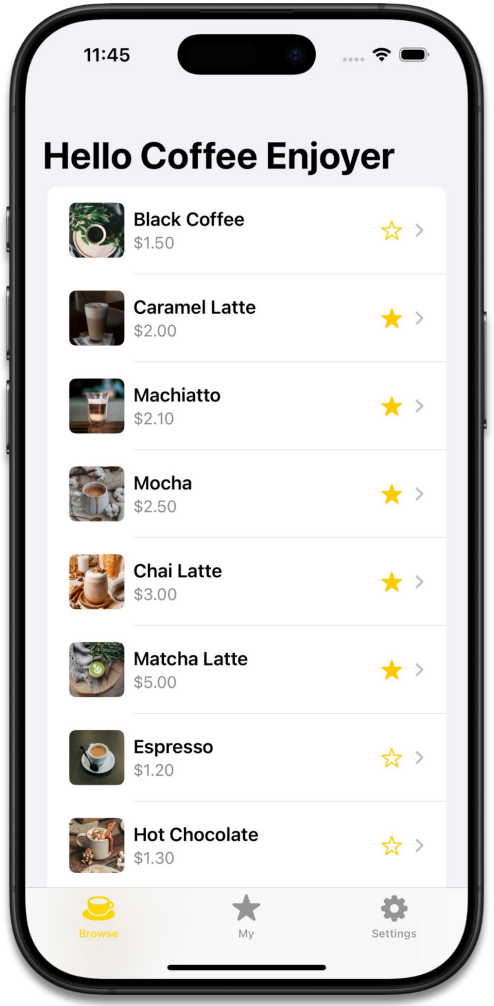
10 Wege,
wie ich deine App zerlege

10 Ways to Tear Your App Apart

23.04.2025 - FH Salzburg - Emanuel Mairoll

About this talk

- Very practical talk
- Many live demos
- Tools for both iOS & Android
  (see icons in bottom right)
- “Coffee Demo” app created
  specifically for this talk
  - Security wise, basically everything is wrong
  - Will show how to exploit it



About Me

- Emanuel Mairoll
- BSc Computer Science, Univ. of Salzburg
- MSc Cyber Security, ETH Zürich (current)
- Former Mobile Developer (iOS & Android), Salzburg, 4 years
- Researcher, Network Security Group @ ETH Zürich
- Forward and Reverse Engineer, passionate CTF player

DISCLAIMER:

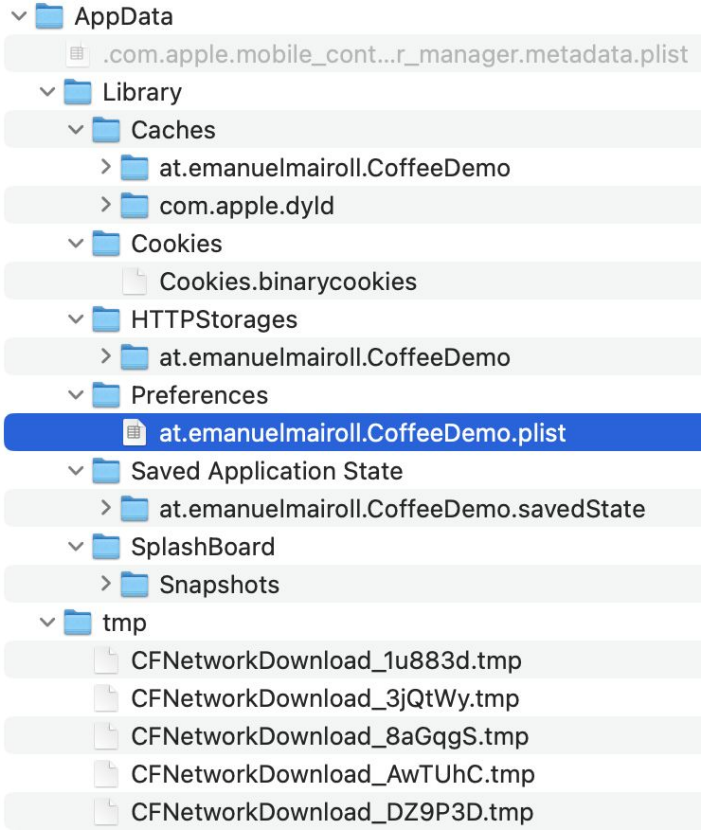
FOR EDUCATIONAL PURPOSES ONLY

Use this knowledge responsibly.

