Mobile Apps mit Claude Code nach OWASP prüfen

Table of Contents

Mobile Apps mit Claude Code nach OWASP prüfen

Von den Mobile Top 10 über MASVS und MASTG zum automatisierten Security-Audit für iOS und Android

Stand: Juli 2026

Dieses Tutorial soll dir zwei Dinge mitgeben:

Du verstehst, wie ein professioneller Mobile-Security-Test in der Praxis wirklich abläuft.

Am Ende hast du ein fertiges, wiederverwendbares Claude-Code-Audit-Paket, das du auf deine eigenen iOS- und Android-Apps ansetzen kannst.

Ich beginne bewusst mit iOS und SwiftUI, weil sich damit alle Konzepte an echtem Code zeigen lassen. Android kannst du später nach demselben Muster ergänzen (Kapitel 23).

Hinweis: „BKK Atomium“ ist eine frei erfundene Beispiel-Krankenkasse, keine reale Institution. Alle Codebeispiele stammen aus einer eigens dafür geschriebenen, absichtlich unsicheren Demo-App.

Wichtig: Die bereitgestellten Skills, Agents und Konfigurationen dienen ausschließlich als Beispiele für dieses Tutorial. Prüfen und aktualisieren Sie diese vor jedem Projekteinsatz hinsichtlich Zweck, Berechtigungen, Tool-Zugriffen, Architektur sowie Datenschutz und Sicherheit. Eine unveränderte Übernahme in produktive Systeme wird nicht empfohlen.

1. Was ist OWASP?

Die **OWASP Foundation** (Open Worldwide Application Security Project) ist eine gemeinnützige Organisation. Sie veröffentlicht offene, herstellerneutrale Standards, Leitfäden und Werkzeuge rund um Anwendungssicherheit. Genauso wichtig ist aber, was OWASP nicht ist:

Keine Behörde. OWASP kann keine App verbieten, zertifizieren oder zulassen.

Kein Zertifizierer. Ein offizielles „OWASP-Siegel“ für eine App gibt es nicht. Es gibt lediglich MASVS-*Level* (L1/L2), die ein unabhängiges Audit-Programm namens MAS Certification vergeben kann. Das ist nicht dasselbe wie „OWASP-zertifiziert“.

Kein Ersatz fürs eigene Denken. OWASP-Listen sind ein guter Einstieg, aber keine vollständige Checkliste für jede App.

In diesem Tutorial unterscheide ich bewusst vier Begriffe:

Begriff

Bedeutung

Beispiel

Risiko

Eine Kategorie möglicher Schwächen

„Unsichere Datenspeicherung“ (M9)

Anforderung

Eine konkrete, prüfbare Kontrolle

„Auth-Tokens werden in der Keychain gespeichert“ (MASVS-STORAGE-1)

Schwachstelle

Ein tatsächlich vorhandener Verstoß gegen eine Anforderung

„Token liegt in UserDefaults“

Test

Das Verfahren, mit dem man die Schwachstelle nachweist

Statische Codeprüfung + Auslesen des Simulator-Containers

Grenzen automatisierter Prüfungen: Ein Tool, egal ob Claude Code oder ein klassischer Scanner, erkennt Muster im Code. Es versteht die Geschäftslogik nicht von selbst, bewertet keine rechtlichen Konsequenzen und kann nie garantieren, vollständig zu sein. Bei einer Krankenkassen-App kommt noch etwas dazu: Versicherten- und Gesundheitsdaten sind nach Art. 9 DSGVO eine besondere Kategorie personenbezogener Daten. Ein „bestandenes“ automatisiertes Audit heißt deshalb nur: In diesem Umfang, mit diesen Tests, wurde keines der gesuchten Muster gefunden. Es heißt nicht „die App ist sicher“, und erst recht nicht „die App ist DSGVO-konform“.

So arbeitet ein Tester, und genau das bildet dieses Tutorial nach:

Auftrag und Scope festlegen
↓
Anwendung verstehen
↓
Bedrohungen identifizieren
↓
Statische Prüfung
↓
Dynamische Prüfung
↓
Befunde verifizieren

↓
Risiko bewerten
↓
Maßnahmen empfehlen
↓
Nachtest durchführen

Claude Code übernimmt in diesem Tutorial vor allem die statische Prüfung, die Dokumentation und die Koordination der Schritte. Verifikation, Risikobewertung und dynamische Tests auf echten Geräten bleiben, wo nötig, Aufgabe eines Menschen (mehr dazu in Kapitel 21).

2. Die drei verwendeten OWASP-Ressourcen

Man spricht oft von „drei OWASP“, gemeint sind aber drei verschiedene OWASP-Ressourcen, die aufeinander aufbauen:

Mobile Top 10 → Welche Risiken sind relevant?
MASVS → Welche Anforderungen muss die App erfüllen?
MASTG → Wie werden diese Anforderungen geprüft?

OWASP Mobile Top 10

Die Mobile Top 10 sind der leicht verständliche Einstieg in die wichtigsten Risikogruppen. Hier die aktuelle Liste von 2024: OWASP Mobile Top 10

M1 – Improper Credential Usage

M2 – Inadequate Supply Chain Security

M3 – Insecure Authentication/Authorization

M4 – Insufficient Input/Output Validation

M5 – Insecure Communication

M6 – Inadequate Privacy Controls

M7 – Insufficient Binary Protections

M8 – Security Misconfiguration

M9 – Insecure Data Storage

M10 – Insufficient Cryptography

OWASP MASVS

Der **Mobile Application Security Verification Standard** formuliert konkrete, prüfbare Kontrollen in acht Gruppen:

STORAGE – Datenspeicherung auf dem Gerät

CRYPTO – Kryptografische Verfahren

AUTH – Authentifizierung und Sessions

NETWORK – Netzwerkkommunikation

PLATFORM – Nutzung der Plattform-APIs

CODE – Codequalität und Build-Konfiguration

RESILIENCE – Widerstandsfähigkeit gegen Reverse Engineering und Manipulation

PRIVACY – Umgang mit personenbezogenen Daten

Jeder Punkt der Mobile Top 10 lässt sich einer oder mehreren MASVS-Gruppen zuordnen. Diese Zuordnung steht in references/masvs-mapping.md (Kapitel 6).

OWASP MASTG

Der **Mobile Application Security Testing Guide** beschreibt, *wie* man MASVS-Anforderungen praktisch prüft:

Testvoraussetzungen (Jailbreak/Root, Tools, Testgeräte)

statische Prüfungen (Quellcode, Binary, Konfiguration)

dynamische Prüfungen (Laufzeitverhalten, Netzwerkverkehr, Instrumentierung)

Werkzeuge (z. B. otool, class-dump, mitmproxy, Frida, Objection)

erwartete Beobachtungen und Bewertung der Ergebnisse

Die offizielle **MAS Checklist** verbindet MASVS-Kontrollen direkt mit passenden MASTG-Tests: OWASP MAS Checklist. Genau diese Verbindung, Kontrolle auf der einen Seite und passender Test auf der anderen, zieht sich später durch jedes Prüfmodul in diesem Tutorial.

---

3. Die absichtlich unsichere Beispiel-App „BKK Atomium“

Für dieses Tutorial habe ich eine kleine, absichtlich unsichere SwiftUI-App gebaut: eine fiktive Krankenkasse namens **BKK Atomium**. Es ist eine typische Versicherten-App ohne KI-Funktionen, aber mit genau den Kernfunktionen, die man von echten Krankenkassen-Apps kennt:

BKK Atomium App

Anmeldung mit Benutzername/Passwort

Face ID zum Entsperren

Zugriff auf ein Backend (Versicherten- und Leistungsdaten)

Postfach – digitale Nachrichten der Kasse (z. B. Leistungsbescheide, Mitgliedsbescheinigungen)

Dokumente laden – Bescheide/Abrechnungen aus dem Postfach herunterladen

Dokumente schicken – z. B. Arztrechnungen oder Rezepte an die Kasse einreichen

Deep Links (Nachricht direkt aus einer Push-Benachrichtigung öffnen)

PDF-Import (Beleg als PDF einreichen)

PDF-Export (Bescheid als PDF sichern/teilen)

Protokollierung (Logging)

Bewusst nicht enthalten: Cloud-KI, ein lokales KI-Modell, RAG-Suche. Dadurch fallen gegenüber einer KI-lastigen App zwar ein paar M4- und M6-Beispiele weg, etwa Prompt Injection oder KI-Telemetrie. Dafür rücken andere Risiken in den Vordergrund, die für Krankenkassen-Apps typischer sind: serverseitig gerenderte Postfach-Inhalte in einer WebView, Versichertennummern in URLs, und vor allem die besondere Schutzbedürftigkeit von Gesundheitsdaten nach Art. 9 DSGVO bei M6 und M9.

Verbindliche Datenschutz-Policy dieser App

Für BKK Atomium gilt eine Architekturvorgabe, die über generisches MASVS-STORAGE hinausgeht: das Backend ist die einzige Datenquelle für Diagnosen, Behandlungsdaten, Dokumente, Versicherungs- und Stammdaten. Diese Daten dürfen nicht dauerhaft lokal gespeichert werden, es darf keine Offline-Kopie geben, und nach Sitzungsende oder Logout müssen alle lokalen Sitzungsdaten entfernt sein. Das schließt neben den üblichen Speicherorten (UserDefaults, SwiftData, Dateien, Caches) auch NSCache, URLCache, App Groups/Shared Container, Notification-Inhalte und Logs mit ein. Die Keychain darf ausschließlich Auth-Artefakte wie kurzlebige Tokens und kryptografische Schlüssel enthalten, keine Gesundheits- oder Stammdaten. Zusätzlich gelten Vorgaben zur Lebensdauer sensibler Daten im Arbeitsspeicher (keine Singletons oder langlebigen ViewModels mit sensiblen Inhalten), zu ephemerem Netzwerk-Caching (URLSessionConfiguration.ephemeral, Cache-Control: no-store) sowie zu Anzeige, Zwischenablage und Drittanbieter-SDKs. Diese Policy ist strenger als das, was M9 in seiner bisherigen Form prüft (Kapitel 15) — dort noch offen, siehe Einordnung unten.

