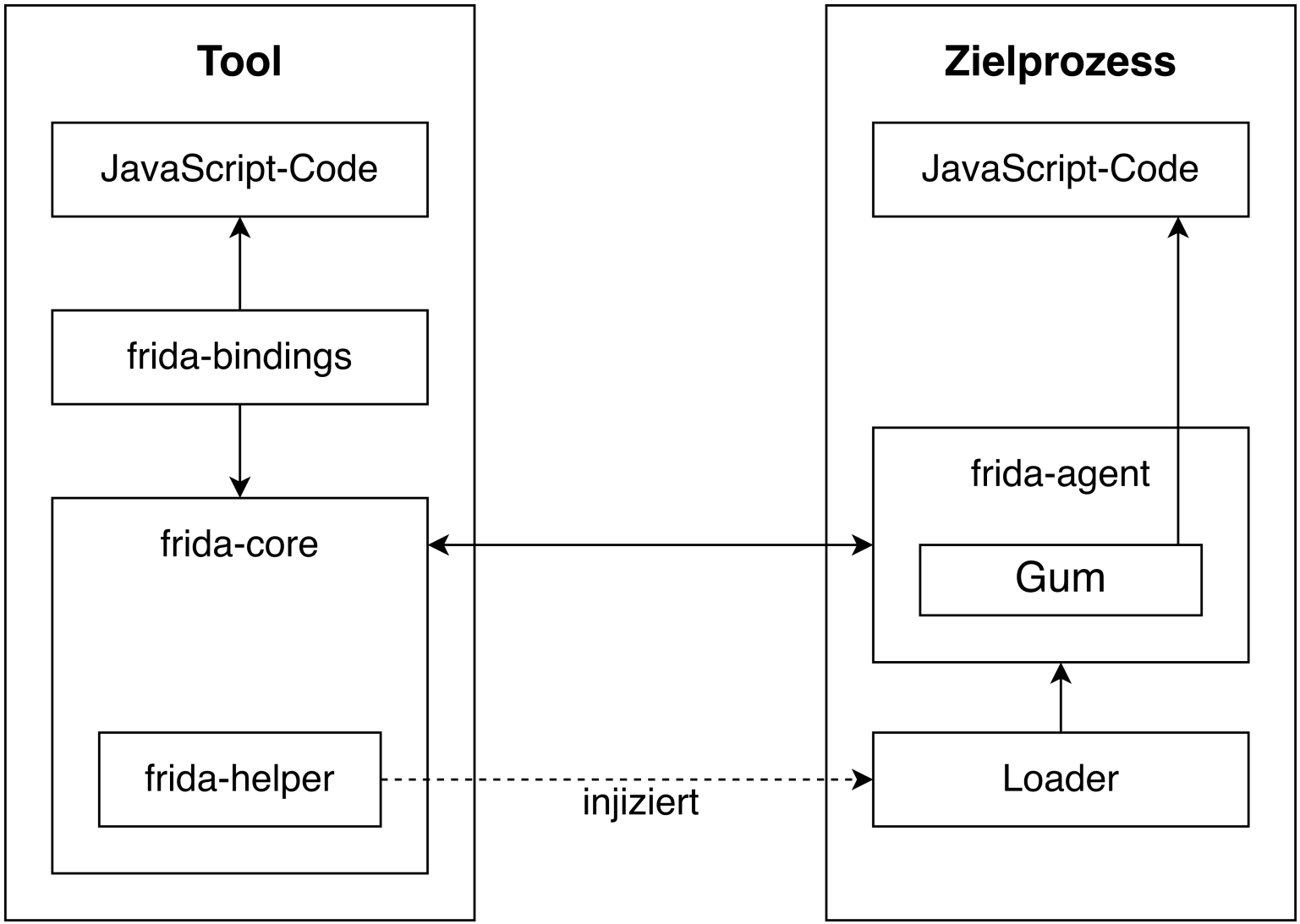
Reverse Engineering mit frida 2/2

- 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

Komponenten von frida

- **frida-core**: Kernlogik, inkludiert Injection-Code
- **Gum**: Instrumentation-Library, welche eine JavaScript-Engine inkludiert und von *frida-core* über JavaScript-Bindings (GumJS) verwendet wird
- **frida-agent**: Shared-Library, läuft im Zielprozess, nimmt Befehle von *frida-core* entgegen und leitet diese an *Gum* weiter
- **frida-helper**: Injiziert *frida-agent* in den Zielprozess
- **frida-bindings**: Stellt Funktionen anderen Programmiersprachen zur Verfügung
- **frida-server**: Macht *frida-core* für Remote-Hosts über TCP verfügbar

Architektur von frida



(Vergleiche `https://frida.re/docs/hacking`)

Injection-Prozess von frida

- *frida-helper* nutzt `ptrace`, um mit Zielprozess zu interagieren und speichert aktuellen Programmstatus
- *frida-helper* allokiert mittels `mmap` einen Speicherbereich für den Loader-Code
- Loader wird an allokierten Speicherbereich kopiert
- Loader wird ausgeführt und *frida-agent* wird geladen
- Wiederaufnahme der normalen Programmausführung

frida Demo

frida – ein Ausblick

- über die Jahre wurden verschiedene Tools basierend auf frida entwickelt
  - von Projekten mit eingeschränkten Funktionsumfang bis hin zu großen Frameworks
- das bekannteste Beispiel ist **objection**
  - stellt vorgefertigte frida-Scripts für verschiedenste Zwecke zur Verfügung
- weitere Beispiele sind:
  - **r2frida:** ermöglicht das Einhängen des Deassemblierungs- und Dekompilierungsframeworks zur Laufzeit
  - **fridump:** ermöglicht Dumpen des Arbeitsspeichers eines Prozesses

Netzwerkverkehranalyse bei Applikationen

- heutzutage haben viele Apps eine Internet-Anbindung
- oftmals werden wichtige Informationen nachgeladen oder an einen Server geschickt
- es existieren mehrere Varianten, um den Netzwerkverkehr mitzuschneiden

$\Rightarrow$ Einheit **Absicherung von Netzwerkverkehr** am **24.11.2022**

Zusammenfassung

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

Literaturverzeichnis 1/2

- Android Runtime (ART) and Dalvik: https://source.android.com/devices/tech/dalvik, abgerufen: 07. 11. 2022
- Dalvik Executable format: https://source.android.com/devices/tech/dalvik/dex-format, abgerufen: 07. 11. 2022
- frida: The Engineering Behind the Reverse Engineering: https://frida.re/slides/osdc-2015-the-engineering-behind-the-reverse-engineering.pdf, abgerufen: 07. 11. 2022
- GitHub: frida: https://github.com/frida, abgerufen: 07. 11. 2022

Literaturverzeichnis 2/2

- 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.

Vielen Dank!

https://establishing-security.at/