0: Local Storage - Teaser

Local Storage

- Local App Storage is trivial to access
  - Preference File
  - Documents / scratch folder
  - SQLite stuff
- Android: FX File Explorer, adb etc.
- iOS: XCode, ios-deploy etc.
  - Needs to be Sideloaded
    (will see later what that means)
  - Alternative: via “Restore from Backup”



App Deployment Overview

App Deployment: iOS

- Apps packaged as .ipa files (ZIP containers)
- Contains Mach-O binary (compiled Swift/Obj-C code)
- Code signing required to run on iOS
- Protected by FairPlay DRM
- Strict sandbox environment, interactions via restrictive APIs
- No direct root or shell access; neither on device or development machine

App Deployment: Android

- Apps packaged as .apk files (ZIP containers)
- Contains Java/Kotlin bytecode (.classes.dex)
- Typically obfuscated (using R8) to hinder reverse engineering
- Signature required but not strictly enforced by the Play Store
- Runtime environment more open; apps can request permissions for cross-app data access

From Tiny C to Mobile Binaries

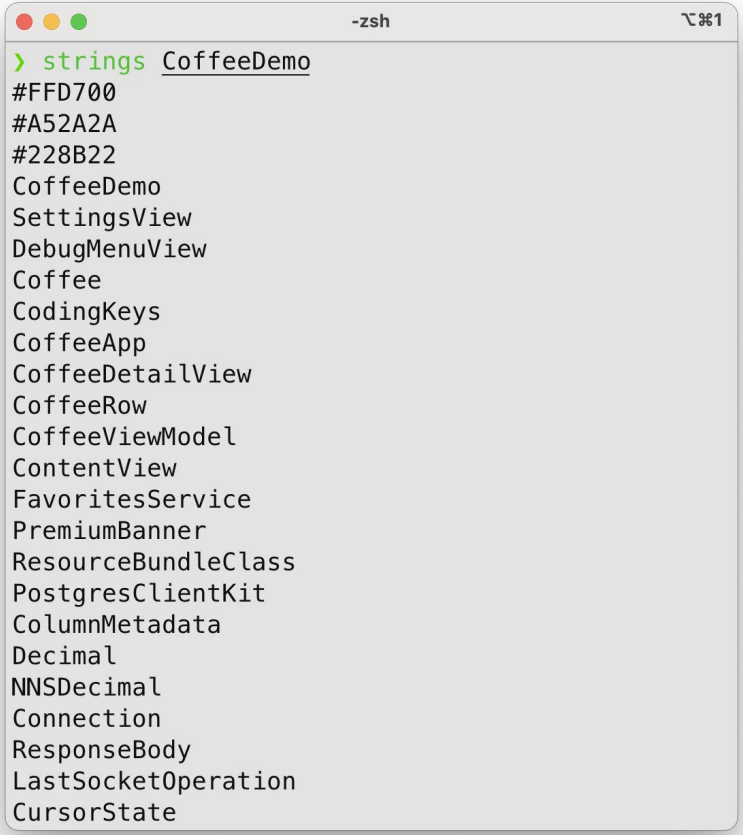
- Mobile apps bundle everything they need to run
- Results in very large, complicated binaries
- To understand the basics, we first look at this simple program
- Reads password from stdin, compares it to hard coded string

```c
#include <stdio.h>
#include <string.h>

int main(void) {
    char pw[64];

    fputs("Password: ", stdout);
    fgets(pw, sizeof(pw), stdin);

    if (strncmp(pw, "supersecretpassword", 19) == 0) {
        puts("Access granted");
        return 0;
    } else {
        puts("Access denied");
        return 1;
    }
}
```

1: strings

strings: What is it?

- `strings` finds all plaintext in binaries
- Prints it to the console
- Can be used to easily extract hard coded credentials
- Works on many different file formats
  - Binaries (ELF, Mach-O, etc.)
  - Container Formats (.zip, .apk, .ipa)
  - Many more



strings: Best practices

- Be aware that plain strings in binaries are easily extractable
- Never store keys or secrets in cleartext
- Use complex validation logic for sensitive inputs
- Obfuscate or encrypt critical strings

2: App Depackaging

App Depackaging

- Apps distributed as .ipa / .apk bundle formats
  - essentially ZIP files
  - Contains code, frameworks and resources
- .ipa can be unzipped with standard zip tool
  - Code is in form of Mach-O binary
- .apk is best decompressed with jadx
  - Recovers “somewhat readable” Java Code

3: Disassembling

Disassembling and Decompilation Overview

- Process of translating binary code back to readable source code
- Symbols (eg. Class and Function names) usually stripped
- Manually recover application logic, algorithms, and potential vulnerabilities
  - APIs still show symbols
  - Logging patterns reveal intent
- Obfuscation techniques complicate this process, but cannot fully prevent it