Jedes der zehn Prüfmodule in Teil 4 arbeitet mit einem Ausschnitt aus genau dieser App. So sieht die Projektstruktur aus:

```
example/
├──── BKKAtomium.xcodeproj/ ← Xcode-Projekt
└──── BKKAtomium/          ← BKK Atomium mit demonstrativen Schwachstellen
```

Die folgenden Ausschnitte sind didaktische Minimalbeispiele für die jeweilige Schwachstellenklasse. Sie sind nicht als wortgleiche Dateien in example/BKKAtomium enthalten. Die ausführbare Demo-App enthält vergleichbare Prüfanlässe in ihrer tatsächlichen SwiftUI-/SwiftData-Struktur; maßgeblich für reproduzierbare Befunde sind immer die realen Dateien und Zeilen im Repository.

```swift
// PostfachService.swift
let apiKey = "atk-live-9f3c2b1a7d"

// AuthenticationService.swift
UserDefaults.standard.set(accessToken, forKey: "accessToken")

// PostfachStore.swift
print("Nachricht geladen: \(message.subject) für Versicherte \(message.insuranceNumber)")
```

---

4. Scope und Prüfauftrag

Bevor Claude Code überhaupt irgendetwas prüft, legt der Auftraggeber Umfang und Grenzen fest. Diese Datei liegt im Projekt unter reports/scope.md und wird von jedem Skill als Erstes gelesen.

```
# Audit Scope – BKK Atomium App (iOS)

## Auftraggeber

Interner Security-Review.

## Im Umfang

- iOS-App „BKK Atomium“ (Swift-Quellcode unter `example/BKKAtomium`)
```

- Xcode-Projekt und Build-Einstellungen (`*example/BKKAtomium.xcodeproj*`)
- Swift-Package-Abhängigkeiten
- lokale Datenspeicherung (Keychain, SwiftData, Dateisystem, UserDefaults)
- Netzwerkkommunikation zum Backend (Versicherten-, Leistungs- und Postfachdaten)
- Authentifizierung, Face ID, Sessions
- Postfach, Dokumenten-Up-/Download, Deep Links, PDF-Import/-Export
- Einhaltung der Zero-Local-Persistence-Policy aus `*reports/data-policy.md*`

Nicht im Umfang

- das produktive Backend selbst (nur die Client-seitige Kommunikation damit)
- die Kernversicherungssysteme der Kasse
- Social Engineering
- Denial-of-Service-Tests
- aktive Angriffe gegen fremde/dritte Systeme

Regeln für den Agenten

Claude Code darf ausschließlich Dateien innerhalb dieses Repositories lesen, analysieren und Testskripte lokal ausführen. Keine Requests gegen produktive Endpunkte, keine Angriffe auf Infrastruktur Dritter.

Zielverzeichnis für Skripte

Wo Skripte einen `*TARGET*`-Parameter erwarten, Standardwert:

```text
example/BKKAtomium
```

Diese Trennung ist keine Formsache. Der Agent darf nur autorisierte Ziele prüfen, aktive Angriffe auf fremde Server sind ausdrücklich tabu. Jeder Skill in diesem Tutorial liest reports/scope.md, bevor er loslegt, und bricht ab, sobald eine Anfrage außerhalb des Scopes läge.

5. Testplan und Bedrohungsmodell

Bevor die einzelnen Module laufen, erstellt Claude Code ein Bedrohungsmodell. Dafür trägt der Agent zusammen:

schützenswerte Daten (Versichertennummer, Gesundheitsdaten in Bescheiden/Belegen, Auth-Token, biometrische Referenzen)

Benutzerrollen (angemeldet, nicht angemeldet)

Vertrauensgrenzen (Gerät ↔ Backend)

externe Dienste (Backend-API der Kasse, Push-/Benachrichtigungsdienst)

lokale Speicher (ausschließlich Keychain für Auth-Token; für BKK Atomium gilt eine Zero-Local-Persistence-Policy, siehe Kapitel 16 – Fachdaten wie Postfach-Nachrichten, Bescheide und Belege dürfen nicht dauerhaft auf dem Gerät liegen)

Schnittstellen (Deep Links, Share-Sheet, PDF-Import/-Export, Postfach)

mögliche Angreifer (böswillige App auf demselben Gerät, jemand mit physischem Zugriff, Angreifer im WLAN)

Auswirkungen eines erfolgreichen Angriffs. Hier besonders heikel: Rückschlüsse auf Diagnosen oder Behandlungen allein über den Titel eines Bescheids oder Belegs

```
Versicherte Person
  ↓
iOS-App (BKK Atomium)
   └──── Keychain (nur Auth-Token, keine Fachdaten)
  ↓
Backend (einzige Datenquelle für Versicherten-, Leistungs- und Postfachdaten)
```

Das Ergebnis landet in reports/threat-model.md, mit einer kurzen Einschätzung zu jeder Vertrauensgrenze, etwa: „Gerät ↔ Backend: TLS vorhanden, aber Zertifikatsvalidierung noch nicht geprüft, siehe M5." Jedes spätere Prüfmodul greift auf diese Datei zurück, statt das Modell jedes Mal neu zu erfinden.

6. Aufbau des Audit-Pakets

Für Claude Code empfiehlt sich aktuell, **Skills** unter .claude/skills/ anzulegen. Das ältere Verzeichnis .claude/commands/ funktioniert zwar weiterhin, aber Skills lassen sich sowohl manuell per Slash-Command als auch automatisch vom Agenten aufrufen. Claude Code Skills

```
mobile-security-audit/
├──── CLAUDE.md
├──── .claude/
│    ├──── settings.json
│    ├──── agents/
│    │    ├──── ios-security-reviewer.md
│    │    ├──── android-security-reviewer.md
│    │    └──── audit-verifier.md
│    └──── skills/
│         ├──── audit-mobile/
│         │    └──── SKILL.md
│         ├──── audit-credentials/      ← M1
│         │    └──── SKILL.md
│         ├──── audit-supply-chain/     ← M2
│         │    └──── SKILL.md
│         ├──── audit-auth/             ← M3
│         │    └──── SKILL.md
│         ├──── audit-validation/       ← M4
│         │    └──── SKILL.md
│         ├──── audit-network/          ← M5
│         │    └──── SKILL.md
│         ├──── audit-privacy/          ← M6
│         │    └──── SKILL.md
│         ├──── audit-binary/           ← M7
│         │    └──── SKILL.md
```

```
│       ├── audit-configuration/    ← M8
│       │   └── SKILL.md
│       ├── audit-storage/          ← M9
│       │   └── SKILL.md
│       ├── audit-data-lifecycle/   ← Kapitel 16, Zero-Local-Persistence-Policy
│       │   └── SKILL.md
│       ├── audit-crypto/           ← M10
│       │   └── SKILL.md
│       ├── pre-pentest-check/      ← Kapitel 22
│       │   └── SKILL.md
│       └── create-audit-report/
│           └── SKILL.md
├── references/
│   ├── mobile-top-10.md
│   ├── masvs-mapping.md
│   ├── mastg-ios-tests.md
│   ├── mastg-android-tests.md
│   └── severity-model.md
├── scripts/
│   ├── scan-secrets.sh
│   ├── inspect-ios-project.sh
│   ├── inspect-android-project.sh
│   ├── inspect-dependencies.sh
│   └── collect-evidence.sh
├── templates/
│   ├── scope.md
│   ├── finding.md
│   ├── test-result.md
│   └── final-report.md
└── reports/
    ├── findings/
    └── pre-pentest/        ← vertraulich, siehe Kapitel 22, gehört in .gitignore
                              sobald Externe das Repository mitlesen
```

CLAUDE.md ist dabei die wichtigste einzelne Datei im ganzen Paket: Sie liegt im Projekt-Wurzelverzeichnis und wird von Claude Code beim Arbeiten in diesem Verzeichnis automatisch gelesen, noch bevor überhaupt ein Skill aufgerufen wird. Sie fasst zusammen, was zuerst zu lesen ist (reports/scope.md, ggf. reports/threat-model.md), welche Befund-Zustände gelten, wie man das Audit startet und welche projektspezifische Vorgabe über das generische MASVS-STORAGE hinausgeht:

```
# Mobile Security Audit – BKK Atomium

Dieses Repository enthält das Claude-Code-Audit-Paket aus `claudeCodeOwasp-de.md`
sowie unter `example/BKKAtomium` die echte iOS-App, gegen die es laufen soll.

## Bevor du irgendetwas prüfst

1. Lies `reports/scope.md`. Prüfe ausschließlich, was dort im Umfang steht.
2. Lies `reports/threat-model.md`, falls vorhanden, sonst erstelle es zuerst
```

(siehe `.claude/skills/audit-mobile/SKILL.md`, Schritt 3).
3. Halte dich an die Zustände aus Kapitel 18 des Tutorials:
   Nicht geprüft, Automatisch erkannt, Manuelle Prüfung erforderlich,
   Bestätigt, Falsch positiv, Behoben, Nachtest bestanden.
   Ein automatisch erkannter Treffer ist nie automatisch „Bestätigt".

Wie man startet

```bash
/audit-mobile
```

führt das vollständige Audit aus (alle zehn Mobile-Top-10-Module plus, weil für diese App eine Zero-Local-Persistence-Policy gilt, `/audit-data-lifecycle`). Einzelne Module lassen sich auch separat aufrufen, z. B. `/audit-storage` oder `/audit-crypto`.

Projektspezifische Vorgabe

Für BKK Atomium gilt eine strengere Vorgabe als generisches MASVS-STORAGE: Diagnosen, Behandlungsdaten, Dokumente, Versicherungs- und Stammdaten dürfen nicht dauerhaft lokal gespeichert werden. Details stehen in `reports/data-policy.md` und werden von `.claude/skills/audit-data-lifecycle/SKILL.md` geprüft. Siehe auch Kapitel 16 des Tutorials.

Referenzmaterial

- Methodik und Beispiele je Modul: `claudeCodeOwasp-de.md`
- OWASP-Zuordnung: `references/mobile-top-10.md`, `references/masvs-mapping.md`, `references/mastg-ios-tests.md`
- Schweregrad-Definitionen: `references/severity-model.md`

Vertraulichkeit

`reports/pre-pentest/` ist ausschließlich intern (siehe Kapitel 22 des Tutorials). Sobald dieses Repository für einen externen Pentest-Anbieter sichtbar gemacht wird, muss dieser Ordner vorher per `.gitignore` oder separatem Export ausgeschlossen werden.