Popular Binary Decompilers

- **Ghidra**: NSA open-source, good for large-scale analysis
- **IDA Pro**: Industry standard, extensive processor support
- **Binary Ninja**: User-friendly, scripting-friendly
- **Hopper**: macOS-focused, lightweight, good Obj-C support
- **Radare2**: Command-line driven, powerful but steep learning curve





JADX (Dex to Java decompiler)

- All-in-one Decompiler specifically for Android apps
- Converts class files to smali or back to Java source code
- Extracts resources such as XML layouts, images, and strings
- Java output is “readable” but obfuscation still present
- Alternatives: apktool, JD-GUI, fernflower

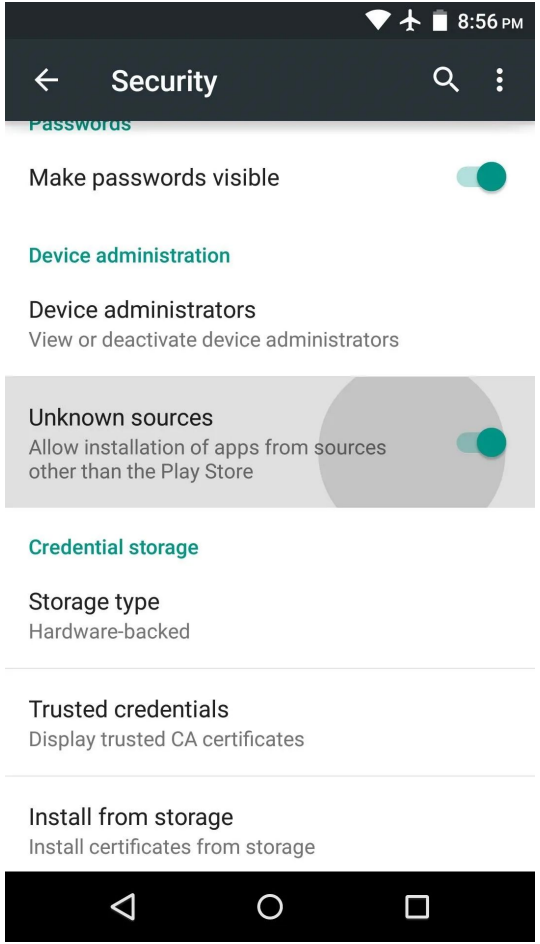
4: Sideloading

Sideloading

- Bypass App Store / Play Store
- Instead, install from local .ipa / .apk
- Deploy unsigned or modified app builds
- Very useful for developing, testing and tinkering
- However, also commonly used for piracy!

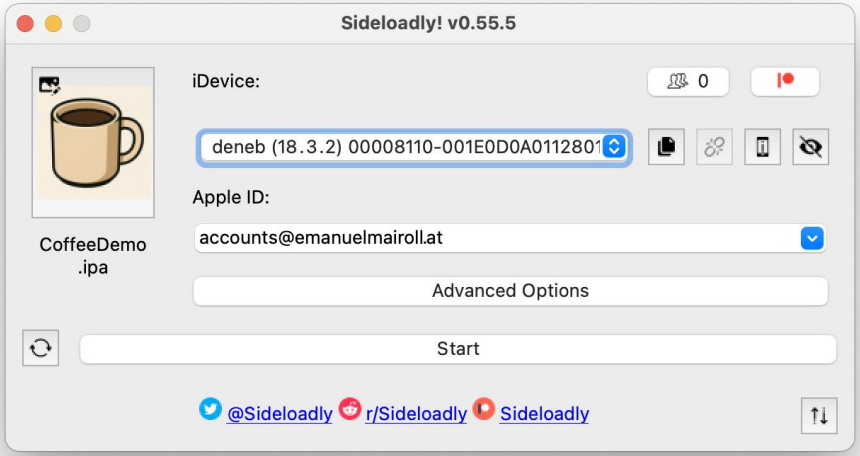
Sideloading on Android

- Enable “Unknown sources” in Settings
- Then install .apk directly on device
- Or via **`adb install app.apk`**
- Noteworthy Mention: **APK Explorer & Editor**
  - Browse and export APK resources (icons, XML, fonts).
  - Install, share, and manage APKs or bundles.
  - Edit, rebuild, and resign modified APK files.



Sideloading on iOS

- Only signed .ipa files can run
- Need to resign with own Apple ID
- Tool of Choice: Sideloadly
  - Sideload apps with free or paid Apple ID
  - Wi-Fi sideloading with automatic app refreshing
  - Customize installs: icons, bundle ID, **injections**
- Alternatives: AltStore, Cydia Impactor, XCode codesign



Sourcing App Packages

- **Android .apk:** relatively easy
  - Google “My App apk download”
  - Extract from device: **`adb shell pm path com.example.app`**
- **iOS .ipa:** complicated due to FairPlay DRM
  - Find on “decrypted app stores” (preferred)
  - Use Clutch or frida-ios-dump to strip FairPlay

Apple Fairplay DRM

- AppStore IPAs are encrypted with per-device key at download
- Protects code, strings & Objective-C metadata
- To analyze/install on other devices, DRM needs to be removed
- DRM stripping tools: Clutch, frida-ios-dump
- Requires a jailbroken device
- Stripping yields a decrypted IPA ready for sideloading

Mitigations and Protecting Against Piracy

- **Android:**
  - Verify APK signature programmatically
  - Use Play Integrity API or SafetyNet for tamper detection
- **iOS:**
  - Verify code signature & provisioning profile at runtime
  - Check team-ID and certificate chain

5: Patching

Patching

- *Readable* code means *Editable* code
- Can patch logic directly
  - Flipping conditional jump
  - Hard code function return value
- Prime targets are “**isPremium**”-like functions
- Can be done in disassembler or even simple hex editor

```
CSEL            X21, X9, X8, NE
MOV             X1, X21
BL              _$sSS6appendyySSF ; String.append(_:)
MOV             X0, X21
BL              _swift_bridgeObjectRelease
LDP             X0, X20, [X29,#var_40]
ADRP            X2, #__swiftEmptyArrayStorage_ptr@PAGE
LDR             X2, [X2,#__swiftEmptyArrayStorage_ptr@PAG
MOV             X1, X20
BL              _$s10Foundation5NSLogyySS_s7CVarArg_pdtF
MOV             X0, X20
BL              _swift_bridgeObjectRelease
MOV             X0, X19
SUB             SP, X29, #0x30 ; '0'
LDP             X29, X30, [SP,#0x30+var_s0]
LDP             X20, X19, [SP,#0x30+var_10]
LDP             X22, X21, [SP,#0x30+var_20]
LDP             X24, X23, [SP+0x30+var_30],#0x40
RET
; End of function sub_10000F368
```

Interim Summary

- No sensitive strings in binary
- Avoid simple premium checks
- Obfuscate licensing logic
- Implement signature checks
- Fundamental principle:

Interim Summary

- ~~No sensitive strings in binary~~
- ~~Avoid simple premium checks~~
- ~~Obfuscate licensing logic~~
- ~~Implement signature checks~~
- Fundamental principle:
  NEVER TRUST THE CLIENT

Breakpoint - 5 Min Pause

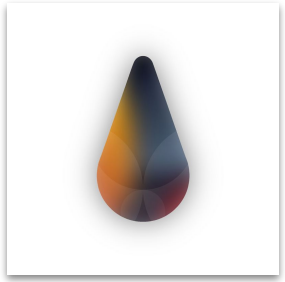
6: Jailbreaking / Rooting

iOS Jailbreaking: A bit of history

- Jailbreaking predates the Apple App Store
- Just days after the first iPhone launch, Jay Freeman's "saurik" team released the original "jailbreak"
- Cydia emerged as the primary distribution platform for tweaks
- Over the years, numerous jailbreaks appeared for successive iOS versions
- Apple's Secure Bootchain & SEP enhancements have made modern jailbreaks increasingly challenging

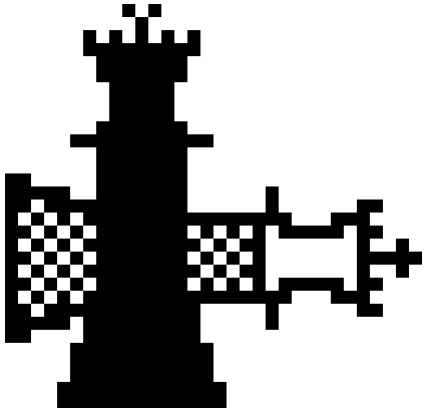
What Is Jailbreaking?