Jedes Prüfmodul in Teil 4 folgt derselben Struktur:

1. Risiko verstehen
2. Bezug zu MASVS
3. Passende MASTG-Tests auswählen
4. Unsicheren Beispielcode untersuchen
5. Automatische Prüfung ausführen
6. Manuelle Prüfung durchführen
7. Befund dokumentieren

8. Sichere Umsetzung zeigen
9. Nachtest ausführen

Ab jetzt zeigt jedes Kapitel diese Struktur an einem echten Ausschnitt von BKK Atomium, samt dem vollständigen Inhalt der jeweiligen SKILL.md. So siehst du genau, wie so eine Datei am Ende wirklich aussieht.

---

7. M1 – Unsachgemäße Verwendung von Zugangsdaten

Risiko verstehen

M1 fasst alles zusammen, was beim Umgang mit Zugangsdaten im Client schiefgehen kann: API-Schlüssel im Quellcode, Tokens in Konfigurationsdateien, Zugangsdaten in Logs oder Testdaten, Secrets in der Git-Historie, ungeeignete Speicherung in UserDefaults.

MASVS: MASVS-AUTH, MASVS-STORAGE **MASTG:** MASTG-TEST-0213 (Use of Hardcoded Cryptographic Keys in Code), MASTG-TEST-0214 (Hardcoded Cryptographic Keys in Files) und MASTG-TEST-0297 (Sensitive Data Exposure Through Logging APIs); für generische API-Secrets zusätzlich statische Quellcode- und Binary-Prüfung

Unsicheres Beispiel

```swift
// PostfachService.swift – BKK Atomium (insecure-ios)
import Foundation

struct PostfachService {
    // Absichtlich unsicher: API-Schlüssel des Push-/Benachrichtigungsdienstes
    // als Stringliteral im Client
    private let apiKey = "atk-live-9f3c2b1a7d"

    func fetchMessages(for insuranceNumber: String) async throws -> [PostfachMessage] {
        var request = URLRequest(url: URL(string:
"https://api.bkk-atomium.example/v1/postfach")!)
        request.setValue("Bearer \(apiKey)", forHTTPHeaderField: "Authorization")
        request.setValue(insuranceNumber, forHTTPHeaderField: "X-Insurance-Number")
        let (data, _) = try await URLSession.shared.data(for: request)
        return try JSONDecoder().decode([PostfachMessage].self, from: data)
    }
}
```

Jeder, der die App aus dem App Store lädt und das Binary öffnet, zum Beispiel mit strings oder class-dump, findet diesen Schlüssel im Klartext. Das gilt völlig unabhängig davon, wie gut Auth oder Netzwerk sonst abgesichert sind. Mit diesem einen Schlüssel ließe sich der Postfach-Endpunkt auch außerhalb der App ansprechen.

Der Skill

.claude/skills/audit-credentials/SKILL.md:

```
---
name: audit-credentials
```

description: *Prüft eine iOS/Android-Codebasis auf M1 (Improper Credential Usage) nach OWASP Mobile Top 10 – hartkodierte Secrets, unsichere Token-Speicherung, Zugangsdaten in Logs oder Git-Historie.*

Audit: Credentials (M1)

Voraussetzung

Lies zuerst `reports/scope.md`. Prüfe ausschließlich Dateien innerhalb des dort definierten Scopes (Standardziel: `example/BKKAtomium`).

Ablauf

1. Führe `scripts/scan-secrets.sh example/BKKAtomium` aus. Das Skript sucht per Regex nach Mustern wie `api[_-]?key`, `atk-[A-Za-z0-9-]{10,}`, `Bearer `, `password *=`, sowie nach bekannten Provider-Präfixen.
2. Durchsuche zusätzlich manuell:
   - Stringliterale in `.swift`-Dateien, die wie Secrets aussehen
   - `Info.plist` und `.xcconfig`-Dateien
   - `UserDefaults.standard.set(...)`-Aufrufe mit Token/Passwort-nahen Keys
   - `print(...)`- und `os_log(...)`-Aufrufe, die Tokens, Passwörter oder Versichertennummern ausgeben könnten
   - `git log -p -- '*.swift' '*.plist'` auf zuvor entfernte, aber historisch noch vorhandene Secrets (nur lesend, keine History-Rewrites vorschlagen)
3. Für jeden Treffer: Datei, Zeile und Kontext notieren, aber NICHT sofort als bestätigt einstufen. Status zunächst „Automatisch erkannt".
4. Ordne jeden Treffer MASVS-AUTH oder MASVS-STORAGE zu.
5. Schreibe jeden Treffer nach `templates/finding.md` in `reports/findings/M1-<n>.md`.
6. Schlage für jeden Treffer eine sichere Alternative vor (siehe Abschnitt „Sichere Umsetzung" in diesem Skill).

Sichere Umsetzung

- Keine dauerhaften Backend-/Dienst-Schlüssel im Client. Der Client erhält ausschließlich ein kurzlebiges, nutzergebundenes Session-Token vom Backend nach erfolgreicher Anmeldung.
- Nutzer-Tokens ausschließlich in der Keychain speichern (`kSecClassGenericPassword`), nie in `UserDefaults` oder Dateien.
- Keine Zugangsdaten oder Versichertennummern in Log-Ausgaben, auch nicht in Debug-Builds.

Nicht im Scope dieses Skills

- Laufzeitanalyse des Backends selbst
- Bewertung der Backend-seitigen Schlüsselverwaltung

Automatische Prüfung ausführen

/audit-credentials

scripts/scan-secrets.sh (Ausschnitt):

```
#!/usr/bin/env bash
set -euo pipefail
TARGET="${1:-example/BKKAtomium}"

grep -RnoE '(atk-[A-Za-z0-9_-]{8,}|api[_-]?key\s*=\s*"[^"]+"|Bearer [A-Za-z0-9._-]+)' \
  --include="*.swift" --include="*.plist" "$TARGET"
```

Manuelle Prüfung

Claude Code markiert den Treffer zunächst als „Automatisch erkannt“. Die manuelle Prüfung klärt dann, ob der String wirklich ein aktives Secret ist oder nur ein Platzhalter. Hier: ja, das Präfix atk-live- deutet auf einen produktiven Schlüssel hin, und der Kommentar „Absichtlich unsicher“ bestätigt, dass das genau die Absicht der Beispiel-App ist.

Befund

reports/findings/M1-001.md:

```
# MOB-M1-001: API-Schlüssel des Postfach-Dienstes im Client

- Schweregrad: Kritisch
- Status: Bestätigt
- OWASP Mobile Top 10: M1
- MASVS: MASVS-AUTH
- MASTG: statische Suche nach hartkodierten Secrets

## Betroffene Komponente

`PostfachService.swift`, Zeile 6

## Beschreibung

Der API-Schlüssel für den Postfach-/Benachrichtigungsdienst ist als
Stringliteral im Quellcode hinterlegt und damit Bestandteil jedes
ausgelieferten Binaries.

## Nachweis

```swift
private let apiKey = "atk-live-9f3c2b1a7d"
```

Auswirkung

Jeder, der das App-Binary extrahiert (z. B. via strings auf die IPA-Datei), erhält den vollen API-Schlüssel und könnte damit außerhalb der App auf Postfach-Endpunkte zugreifen, potenziell auch für andere Versicherte, falls der Schlüssel nicht nutzergebunden ist.
```

Empfehlung

Der Client erhält nach Anmeldung ausschließlich ein kurzlebiges, nutzergebundenes Session-Token vom Backend. Ein statischer Dienst-Schlüssel gehört ausschließlich serverseitig gespeichert.

Nachtest

Noch nicht durchgeführt.

Sichere Umsetzung

```swift
// PostfachService.swift – fixed-ios
struct PostfachService {
  private let backendBaseURL = URL(string: "https://api.bkk-atomium.example")!

  func fetchMessages(sessionToken: String) async throws -> [PostfachMessage] {
    var request = URLRequest(url: backendBaseURL.appendingPathComponent("v1/postfach"))
    request.setValue("Bearer \(sessionToken)", forHTTPHeaderField: "Authorization")
    let (data, _) = try await URLSession.shared.data(for: request)
    return try JSONDecoder().decode([PostfachMessage].self, from: data)
  }
}
```

Der statische Dienst-Schlüssel existiert jetzt nur noch serverseitig. Der Client kennt ausschließlich sein eigenes, kurzlebiges Session-Token, und die Versichertennummer wird serverseitig aus diesem Token abgeleitet, statt als eigener Header mitzufliegen.

Nachtest

/retest-mobile M1-001

Da das Repository keine separate fixed-ios-App enthält, ist dies ein **erwartetes Nachtestergebnis**, kein bereits ausgeführter Nachtest: Nach Übertragung des Fixes auf example/BKKAtomium wird scan-secrets.sh erneut ausgeführt. Erst wenn der ursprüngliche Treffer verschwunden ist und keine Regression vorliegt, darf der Status auf „Nachtest bestanden“ gesetzt werden.

---

8. M2 – Unsichere Lieferkette

Risiko verstehen

M2 betrifft alles, was über Abhängigkeiten ins Projekt kommt: Swift Packages, bekannte CVEs, nicht fixierte Versionen, unbekannte Paketquellen, unnötige SDKs, Build-Skripte und Plugins, Dependency Confusion, eine fehlende Software Bill of Materials (SBOM).

MASVS: MASVS-CODE **MASTG:** MASTG-TEST-0273 (Identify Dependencies with Known Vulnerabilities by Scanning Dependency Manager Artifacts) und MASTG-TEST-0275 (Dependencies with Known Vulnerabilities in the App’s SBOM)

Unsicheres Beispiel

```
// Package.swift – BKK Atomium (insecure-ios)
dependencies: [
   .package(url: "https://github.com/some-user/pdf-quicklook", from: "0.1.0"),
   .package(url: "https://github.com/some-user/push-notifications-lite", .branch("main"))
]
```

Zwei Risiken auf einen Blick: from: "0.1.0" erlaubt kompatible Updates innerhalb der von Swift Package Manager definierten Versionsspanne. .branch("main") bindet die App an eine veränderliche Referenz statt an eine unveränderliche Version oder einen Commit. Das sind Supply-Chain-Risiken, aber nicht automatisch **Dependency Confusion**. Dependency Confusion bezeichnet die Auflösung eines gleichnamigen Pakets aus einer unerwarteten Registry oder Quelle und muss separat geprüft werden.