- Privilege escalation to remove iOS restrictions
  - Root access / full filesystem
  - Run unsigned code
  - Hooking frameworks (Cydia Substrate, libSubstitute)
  - **Tweaks** (UI & behavior modifications, **instrumentation tools**)
- Requires specific exploit per iOS version
- Current jailbreaks:
  - Dopamine: iOS 16.0 - 16.5
  - palera1n: iPhone 5S - iPhone X (checkm8)



checkm8 BootROM Exploit

- Hardware-level vulnerability in the BootROM of A5–A11 devices
  - iPhone 5S - iPhone X
  - select iPads
- Unpatchable by software or firmware updates
- Grants persistent low-level access at every boot
- Makes these devices excellent “lab devices”
  - Can forever be used to perform the upcoming techniques



Android Rooting

- Similar to jailbreaking, but far easier
- Most device bootloaders unlockable/unlocked by default
- Can adapt security constraints of device as desired
  - Full filesystem & kernel access
- Even able to run fully custom OS (GrapheneOS, LineageOS)
- Sidenote: Android OS is fully open source, anyone can fork

Magisk: Systemless Root

- Preferred Rooting tool in 2025
- Patches boot image without altering /system
- **Magisk Manager**: GUI for managing root grants & installing modules
- **MagiskSU**: selectively grants superuser privileges to apps
- **Magisk Modules**: extend OS functionality
  - SafetyNet fix
  - Ad-blocking
  - UI tweaks
- **Magisk Hide**: conceal root from sensitive apps (banking, DRM)

Protecting Against jailbroken / rooted devices

- Detect jailbreak / root at runtime
  - However: various bypasses exist
- **Android:** almost impossible to fully block
- **iOS:** only support latest iOS (but that’s a dick move)

7: Debugger

Debuggers for Mobile Apps

- Allows deep introspection of processes on machine code level
- Halt process, single step, inspect registers and memory, set breakpoints
- needs "debug server" either in process (debuggable app) or globally on jailbroken/rooted device (needs PTRACE / task_for_pid)
- Different debuggers for respective platforms
  - **gdb** for native Android components
  - **jdb** for Java/Kotlin Android Apps
  - **lldb** for Apps on iOS/macOS
- IMO: usefulness a bit limited in mobile apps

8: Frida

Frida

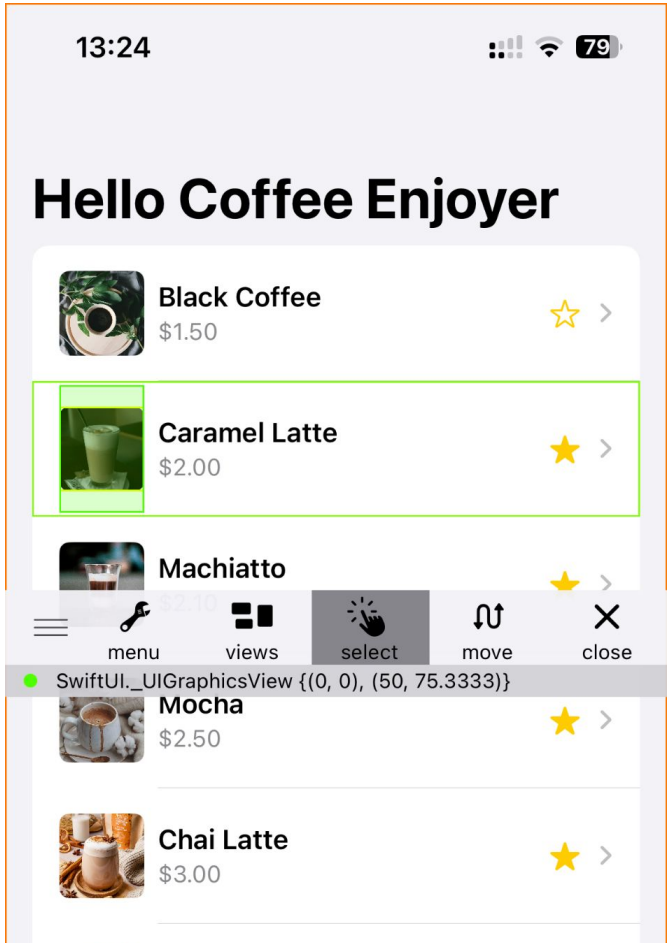
FRIDA

- Dynamic instrumentation framework:
  injects a JS engine into live processes
- Intercept, inspect & hook functions at runtime
- Replace implementations on-the-fly with custom JS scripts
- Enumerate loaded classes, methods & memory objects
- Deploy as `frida-server` on jailbroken/rooted devices or embed as a gadget

9: FLEX

💪 FLEX

- Runtime view-hierarchy inspector & editor
- Apply temporary UI patches
  (colors, labels, layout)
- Built-in explorers for Network,
  Local Storage, View Delegates, etc.
- Deploy as a Cydia tweak on jailbroken
  devices or as AutoFLEX gadget
- iOS only - Android UI (XML/Compose)
  usually decompiles “nice enough”
  to be readable



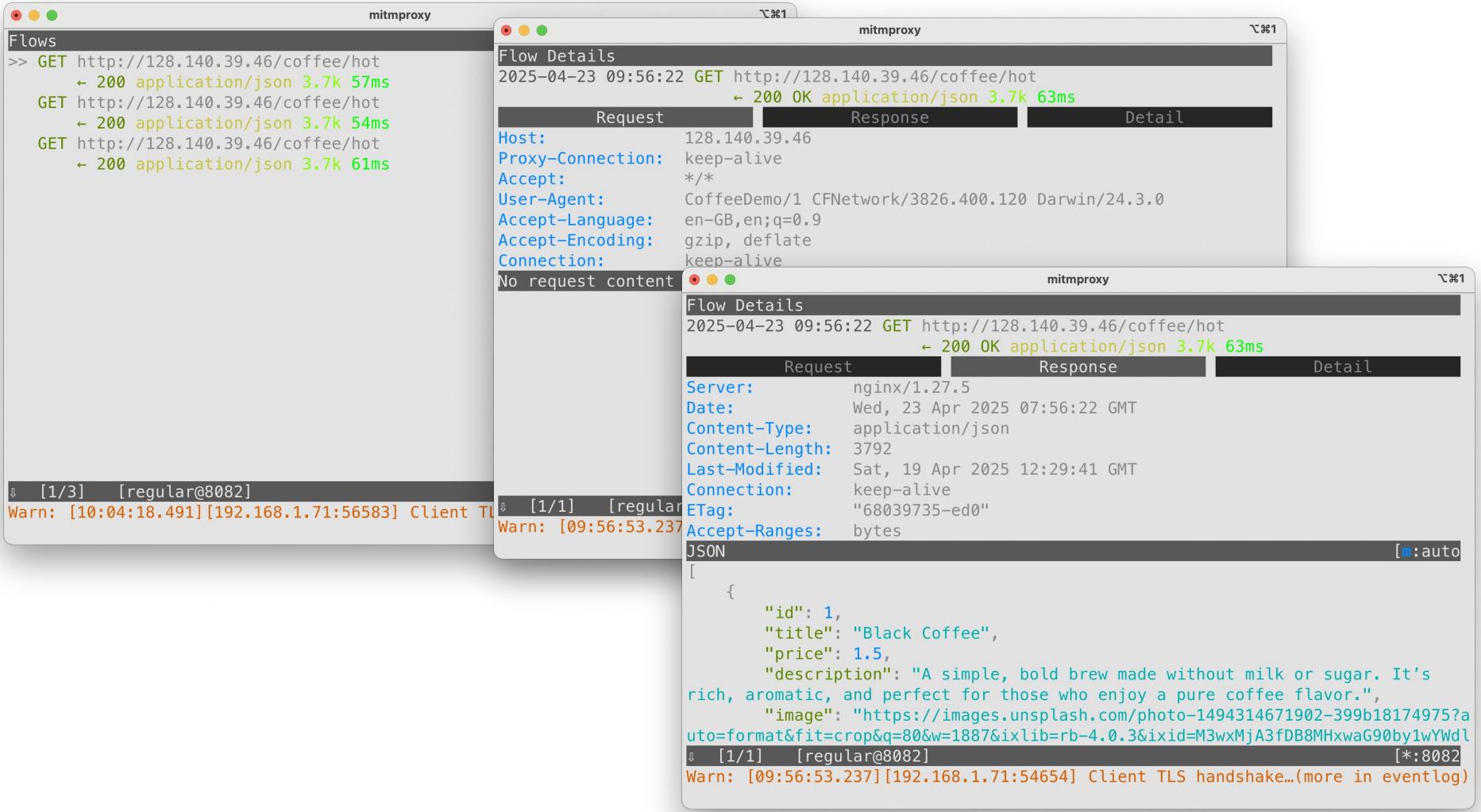
Some other noteworthy tweaks

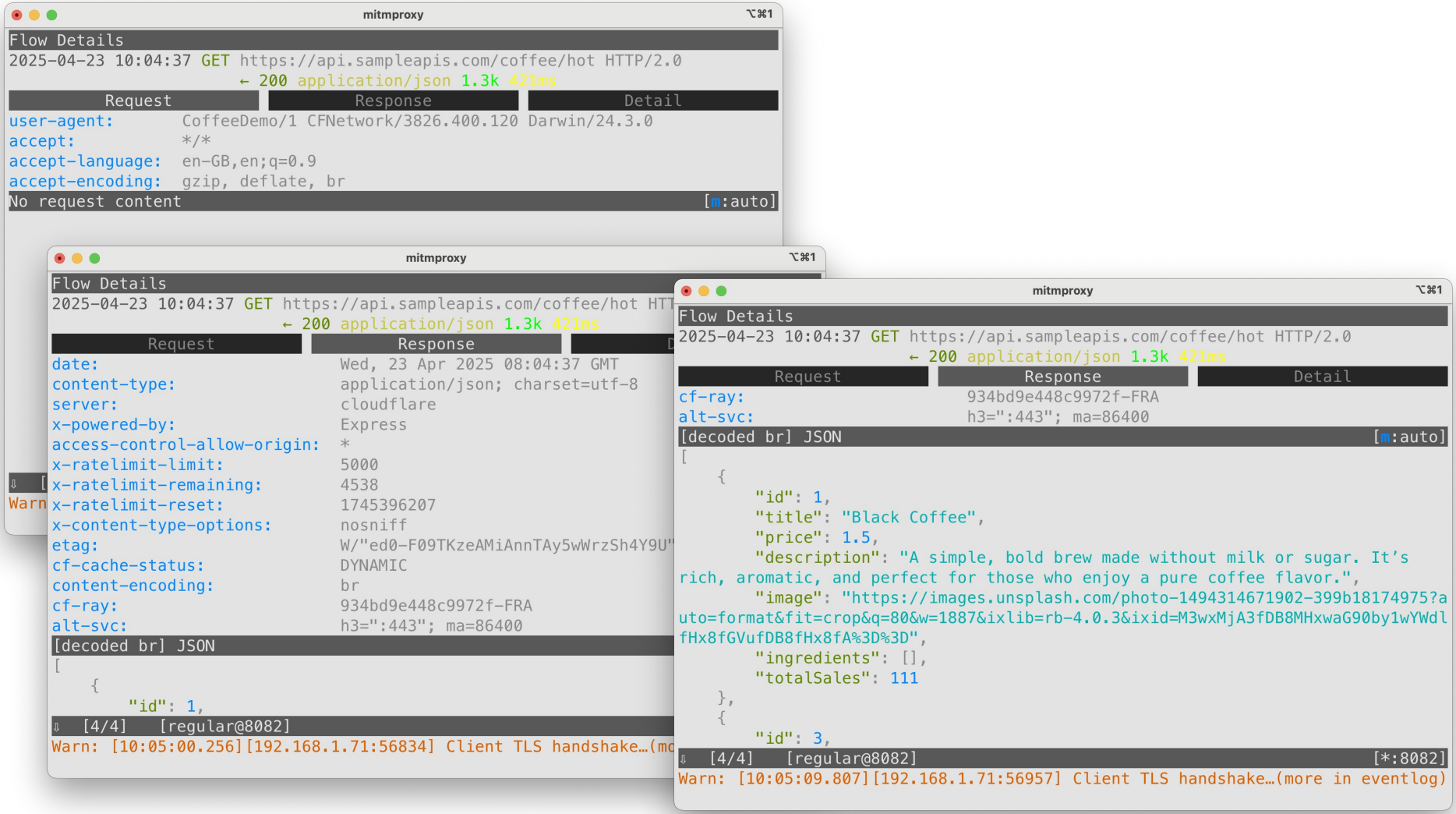
- Diverse Jailbreak/Root detection bypasses
- **Objection**: Frida-based toolkit (SSL-unpinning, runtime hooks)
- **Clutch** & **frida-ios-dump**: strip FairPlay DRM & decrypt IPAs
- **Cycript**: interactive ObjC runtime code exploration
- Legacy: In-App Purchase crackers (LocalIAPStore)

10: mitmproxy

mitmproxy

- Intercept and inspect HTTP/HTTPS traffic via a local proxy
- Configure your mobile device’s Wi-Fi proxy to point at mitmproxy
- Logs all requests & responses (headers, payloads)
- Install mitmproxy’s root CA certificate on the device for HTTPS decryption
- Useful for discovering API calls, tokens & parameters
- SSL-pinning in apps must be bypassed (e.g., with Frida)