Der Skill

.claude/skills/audit-supply-chain/SKILL.md:

```
---
name: audit-supply-chain
description: Prüft Swift-Package- und Gradle-Abhängigkeiten auf M2 (Inadequate Supply Chain
Security) – unfixierte Versionen, Branch-Referenzen, bekannte Schwachstellen, unnötige SDKs.
---

# Audit: Supply Chain (M2)

## Ablauf

1. Lies `reports/scope.md`.
2. Führe `scripts/inspect-dependencies.sh example/BKKAtomium` aus – listet
   alle Einträge aus `Package.resolved`/`Package.swift` bzw. dem
   Xcode-Projekt selbst auf.
3. Markiere jede Abhängigkeit, die:
   - über `.branch(...)` statt Tag/Commit referenziert wird
   - eine Versionsspanne statt einer fixierten Version nutzt (`from:` ohne
     Obergrenze)
   - von einem nicht offiziell verifizierten Publisher stammt
4. Gleiche Paketnamen/Versionen, wo möglich, gegen bekannte Advisories ab
   (z. B. GitHub Security Advisories, sofern über MCP/Web erreichbar).
5. Prüfe, ob eine SBOM existiert (`sbom.json`/`sbom.spdx`). Falls nicht:
   zunächst als Beobachtung dokumentieren. Ob daraus ein Befund entsteht und
   welcher Schweregrad angemessen ist, hängt von Abhängigkeiten,
   regulatorischen Vorgaben und Release-Prozess ab.
6. Dokumentiere Treffer wie in `audit-credentials` nach `templates/finding.md`
   in `reports/findings/M2-<n>.md`.
```

Automatische Prüfung

/audit-supply-chain

Befund (Ausschnitt)

MOB-M2-001: Ungesicherte Branch-Referenz in Package.swift

- Schweregrad: Hoch
- Status: Bestätigt
- MASVS: MASVS-CODE

Nachweis

.package(url: "https://github.com/some-user/push-notifications-lite", .branch("main"))

Auswirkung

Ein kompromittierter `*main*`-Branch dieser Push-Bibliothek würde bei der nächsten Build-Ausführung ungeprüft in die App übernommen – mit Zugriff auf eingehende Postfach-Benachrichtigungen.

Empfehlung

Auf ein festes, signiertes Release-Tag oder einen konkreten Commit-Hash pinnen und Updates bewusst über einen Pull Request durchführen.

Sichere Umsetzung

.package(url: "https://github.com/some-user/pdf-quicklook", exact: "1.4.2"),
.package(url: "https://github.com/some-user/push-notifications-lite", exact: "2.0.0")

---

9. M3 – Unsichere Authentifizierung und Autorisierung

Risiko verstehen

Geprüft werden: Tokenlebensdauer, Abmeldung, Face-ID-Integration, erneute Authentifizierung für kritische Aktionen, Rollen und Berechtigungen, Zugriff auf fremde Datensätze, und ob die Autorisierung nur clientseitig passiert.

MASVS: MASVS-AUTH **MASTG:** MASTG-TEST-0266 (References to APIs for Event-Bound Biometric Authentication) und MASTG-TEST-0267 (Runtime Use Of Event-Bound Biometric Authentication); Autorisierung und Session-Logik zusätzlich manuell prüfen

Unsicheres Beispiel

```
// PostfachMessageView.swift – insecure-ios
struct PostfachMessageView: View {
  let message: PostfachMessage

  var body: some View {
    VStack {
      Text(message.subject)
      Text(message.body)
```

```
        }
        // Face ID entsperrt nur den App-Zugang – nicht den Zugriff
        // auf EINZELNE Postfach-Nachrichten. Sobald die App entsperrt ist,
        // kann jede messageId per Deep Link direkt geöffnet werden,
        // ohne erneut zu prüfen, ob sie dem angemeldeten Versicherten gehört.
        .onAppear {
          AuditLog.record("Nachricht geöffnet: \(message.id)")
        }
    }
}

struct PostfachStore {
    func message(id: String) -> PostfachMessage? {
        // Kein Abgleich mit currentUser.insuranceNumber!
        allMessages.first { $0.id == id }
    }
}
```

Face ID schützt hier nur den App-Start, nicht die einzelne sensible Aktion. Und die Autorisierung prüft gar nicht erst, ob die angeforderte Postfach-Nachricht, zum Beispiel ein Leistungsbescheid mit Diagnosehinweis, überhaupt der angemeldeten Person gehört. In der Security-Sprache heißt das Broken Object Level Authorization, und hier passiert es rein clientseitig.

Der Skill

```
---
name: audit-auth
description: Prüft M3 (Insecure Authentication/Authorization) – Tokenlebensdauer, Face-ID-Nutzung, Re-Auth bei kritischen Aktionen, Zugriff auf fremde Datensätze.
---

# Audit: Authentication & Authorization (M3)

## Ablauf

1. Lies `reports/scope.md` und `reports/threat-model.md`.
2. Suche alle Stellen, die Face ID/Touch ID nutzen (`LAContext`), und prüfe,
   ob sensible Einzelaktionen (Bescheid einsehen, Beleg einreichen,
   Postfach-Nachricht löschen) erneut authentifizieren oder sich nur auf
   den App-Start verlassen.
3. Suche alle Funktionen, die Objekte per ID laden (`func \w+\(id:`), und
   prüfe, ob ein Abgleich mit der aktuell angemeldeten versicherten Person
   stattfindet.
4. Prüfe Tokenlebensdauer/Refresh-Logik: gibt es eine Ablaufzeit, wird sie
   serverseitig durchgesetzt oder nur clientseitig angezeigt?
5. Prüfe die Abmelde-Funktion: werden lokale Tokens UND serverseitige
   Sessions invalidiert? (Vollständigkeit des Logouts wird zusätzlich in
   `audit-data-lifecycle` geprüft: Cookies, WKWebView-Daten, Objektreferenzen.)
```

6. Dokumentiere Treffer nach `templates/finding.md` in
   `reports/findings/M3-<n>.md`.

Befund (Ausschnitt)

```
# MOB-M3-001: Fehlende Autorisierungsprüfung beim Postfach-Zugriff

- Schweregrad: Kritisch
- Status: Bestätigt
- MASVS: MASVS-AUTH

## Nachweis

`PostfachStore.message(id:)` lädt jede Nachricht allein anhand der ID,
ohne zu prüfen, ob `message.insuranceNumber == currentUser.insuranceNumber`.

## Auswirkung

Eine angemeldete Person kann über eine erratene oder aus einem Deep Link
entnommene `messageId` Postfach-Nachrichten anderer Versicherter lesen –
darunter potenziell Leistungsbescheide mit Rückschlüssen auf Diagnosen
oder Behandlungen (besondere Kategorie personenbezogener Daten, Art. 9 DSGVO).

## Empfehlung

Serverseitige Autorisierung bei jedem Abruf erzwingen; clientseitig
zusätzlich vor der Anzeige prüfen und im Zweifel ablehnen.
```

Sichere Umsetzung

```
struct PostfachStore {
  func message(id: String, for user: InsuredPerson) -> PostfachMessage? {
    allMessages.first { $0.id == id && $0.insuranceNumber == user.insuranceNumber }
  }
}

struct PostfachMessageView: View {
  let message: PostfachMessage
  @State private var isUnlockedForThisAction = false

  var body: some View {
    VStack { Text(message.subject); Text(message.body) }
      .task {
        isUnlockedForThisAction = await BiometricAuth
          .reauthenticate(reason: "Nachricht anzeigen")
      }
  }
}
```

---

10. M4 – Unzureichende Ein- und Ausgabevalidierung

Risiko verstehen

Geprüft werden: Deep Links, URL-Parameter, importierte Dateien wie PDF-Belege, Backend-Antworten, WebViews, und ob es beim Dokumenten-Upload und -Export überhaupt Größen- und Typbeschränkungen gibt.

MASVS: MASVS-PLATFORM, MASVS-CODE **MASTG:** MASTG-TEST-0370 (Missing Input Validation in Custom URL Scheme Handlers), MASTG-TEST-0371 (Missing Source Validation in Custom URL Scheme Handlers) und MASTG-TEST-0332 (Attacker-Controlled URI in WebViews)

Unsicheres Beispiel

```
// DeepLinkHandler.swift – insecure-ios
func handle(url: URL) {
  // Keine Validierung von Host/Scheme/Herkunft
  let messageId = url.pathComponents.last ?? ""
  openPostfachMessage(id: messageId)
}

// PostfachMessageDetailView.swift
struct PostfachMessageDetailView: View {
  let message: PostfachMessage

  var body: some View {
    // Der Nachrichtentext kommt als HTML vom Backend (z. B. für
    // Formatierung von Bescheiden) und wird ungefiltert gerendert.
    HTMLWebView(html: message.bodyHTML)
  }
}

// BelegUploadView.swift
func importBeleg(at url: URL) throws {
  let data = try Data(contentsOf: url) // keine Größen-, Typ- oder Seitenzahlprüfung
  try uploadService.upload(pdf: data, for: currentInsuranceNumber)
}
```

Ein Deep Link wie bkkatomium://postfach/<beliebige-id> wird geöffnet, ohne Scheme oder Host zu prüfen. Der HTML-Body einer Postfach-Nachricht landet direkt in einer WebView (HTMLWebView, intern WKWebView.loadHTMLString). Kommt dort, etwa über eine kompromittierte Backend-Komponente oder einen fehlerhaften Bescheid-Generator, unbemerkt manipulierter oder falsch escapeter HTML- beziehungsweise JavaScript-Inhalt an, führt die App ihn im Kontext der WebView einfach aus. Beim PDF-Upload fehlt zusätzlich jede Grenze für Dateigröße, Seitenzahl oder den tatsächlichen Dateityp.

Der Skill

```
---
name: audit-validation
description: Prüft M4 (Insufficient Input/Output Validation) – Deep Links, WebView-Rendering
von Backend-Inhalten, PDF-Import/-Export, fehlende Größen-/Typbeschränkungen.
---
```

Audit: Input/Output Validation (M4)

Ablauf

1. Lies `reports/scope.md`.
2. Suche alle Deep-Link-/Universal-Link-Handler; prüfe Validierung von Scheme, Host und Pfad-Parametern vor deren Verwendung.
3. Suche alle `WKWebView`/`loadHTMLString`-Aufrufe, die Backend- oder Nachrichteninhalte direkt rendern; prüfe, ob JavaScript deaktiviert ist (sofern nicht benötigt) und ob der Inhalt vor dem Rendern bereinigt wird.
4. Suche Datei-Import-/-Exportpfade (PDF-Beleg-Upload, Bescheid-Download); prüfe Größen-, Typ- und Seitenzahlprüfung vor der Weiterverarbeitung bzw. vor dem Export/Teilen.
5. Prüfe, ob beim PDF-Export der Dateiname/Pfad aus Backend-Daten ungeprüft übernommen wird (Path-Traversal-Risiko).
6. Dokumentiere Treffer nach `templates/finding.md` in `reports/findings/M4-<n>.md`.