mitmproxy
⌥⌘1
Flows
>> GET http://128.140.39.46/coffee/hot
← 200 application/json 3.7k 57ms
GET http://128.140.39.46/coffee/hot
← 200 application/json 3.7k 54ms
GET http://128.140.39.46/coffee/hot
← 200 application/json 3.7k 61ms
⇩ [1/3] [regular@8082]
Warn: [10:04:18.491][192.168.1.71:56583] Client TL
mitmproxy
⌥⌘1
Flow Details
2025-04-23 09:56:22 GET http://128.140.39.46/coffee/hot
← 200 OK application/json 3.7k 63ms
Request
Response
Detail
Host: 128.140.39.46
Proxy-Connection: keep-alive
Accept: */*
User-Agent: CoffeeDemo/1 CFNetwork/3826.400.120 Darwin/24.3.0
Accept-Language: en-GB,en;q=0.9
Accept-Encoding: gzip, deflate
Connection: keep-alive
No request content
⇩ [1/1] [regular
Warn: [09:56:53.237
mitmproxy
⌥⌘1
Flow Details
2025-04-23 09:56:22 GET http://128.140.39.46/coffee/hot
← 200 OK application/json 3.7k 63ms
Request
Response
Detail
Server: nginx/1.27.5
Date: Wed, 23 Apr 2025 07:56:22 GMT
Content-Type: application/json
Content-Length: 3792
Last-Modified: Sat, 19 Apr 2025 12:29:41 GMT
Connection: keep-alive
ETag: "68039735-ed0"
Accept-Ranges: bytes
JSON [m:auto
[
{
"id": 1,
"title": "Black Coffee",
"price": 1.5,
"description": "A simple, bold brew made without milk or sugar. It's rich, aromatic, and perfect for those who enjoy a pure coffee flavor.",
"image": "https://images.unsplash.com/photo-1494314671902-399b18174975?auto=format&fit=crop&q=80&w=1887&ixlib=rb-4.0.3&ixid=M3wxMjA3fDB8MHxwaG90by1wYWdl
⇩ [1/1] [regular@8082] [*:8082
Warn: [09:56:53.237][192.168.1.71:54654] Client TLS handshake…(more in eventlog)


mitmproxy
⌥⌘1
Flow Details
2025-04-23 10:04:37 GET https://api.sampleapis.com/coffee/hot HTTP/2.0
← 200 application/json 1.3k 421ms
Request
Response
Detail
user-agent: CoffeeDemo/1 CFNetwork/3826.400.120 Darwin/24.3.0
accept: */*
accept-language: en-GB,en;q=0.9
accept-encoding: gzip, deflate, br
No request content
[m:auto]
mitmproxy
⌥⌘1
Flow Details
2025-04-23 10:04:37 GET https://api.sampleapis.com/coffee/hot HTT
← 200 application/json 1.3k 421ms
Request
Response
date: Wed, 23 Apr 2025 08:04:37 GMT
content-type: application/json; charset=utf-8
server: cloudflare
x-powered-by: Express
access-control-allow-origin: *
x-ratelimit-limit: 5000
x-ratelimit-remaining: 4538
x-ratelimit-reset: 1745396207
x-content-type-options: nosniff
etag: W/"ed0-F09TKzeAMiAnnTAy5wWrzSh4Y9U"
cf-cache-status: DYNAMIC
content-encoding: br
cf-ray: 934bd9e448c9972f-FRA
alt-svc: h3=":443"; ma=86400
[decoded br] JSON
[
{
"id": 1,
[4/4] [regular@8082]
Warn: [10:05:00.256][192.168.1.71:56834] Client TLS handshake…(mo
mitmproxy
⌥⌘1
Flow Details
2025-04-23 10:04:37 GET https://api.sampleapis.com/coffee/hot HTTP/2.0
← 200 application/json 1.3k 421ms
Request
Response
Detail
cf-ray: 934bd9e448c9972f-FRA
alt-svc: h3=":443"; ma=86400
[decoded br] JSON
[m:auto]
[
{
"id": 1,
"title": "Black Coffee",
"price": 1.5,
"description": "A simple, bold brew made without milk or sugar. It's rich, aromatic, and perfect for those who enjoy a pure coffee flavor.",
"image": "https://images.unsplash.com/photo-1494314671902-399b18174975?auto=format&fit=crop&q=80&w=1887&ixlib=rb-4.0.3&ixid=M3wxMjA3fDB8MHxwaG90by1wYWdlfHx8fGVufDB8fHx8fA%3D%3D",
"ingredients": [],
"totalSales": 111
},
{
"id": 3,
[4/4] [regular@8082]
[*:8082]
Warn: [10:05:09.807][192.168.1.71:56957] Client TLS handshake…(more in eventlog)

Final Summary

Final Summary: Client-Side Security

- No unencrypted security-critical strings in binaries
- No plain “isPremium” functions; obfuscate licensing logic
- Enforce in-app code-signature checks
- But: DRM & obfuscation only delay attackers
- Sideloading, jailbreak/root, Frida & Flex fully expose internals
- **NEVER TRUST THE CLIENT**

Final Summary: Mitigations & Best Practices

- Push critical logic to server with strong authentication
- Treat API keys as public: use least privilege & rate limits
- Store keys in Secure Enclave / TEE
- Use attestation APIs (SafetyNet, DeviceCheck, etc.)
- Log & monitor server-side for anomalies

Q&A