Manuelle Prüfung

Testschritt (WebView-Rendering):
1. Über einen Test-Account eine Postfach-Nachricht anlegen, deren bodyHTML folgendes enthält:
<img src=x onerror="alert(document.cookie)">
2. Nachricht in der App öffnen.
3. Beobachten, ob das Skript in der WebView ausgeführt wird.

Testschritt (Deep Link):
1. bkkatomium://postfach/<messageId eines fremden Kontos> in Safari öffnen.
2. Beobachten, ob die App die Nachricht ohne weitere Prüfung anzeigt.

Befund (Ausschnitt)

MOB-M4-001: Ungefiltertes HTML-Rendering von Postfach-Nachrichten

- Schweregrad: Hoch
- Status: Manuelle Prüfung erforderlich
- MASVS: MASVS-PLATFORM

Nachweis

`PostfachMessageDetailView` rendert `message.bodyHTML` direkt über `HTMLWebView` (`WKWebView.loadHTMLString`), ohne Bereinigung und ohne JavaScript zu deaktivieren.

Auswirkung

Fehlerhaft escapeter oder manipulierter HTML-Inhalt in einer

Postfach-Nachricht könnte JavaScript im Kontext der App-WebView ausführen.

Empfehlung

JavaScript in dieser WebView vollständig deaktivieren (`*WKWebViewConfiguration.preferences.javaScriptEnabled = false*`), Inhalt serverseitig und clientseitig sanitisieren, wo möglich auf reinen Text oder ein streng eingeschränktes Markup-Subset umstellen.

Sichere Umsetzung

```
func handle(url: URL) {
  guard url.scheme == "bkkatomium", url.host == "postfach" else { return }
  guard let messageId = url.pathComponents.last, isValidMessageId(messageId) else { return }
  openPostfachMessage(id: messageId)
}

struct HTMLWebView: UIViewRepresentable {
  let html: String
  func makeUIView(context: Context) -> WKWebView {
    let config = WKWebViewConfiguration()
    config.preferences.javaScriptEnabled = false
    return WKWebView(frame: .zero, configuration: config)
  }
  func updateUIView(_ webView: WKWebView, context: Context) {
    webView.loadHTMLString(HTMLSanitizer.sanitize(html), baseURL: nil)
  }
}

func importBeleg(at url: URL) throws {
  let data = try Data(contentsOf: url)
  guard data.count <= maxBelegSizeBytes, PDFValidator.isValidPDF(data) else {
    throw BelegImportError.invalidFile
  }
  try uploadService.upload(pdf: data, for: currentInsuranceNumber)
}
```

---

11. M5 – Unsichere Kommunikation

Risiko verstehen

Geprüft werden: HTTP statt HTTPS, App Transport Security (ATS), Zertifikatsvalidierung, unsichere Ausnahmen, sensible Daten in URLs, Timeouts, Wiederholungslogik, und was im Netzwerk-Logging landet.

MASVS: MASVS-NETWORK **MASTG:** MASTG-TEST-0321 (Hardcoded HTTP URLs), MASTG-TEST-0322 (App Transport Security Configurations Allowing Cleartext Traffic), MASTG-TEST-0342 (References to Weak ATS TLS Policy Exceptions in Info.plist), MASTG-TEST-0348 (Insecure TLS Protocols in Network Traffic) und MASTG-TEST-0396 (References to URLSessionDelegate Bypassing Certificate Validation)

Unsicheres Beispiel

```
<!-- Info.plist – insecure-ios -->
<key>NSAppTransportSecurity</key>
<dict>
  <key>NSAllowsArbitraryLoads</key>
  <true/>
</dict>
```

```
// BackendService.swift
func fetchBescheid(id: String, insuranceNumber: String, token: String) async throws -> Data {
  let url = URL(string: "https://api.bkk-atomium.example/bescheide?id=\
(id)&versichertennummer=\(insuranceNumber)&token=\(token)")!
  let (data, _) = try await URLSession.shared.data(from: url)
  return data
}
```

NSAllowsArbitraryLoads schaltet ATS komplett aus, auch für Hintergrunddienste, die eigentlich HTTPS erzwingen sollten. Dazu landen Versichertennummer und Auth-Token als URL-Parameter, und die können in Server-Logs, Proxys und Browser-Historien auftauchen. Kombiniert mit dem Bescheid-Typ ergibt das bei einer Krankenkasse einen besonders sensiblen Datensatz.

Der Skill

```
---
name: audit-network
description: Prüft M5 (Insecure Communication) – ATS-Konfiguration, Zertifikatsvalidierung, sensible Daten (insb. Versichertennummer/Token) in URLs, Netzwerklogging, Caching sensibler Antworten.
---

# Audit: Network (M5)

## Ablauf

1. Lies `reports/scope.md`.
2. Führe `scripts/inspect-ios-project.sh example/BKKAtomium` aus und prüfe `Info.plist` auf `NSAllowsArbitraryLoads` und weitere ATS-Ausnahmen (`NSExceptionDomains`); jede pauschale Ausnahme ist ein Befund, gezielte Ausnahmen für einzelne, begründete Domains genauer bewerten.
3. Suche `URLSession`-Aufrufe mit `http://` statt `https://`.
4. Suche Delegate-Implementierungen, die Zertifikatsprüfungen umgehen (`URLSession:didReceiveChallenge:` mit `.useCredential` ohne Prüfung).
5. Suche sensible Werte (Token, Versichertennummer) in URL-Query-Parametern statt in Headern/Body.
6. Prüfe, ob für Endpunkte mit sensiblen Antworten `URLSessionConfiguration.ephemeral` und `Cache-Control: no-store` vorgesehen sind (Basisprüfung; die vollständige Tiefenprüfung dazu läuft in `audit-data-lifecycle`).
7. Prüfe Logging-Code auf mitgeloggte Request-/Response-Bodies.
```

8. Dokumentiere Treffer nach `templates/finding.md` in `reports/findings/M5-<n>.md`.

Befund (Ausschnitt)

```
# MOB-M5-001: Versichertennummer und Token als URL-Parameter

- Schweregrad: Hoch
- Status: Bestätigt
- MASVS: MASVS-NETWORK

## Nachweis

`.../bescheide?id=\(id)&versichertennummer=\(insuranceNumber)&token=\(token)`

## Auswirkung

Versichertennummer und Token können in Server-/Proxy-Logs sowie in
etwaigen Client-seitigen Verlaufsspeichern landen. In Kombination mit
dem Bescheid-Endpunkt lassen sich daraus zusätzlich Rückschlüsse auf
den Anlass des Bescheids ziehen.

## Empfehlung

Versichertennummer serverseitig aus dem Session-Token ableiten statt
als Parameter zu übertragen; Token ausschließlich im
`Authorization`-Header übertragen, niemals als Query-Parameter.
```

Sichere Umsetzung

```
<key>NSAppTransportSecurity</key>
<dict>
  <key>NSAllowsArbitraryLoads</key>
  <false/>
</dict>
```

```
func fetchBescheid(id: String, token: String) async throws -> Data {
  var request = URLRequest(url: URL(string: "https://api.bkk-atomium.example/bescheide?
id=\(id)")!)
  request.setValue("Bearer \(token)", forHTTPHeaderField: "Authorization")
  let (data, _) = try await URLSession.shared.data(for: request)
  return data
}
```

---

12. M6 – Unzureichender Datenschutz

Risiko verstehen

Geprüft werden: welche Daten überhaupt erhoben werden, Tracking-SDKs, Privacy Manifest, Einwilligungen, Aufbewahrungsfristen, Export und Löschung, sensible Inhalte in Telemetrie und Logs. Bei einer Krankenkassen-App kommt noch etwas dazu: Gesundheitsdaten gelten nach **Art. 9 DSGVO** als besondere Kategorie personenbezogener Daten, mit höheren Anforderungen an Rechtsgrundlage, Zweckbindung und Schutzmaßnahmen.

MASVS: MASVS-PRIVACY **MASTG:** MASTG-TEST-0281 (Undeclared Known Tracking Domains), ergänzt um die manuelle Analyse von PrivacyInfo.xcprivacy, Telemetrie, Rechtsgrundlage und Datenminimierung

Unsicheres Beispiel

```
// AnalyticsService.swift – insecure-ios
func trackMessageOpened(_ message: PostfachMessage) {
  Analytics.log(event: "postfach_message_opened", properties: [
    "subject": message.subject,          // z. B. "Bescheid: Psychotherapie-Antrag"
    "category": message.category,        // z. B. "Leistungsantrag"
    "insurance_number": currentUser.insuranceNumber
  ])
}
```

Der komplette Nachrichtenbetreff, der bei einer Krankenkasse oft schon auf eine Behandlungsart oder Diagnose hindeutet, dazu die Kategorie und die Versichertennummer: All das landet in einem Analytics-Event, das an einen Drittanbieter geht. Ohne sichtbare Einwilligung, ohne Deklaration im Privacy Manifest. Damit geht es hier unmittelbar um Gesundheitsdaten im Sinne von Art. 9 DSGVO.

Der Skill

```
---
name: audit-privacy
description: Prüft M6 (Inadequate Privacy Controls) – Datenerhebung, Weitergabe an Dritte, Privacy Manifest, Einwilligung, Aufbewahrung; besonderes Augenmerk auf Gesundheitsdaten (Art. 9 DSGVO).
---

# Audit: Privacy (M6)

## Ablauf

1. Lies `reports/scope.md`.
2. Liste alle Analytics-/Tracking-Aufrufe und die darin übertragenen Felder auf.
3. Gleiche jedes Feld gegen `PrivacyInfo.xcprivacy` ab: ist die Datenkategorie
   dort deklariert?
4. Prüfe insbesondere, ob Nachrichtenbetreffs, Bescheid-Kategorien,
   Diagnose- oder Behandlungshinweise irgendwo in Telemetrie, Crash-Reports
   oder Logs landen – auch indirekt (z. B. über Dateinamen von Exporten).
5. Prüfe, ob für die Verarbeitung dieser besonderen Datenkategorie eine
   erkennbare Rechtsgrundlage/Einwilligung vorgesehen ist.
6. Prüfe, ob Export- und Löschfunktionen für Versichertendaten existieren.
```

7. Prüfe eingebundene Drittanbieter-SDKs darauf, ob sie standardmäßig Bildschirminhalte oder Netzwerkdaten automatisch erfassen (Vertiefung dazu in `audit-data-lifecycle`).
8. Dokumentiere Treffer nach `templates/finding.md` in `reports/findings/M6-<n>.md`.

Befund (Ausschnitt)

```
# MOB-M6-001: Gesundheitsbezogene Daten im Analytics-Event

- Schweregrad: Kritisch
- Status: Bestätigt
- MASVS: MASVS-PRIVACY

## Nachweis

`AnalyticsService.trackMessageOpened` überträgt Nachrichtenbetreff,
Bescheid-Kategorie und Versichertennummer an den Analytics-Anbieter.

## Auswirkung

Der Betreff eines Bescheids lässt häufig bereits Rückschlüsse auf
Behandlungsart oder Diagnose zu. Diese Daten gelten nach Art. 9 DSGVO
als besondere Kategorie personenbezogener Daten; ihre Übertragung an
einen Drittanbieter ohne erkennbare Rechtsgrundlage und ohne Deklaration
im Privacy Manifest ist ein schwerwiegender Befund.

## Empfehlung

Nur anonymisierte Ereignisnamen ohne Inhalte oder Versichertennummer
übertragen; Deklaration im Privacy Manifest ergänzen oder Datenumfang
grundsätzlich reduzieren; Rechtsgrundlage für jede Verarbeitung
gesundheitsbezogener Daten dokumentieren.
```

Sichere Umsetzung

```
func trackMessageOpened(_ message: PostfachMessage) {
  // Für geschützte Postfach-Inhalte wird kein Analytics-Event erzeugt.
  // Auch technische Kategorien können Rückschlüsse auf Leistungen oder
  // Behandlungen ermöglichen.
}
```

---

13. M7 – Unzureichender Binärschutz

Risiko verstehen

Geprüft werden: Debug-Symbole, sensible Strings im App-Paket, wie leicht sich die App manipulieren lässt, Reverse Engineering, Jailbreak- und Root-Risiken, Integritätsprüfungen, Debugging in Release-Builds.

MASVS: MASVS-RESILIENCE **MASTG:** MASTG-TEST-0261 (Debuggable Entitlement Enabled), MASTG-TEST-0219 (Testing for Debugging Symbols), MASTG-TEST-0240/0241 (Jailbreak Detection) und MASTG-TEST-0401/0402 (Debugging Detection APIs)

Wichtig an dieser Stelle: Schutzmaßnahmen erschweren Angriffe, machen eine Client-App aber niemals vollständig immun gegen Analyse. Wer physischen Zugriff auf das Binary hat, kann es grundsätzlich untersuchen. Das Ziel ist, den Aufwand deutlich zu erhöhen, nicht Unmöglichkeit zu versprechen.

Unsicheres Beispiel

```
// AppDelegate.swift – insecure-ios
func applicationDidFinishLaunching(_ application: UIApplication) {
   #if DEBUG
   print("Running in debug mode with verbose logging")
   #endif
   // Kein Jailbreak-Check, keine Integritätsprüfung
}
```

Build-Einstellungen im Xcode-Projekt zeigen zusätzlich ENABLE_BITCODE = NO, DEBUG_INFORMATION_FORMAT = dwarf auch im Release-Build (statt dwarf-with-dsym getrennt archiviert) und MTL_ENABLE_DEBUG_INFO = YES im Release.

Der Skill

```
---
name: audit-binary
description: Prüft M7 (Insufficient Binary Protections) – Debug-Einstellungen in Release-Builds,
fehlende Jailbreak-/Integritätsprüfungen, sensible Strings im Binary.
---

# Audit: Binary Protections (M7)

## Ablauf

1. Lies `reports/scope.md`.
2. Führe `scripts/inspect-ios-project.sh example/BKKAtomium` aus und prüfe
   die Release-Konfiguration auf Debug-Flags, die dort nicht hingehören.
3. Prüfe, ob eine Jailbreak-Erkennung oder zumindest ein Risikohinweis für
   kritische Aktionen (z. B. Beleg-Upload) existiert (kein absoluter Schutz,
   aber Signal).
4. Falls ein gebautes Archiv vorliegt: `strings`/`otool -L` auf das Binary
   anwenden, um verbleibende sensible Strings/Debug-Symbole zu finden
   (nur gegen selbst gebaute, autorisierte Artefakte, siehe Scope).
5. Dokumentiere Treffer nach `templates/finding.md` in
   `reports/findings/M7-<n>.md` und weise explizit darauf hin, dass diese
   Maßnahmen Erschwerung, keine Garantie bedeuten.
```

Befund (Ausschnitt)

```
# MOB-M7-001: Debug-Informationen im Release-Build aktiv

- Schweregrad: Mittel
```

- Status: Bestätigt
- MASVS: MASVS-RESILIENCE

Nachweis

Release-Konfiguration in *`project.pbxproj`* setzt weiterhin *`MTL_ENABLE_DEBUG_INFO = YES`* und verbose Logging über *`#if DEBUG`* hinaus in manchen Pfaden.

Empfehlung

Release-Konfiguration von Debug-Konfiguration klar trennen, verbose Logging vollständig aus Release-Builds entfernen, dSYM getrennt archivieren statt eingebettet auszuliefern.

---

14. M8 – Sicherheitsfehlkonfiguration

Risiko verstehen

Geprüft werden: Entitlements, App-Berechtigungen, Debug-Konfiguration, ATS-Ausnahmen, URL-Schemes, Universal Links, Background Modes, und ob irgendwo ein Testserver im Release-Build hängengeblieben ist.

MASVS: MASVS-PLATFORM, MASVS-CODE **MASTG:** MASTG-TEST-0362 (Entitlements for Unjustified Capability Exposure), MASTG-TEST-0321 (Hardcoded HTTP URLs), MASTG-TEST-0322 (ATS Configurations Allowing Cleartext Traffic) und MASTG-TEST-0342 (Weak ATS TLS Policy Exceptions)

Unsicheres Beispiel

```
<!-- BKKAtomium.entitlements – insecure-ios -->
<key>com.apple.developer.icloud-container-identifiers</key>
<array><string>iCloud.example.bkk-atomium</string></array>
<key>keychain-access-groups</key>
<array><string>*</string></array>
```

```
// Config.swift
#if DEBUG
let backendURL = URL(string: "http://192.168.1.50:8080")!
#else
let backendURL = URL(string: "http://192.168.1.50:8080")! // Testserver blieb im Release-Zweig
#endif
```

keychain-access-groups: ["*"] teilt die Keychain-Gruppe mit praktisch jeder App desselben Entwicklerteams. Das ist deutlich weiter gefasst als nötig, gerade bei einer App, die Auth-Tokens und biometrisch geschützte Referenzen für Gesundheitsdaten verwaltet. Der Testserver aus der Entwicklung wurde außerdem versehentlich auch im Release-Pfad belassen.

Der Skill

---
name: *audit-configuration*
description: *Prüft M8 (Security Misconfiguration) – Entitlements, Berechtigungen, URL-Schemes, Testserver-Reste, Debug-Konfiguration im Release-Build.*

Audit: Configuration (M8)

Ablauf

1. Lies `*reports/scope.md*`.
2. Prüfe `*.entitlements*`-Dateien auf übermäßig weite Berechtigungen (z. B. Wildcard-Keychain-Groups, unnötige Background Modes).
3. Prüfe `*Info.plist*` auf registrierte URL-Schemes/Universal-Link-Domains und ob sie zum aktuellen Funktionsumfang (Postfach, Deep Links) passen.
4. Suche nach Konfigurationswerten (IP-Adressen, `*http://*`, `*localhost*`, interne Hostnamen), die auch im Release-Pfad aktiv sind.
5. Dokumentiere Treffer nach `*templates/finding.md*` in `*reports/findings/M8-<n>.md*`.

Befund (Ausschnitt)

MOB-M8-001: Interner Testserver auch im Release-Build aktiv

- Schweregrad: Hoch
- Status: Bestätigt
- MASVS: MASVS-PLATFORM

Nachweis

`*Config.swift*` verwendet in beiden Build-Konfigurationen dieselbe interne, unverschlüsselte Testserver-Adresse.

Empfehlung

Getrennte Konfigurationswerte pro Build-Konfiguration über `*.xcconfig*`-Dateien, Release zeigt ausschließlich auf die produktive, TLS-gesicherte API.

---

15. M9 – Unsichere Datenspeicherung

Risiko verstehen

Geprüft werden: UserDefaults, SwiftData/Core Data, Dateien und Caches, Keychain, Backups, Zwischenablage, Screenshots im App Switcher, Benachrichtigungen, temporäre Beleg- und Bescheid-Dateien aus Import und Export.

MASVS: MASVS-STORAGE **MASTG:** MASTG-TEST-0296/0297 (Sensitive Data Exposure in Logs), MASTG-TEST-0299/0300/0301/0302/0303 (Data Protection und unverschlüsselte lokale Speicherung), MASTG-TEST-0215/0298 (Backup-Ausschluss), MASTG-TEST-0277/0278/0279/0280 (Pasteboard) und MASTG-TEST-0290 (App-Switcher-Screenshots)

Das ist der generische MASVS-Basisumfang für M9. Für BKK Atomium gilt zusätzlich eine strengere, projektspezifische Vorgabe: keine dauerhafte lokale Speicherung von Fachdaten überhaupt, auch nicht verschlüsselt. Wie dieser Skill um diese Vorgabe ergänzt wird, zeigt Kapitel 16.

Unsicheres Beispiel

```
// AuthenticationService.swift – insecure-ios
func handleLoginSuccess(token: String) {
  UserDefaults.standard.set(token, forKey: "accessToken") // unverschlüsselt, Backup-fähig
}

// BescheidExportService.swift
func exportBescheidPDF(_ data: Data, fileName: String) throws -> URL {
  let url = FileManager.default.temporaryDirectory
    .appendingPathComponent(fileName) // unverschlüsselt im /tmp, für Backup nicht ausgeschlossen
  try data.write(to: url)
  return url
}
```

UserDefaults ist kein geeigneter Geheimnisspeicher: Werte liegen als Preferences-Datei im App-Container und profitieren nicht von den Zugriffskontrollen der Keychain. Ob und wie diese Datei in ein Geräte- oder Cloud-Backup gelangt, hängt von Plattform-, Backup- und Gerätekonfiguration ab; die Sicherheitsentscheidung darf sich nicht auf einen pauschalen Backup-Ausschluss stützen. Auch Dateien im temporären Verzeichnis sind nicht automatisch dauerhaft oder backup-fähig. Das reale Risiko besteht darin, dass sensible Exporte als Klartext geschrieben, zu lange aufbewahrt oder bei Abbruch/Fehler nicht zuverlässig entfernt werden.

Der Skill

```
---
name: audit-storage
description: Prüft M9 (Insecure Data Storage) – UserDefaults, SwiftData, Dateisystem, Keychain, Backups, Zwischenablage, temporäre Beleg-/Bescheid-Exporte.
---

# Audit: Storage (M9)

Das ist der generische MASVS-STORAGE-Basisumfang. Für dieses Repository gilt zusätzlich `reports/data-policy.md` (Zero-Local-Persistence-Policy): keine
```

dauerhafte lokale Speicherung von Fachdaten überhaupt, auch nicht verschlüsselt. Die Tiefenprüfung dazu läuft in *`audit-data-lifecycle`* – dieser Skill hier deckt die klassischen, produktübergreifenden Storage-Fehler ab.

Ablauf

1. Lies *`reports/scope.md`* und *`reports/data-policy.md`*.
2. Suche *`UserDefaults.standard.set(...)`*-Aufrufe mit sicherheitsrelevanten Keys (Token, Passwort, biometrische Referenz).
3. Suche Datei-/Cache-Schreibvorgänge (*`FileManager`*, *`write(to:)`*) und prüfe, ob *`.completeFileProtection`* bzw. Verschlüsselung genutzt wird, insbesondere für exportierte Bescheid-PDFs und importierte Belege. Prüfe zusätzlich, ob solche Dateien überhaupt dauerhaft existieren dürfen (siehe *`reports/data-policy.md`*, Abschnitt 1–2) oder nur für die Dauer einer Systeminteraktion (z. B. Share-Sheet) und danach sofort gelöscht werden.
4. Prüfe *`NSFileProtectionKey`*/*`isExcludedFromBackup`*-Einstellungen für sensible Dateien.
5. Prüfe Copy-to-Clipboard-Stellen für sensible Inhalte.
6. Prüfe, ob sensible Bildschirminhalte (Postfach, Bescheide) im App-Switcher-Snapshot sichtbar bleiben (fehlender Privacy-Screen beim Backgrounding).
7. Prüfe SwiftData/Core-Data-Modelle darauf, ob sie Fachdaten (Postfach, Bescheide, Stammdaten) über die aktuelle Sitzung hinaus persistieren – nach der Zero-Local-Persistence-Policy ist das grundsätzlich unzulässig, unabhängig davon, ob verschlüsselt.
8. Dokumentiere Treffer nach *`templates/finding.md`* in *`reports/findings/M9-<n>.md`*.

Befund

reports/findings/M9-001.md:

MOB-M9-001: Authentifizierungstoken in UserDefaults

Einstufung

- Schweregrad: Hoch
- Status: Bestätigt
- OWASP Mobile Top 10: M9
- MASVS: MASVS-STORAGE
- MASTG: MASTG-TEST-0300 und MASTG-TEST-0301

Betroffene Komponente

`AuthenticationService.swift`

Beschreibung

Die App speichert das Authentifizierungstoken in *`UserDefaults`* statt in der Keychain.

Nachweis

Auslesen des Simulator-Containers:

```bash
plutil -p ~/Library/Developer/CoreSimulator/Devices/<UDID>/data/Containers/Data/Application/<APP-UUID>/Library/Preferences/example.bkk-atomium.plist
```

zeigt den Klartext-Token unter dem Key accessToken.

Auswirkung

Ein Angreifer mit Zugriff auf den App-Container (z. B. über ein Geräte-Backup oder ein Jailbreak) kann das Token direkt auslesen und damit Zugriff auf Postfach-Nachrichten und Bescheide der versicherten Person erlangen.

Empfehlung

Speicherung in der Keychain mit kSecAttrAccessibleWhenUnlockedThisDeviceOnly, zusätzlich serverseitige Begrenzung der Tokenlebensdauer.

Nachtest

Noch nicht durchgeführt.

Sichere Umsetzung

Der Token-Fix ist unabhängig vom Projekt immer richtig: Keychain statt `UserDefaults`. Beim PDF-Export reicht „verschlüsselt speichern“ bei BKK Atomium aber nicht aus, weil die Zero-Local-Persistence-Policy (Kapitel 16) *jede* dauerhafte lokale Kopie von Bescheid-Inhalten verbietet, auch eine verschlüsselte. Der Export darf die Datei deshalb nur für die Dauer des Share-Sheets auf der Platte halten und muss sie danach sofort wieder löschen:

```swift
func handleLoginSuccess(token: String) {
  KeychainStore.set(token, forKey: "accessToken",
                    accessibility: .whenUnlockedThisDeviceOnly)
}

func exportBescheidPDF(_ data: Data, fileName: String) async throws {
  let url = FileManager.default.temporaryDirectory.appendingPathComponent(fileName)
  try data.write(to: url, options: .completeFileProtection)
  try (url as NSURL).setResourceValue(true, forKey: .isExcludedFromBackupKey)
  defer { try? FileManager.default.removeItem(at: url) } // sofort nach dem Teilen löschen
```

```
    await presentShareSheetAndWaitForCompletion(for: url)
}
```

presentShareSheetAndWaitForCompletion muss erst zurückkehren, nachdem der Completion-Handler des Share-Sheets Erfolg, Abbruch oder Fehler gemeldet hat. Nur dann läuft das defer zum richtigen Zeitpunkt. Ein bloßes Präsentieren der UI und sofortiges Zurückkehren würde die Datei zu früh löschen. Zusätzlich muss die Bereinigung bei App-Abbruch und beim nächsten Start defensiv wiederholt werden.

Nachtest

/retest-mobile M9-001

Dies ist das erwartete Nachtestverfahren: Nach Implementierung wird der Container erneut ausgelesen. „Nachtest bestanden“ darf erst gesetzt werden, wenn accessToken nicht mehr in der Preferences-Datei erscheint, der Keychain-Eintrag korrekt geschützt ist und Login sowie Logout weiterhin funktionieren.

---

16. Projektspezifische Zusatzprüfung: Zero-Local-Persistence-Policy

Warum ein eigenes Modul

Die Mobile Top 10 und MASVS-STORAGE prüfen, *wie* lokal gespeicherte Daten geschützt sind: verschlüsselt, in der Keychain, backup-ausgeschlossen. Für BKK Atomium reicht das nicht, weil die Vorgabe eine Ebene früher ansetzt: Fachdaten wie Diagnosen, Behandlungsdaten, Dokumente, Versicherungs- und Stammdaten dürfen gar nicht erst dauerhaft auf dem Gerät landen, egal wie gut sie dort geschützt wären. Das Backend bleibt die einzige Datenquelle, und nach Sitzungsende oder Logout muss lokal nichts mehr davon übrig sein.

Das betrifft mehrere Bereiche gleichzeitig, die in den generischen Modulen nur teilweise oder gar nicht vorkommen:

Speicherorte, die M9 nicht abdeckt: NSCache, URLCache, Widgets, App Groups und Shared Container, Inhalte von Push-Benachrichtigungen.

Lebensdauer im Arbeitsspeicher: Wie lange halten Singletons, globale Variablen oder langlebige ViewModels sensible Daten fest? Werden Objektreferenzen beim Verlassen einer Ansicht freigegeben?

Netzwerk-Caching: Landen Backend-Antworten mit Gesundheitsdaten im URLCache, obwohl eine ephemere Session-Konfiguration und Cache-Control: no-store das verhindern sollten?

Anzeige und Betriebssystemfunktionen: App-Switcher-Verdeckung, Screenshot-/Recording-Erkennung, Sperrbildschirm-Inhalte, Copy-Paste-Schutz.

Logging und Drittanbieter: Erfassen Analytics-, Monitoring- oder Crash-Reporting-SDKs automatisch Bildschirminhalte oder Netzwerkdaten, ohne dass eigener Code das veranlasst?

Logout-Vollständigkeit: Werden neben dem Token auch Cookies, WKWebView-Daten und temporäre Dateien entfernt?

MASVS: MASVS-STORAGE, MASVS-PRIVACY, MASVS-NETWORK (dieses Modul kombiniert bewusst mehrere MASVS-Gruppen, weil die zugrunde liegende Vorgabe selbst modulübergreifend ist) **MASTG:** aktuelle iOS-Atomic-Tests aus M9, insbesondere MASTG-TEST-0290 und MASTG-TEST-0296 bis 0303, ergänzt um manuelle Prüfung von Arbeitsspeicher-Lebensdauer und Netzwerk-Caching; dafür gibt es keinen einzelnen Atomic Test

Unsicheres Beispiel

```swift
// PostfachViewModel.swift – insecure-ios
final class PostfachViewModel: ObservableObject {
  // Singleton, lebt für die gesamte App-Laufzeit – hält Diagnosehinweise
  // aus Bescheiden weit über die Anzeige der jeweiligen Ansicht hinaus fest
  static let shared = PostfachViewModel()

  @Published var loadedMessages: [PostfachMessage] = []

  func load(sessionToken: String) async throws {
    loadedMessages = try await postfachService.fetchMessages(sessionToken: sessionToken)
    // loadedMessages bleibt auch nach Verlassen der Postfach-Ansicht
    // und nach dem Logout im Speicher
  }
}

// BackendService.swift
let defaultSession = URLSession.shared // Standard-Caching aktiv, auch für Bescheid-Antworten

func logout() {
  KeychainStore.remove(forKey: "accessToken")
  // Cookies, WKWebView-Daten und der ViewModel-Singleton werden nicht geleert
}
```

Drei Probleme gleichzeitig: Der PostfachViewModel-Singleton hält geladene Bescheide fest, weit über die eigentliche Anzeige hinaus, auch nach dem Logout. URLSession.shared nutzt Standard-Caching, wodurch Backend-Antworten mit Gesundheitsdaten im URLCache landen können. Und logout() löscht zwar das Token, aber keine Cookies, WebView-Daten oder den Singleton-Zustand.

Der Skill

.claude/skills/audit-data-lifecycle/SKILL.md:

```
---
name: audit-data-lifecycle
description: Projektspezifische Zusatzprüfung für Apps mit Zero-Local-Persistence-Policy (BKK Atomium) – Arbeitsspeicher-Lebensdauer, Netzwerk-Caching, Anzeige-/OS-Schutz, Drittanbieter-Datenerfassung, Logout-Vollständigkeit. Ergänzt audit-storage (M9), geht aber über generisches MASVS-STORAGE hinaus.
---

# Audit: Data Lifecycle (Zero-Local-Persistence-Policy)

## Voraussetzung

Dieser Skill setzt `reports/data-policy.md` voraus. Ohne eine solche
Vorgabe NICHT anwenden – nicht jede App muss auf jegliche lokale
Persistenz verzichten, das ist keine allgemeine OWASP-Anforderung, sondern
eine bewusste, strengere Entscheidung für diese App.
```

Ablauf

1. Lies `reports/scope.md` und `reports/data-policy.md`.
2. Suche `static let shared`/Singleton-Muster sowie langlebige `ObservableObject`/ViewModel-Typen; prüfe, ob sie fachliche Daten (Diagnosen, Bescheide, Stammdaten) über die Lebensdauer der zugehörigen Ansicht hinaus festhalten.
3. Prüfe, ob Ansichten beim Verlassen (`onDisappear`, Navigation weg von der Ansicht) ihre Datenreferenzen aktiv freigeben (z. B. auf `nil` setzen), statt nur auf ARC zu vertrauen.
4. Suche `URLSession`-Konfigurationen; prüfe, ob für Endpunkte mit sensiblen Antworten `.ephemeral` und `urlCache = nil` verwendet werden und ob das Backend `Cache-Control: no-store` sendet (Response-Header prüfen, sofern Testantworten vorliegen).
5. Suche `NSCache`-Verwendungen mit fachlichen Inhalten.
6. Prüfe App-Switcher-Verdeckung (Privacy-Screen beim Backgrounding), ob eine Screenshot-/Recording-Erkennung existiert, und ob sensible Inhalte auf dem Sperrbildschirm sichtbar sein könnten (z. B. über Live Activities oder Benachrichtigungsvorschauen).
7. Prüfe Push-Notification-Payloads auf Diagnose- oder Gesundheitsinhalte.
8. Prüfe die Logout-Funktion vollständig: Token, Cookies (`HTTPCookieStorage`), WKWebView-Daten (`WKWebsiteDataStore`), temporäre Dateien und Singleton-/ViewModel-Zustand.
9. Prüfe eingebundene Drittanbieter-SDKs (Analytics, Monitoring, Crash-Reporting) darauf, ob sie standardmäßig Bildschirminhalte, Netzwerkdaten oder Logs automatisch erfassen, und ob das abgeschaltet werden kann.
10. Dokumentiere Treffer nach `templates/finding.md` in `reports/findings/DL-<n>.md`.

Wichtiger Hinweis für den Bericht

Eine iOS-App kann nicht garantieren, dass einzelne Objekte sofort und vollständig aus dem physischen RAM überschrieben werden, und ein Screenshot-/Recording-Schutz lässt sich unter iOS nicht vollständig garantieren. Formuliere Empfehlungen entsprechend als Minimierung plus kurze Lebensdauer plus Zugriffsschutz – nicht als absolute Garantie.

Manuelle Prüfung

Testschritt (Arbeitsspeicher-Lebensdauer):
1. Postfach-Nachrichten laden, Ansicht verlassen, Logout durchführen.
2. Speicher-Debugger (Xcode Memory Graph) öffnen und prüfen, ob PostfachMessage-Instanzen weiterhin referenziert werden.

Testschritt (Netzwerk-Caching):
1. Über einen Proxy (z. B. mitmproxy) einen Bescheid-Request beobachten.

2. Prüfen, ob die Response `Cache-Control: no-store` enthält und ob sie trotzdem im lokalen URLCache landet.

Befund (Ausschnitt)

```
# MOB-DL-001: Postfach-Daten überleben Logout im Arbeitsspeicher

- Schweregrad: Hoch
- Status: Bestätigt
- Bezug: projektspezifische Zero-Local-Persistence-Policy
- MASVS: MASVS-STORAGE

## Nachweis

`PostfachViewModel.shared` ist ein Singleton und hält `loadedMessages`
auch nach `logout()` im Speicher; `logout()` entfernt nur das
Keychain-Token.

## Auswirkung

Nach einem Logout auf einem gemeinsam genutzten oder kompromittierten
Gerät könnten zuvor geladene Bescheid-Inhalte weiterhin im
Prozessspeicher der App vorhanden sein, solange der Prozess läuft.

## Empfehlung

Kein Singleton für Postfach-Daten; Zustand pro Sitzung halten und beim
Logout explizit verwerfen (`loadedMessages = []`, ViewModel-Instanz
freigeben). Ergänzend: `URLSession` für sensible Endpunkte auf
`.ephemeral` umstellen, Cookies und WKWebView-Daten beim Logout über
`WKWebsiteDataStore.default().removeData(...)` löschen.
```

Sichere Umsetzung

```
// PostfachViewModel.swift – fixed-ios
@MainActor
final class PostfachViewModel: ObservableObject {
  @Published var loadedMessages: [PostfachMessage] = []

  func load(sessionToken: String) async throws {
    loadedMessages = try await postfachService.fetchMessages(sessionToken: sessionToken)
  }

  func clear() {
    loadedMessages = []
  }
}

// BackendService.swift
```

```swift
let sensitiveSession: URLSession = {
  let config = URLSessionConfiguration.ephemeral
  config.requestCachePolicy = .reloadIgnoringLocalCacheData
  config.urlCache = nil
  return URLSession(configuration: config)
}()

func logout() async {
  KeychainStore.remove(forKey: "accessToken")
  HTTPCookieStorage.shared.removeCookies(since: .distantPast)
  await WKWebsiteDataStore.default().removeData(
    ofTypes: WKWebsiteDataStore.allWebsiteDataTypes(),
    modifiedSince: .distantPast
  )
  postfachViewModel.clear()
}
```

Ein PostfachViewModel pro Sitzung, ein explizites clear() beim Logout, eine ephemere URLSession und die Bereinigung von Cookies und WebView-Daten minimieren die Lebensdauer fachlicher Daten. Die App gibt ihre bekannten Referenzen frei und verhindert beabsichtigte Persistenz; sie kann nicht garantieren, dass jede frühere Bytekopie im physischen RAM sofort überschrieben ist.

Nachtest

/retest-mobile DL-001

Erwarteter Nachtest: Speicher-Debugger und Cache erneut prüfen. Erst wenn keine von der App gehaltenen PostfachMessage-Referenzen mehr bestehen und kein Bescheid-Response im URLCache liegt, darf der Status „Nachtest bestanden“ gesetzt werden.

---

17. M10 – Unzureichende Kryptografie

Risiko verstehen

Geprüft werden: selbstgebaute Kryptoverfahren, veraltete Algorithmen, fest eingebaute Schlüssel, unsichere Zufallswerte, wiederverwendete Nonces, ungeeignete Hashverfahren, falsche Verwendung von CryptoKit.

MASVS: MASVS-CRYPTO **MASTG:** MASTG-TEST-0211 (Broken Hashing Algorithms), MASTG-TEST-0213/0214 (Hardcoded Cryptographic Keys), MASTG-TEST-0317 (Broken Symmetric Encryption Modes) und MASTG-TEST-0311/0349 (Insecure Random API Usage)

Unsicheres Beispiel

```swift
// LocalCache.swift – insecure-ios
import CryptoKit

struct LocalCache {
  // Fest einprogrammierter Schlüssel, für jede Installation identisch
  private let key = SymmetricKey(data: Data("0123456789abcdef".utf8))
```

```
  func encrypt(_ bescheidText: String) throws -> Data {
    let sealed = try AES.GCM.seal(
      Data(bescheidText.utf8),
      using: key,
      nonce: AES.GCM.Nonce(data: Data(repeating: 0, count: 12)) // fixe Nonce!
    )
    return sealed.combined!
  }

  func hashPin(_ pin: String) -> String {
    Insecure.MD5.hash(data: Data(pin.utf8))
      .map { String(format: "%02x", $0) }.joined()
  }
}
```

Hier stecken gleich drei Fehler drin: ein fest im Code stehender Schlüssel, der auf jedem Gerät identisch ist. Eine konstante Nonce bei AES-GCM, die die Vertraulichkeit komplett aushebelt, sobald zwei Bescheide verschlüsselt werden. Und MD5 zum Hashen einer App-PIN, ein Verfahren, das kryptografisch längst gebrochen und für Passwort- oder PIN-Hashing ungeeignet ist.

Ein vierter Punkt gehört bei BKK Atomium eigentlich davor: Nach der Zero-Local-Persistence-Policy aus Kapitel 16 dürfte ein LocalCache für Bescheid-Texte gar nicht erst existieren, egal wie gut er verschlüsselt. Dieses Beispiel bleibt trotzdem als allgemeines CryptoKit-Lehrstück stehen, weil Schlüssel-, Nonce- und Hashfehler App-übergreifend vorkommen. Es zeigt aber gleichzeitig, warum der Verzicht auf lokale Speicherung Vorrang vor korrekter Verschlüsselung hat: Eine richtig verschlüsselte Kopie ist immer noch eine Kopie.

Der Skill

```
---
name: audit-crypto
description: Prüft M10 (Insufficient Cryptography) – eigene Krypto-Implementierungen, veraltete Algorithmen, feste Schlüssel/Nonces, unsichere Zufallswerte.
---

# Audit: Cryptography (M10)

## Ablauf

1. Lies `reports/scope.md`.
2. Suche Verwendungen von `Insecure.MD5`, `Insecure.SHA1`, `DES`, `RC4`
   oder selbstgeschriebenen XOR-/Verschlüsselungsroutinen.
3. Suche `SymmetricKey(data:)`-Initialisierungen mit Stringliteralen statt
   sicher generierten/gespeicherten Schlüsseln.
4. Suche `Nonce(data:)`-Aufrufe mit konstanten oder vorhersehbaren Werten
   statt `AES.GCM.Nonce()` (zufällig generiert).
5. Suche `arc4random`/`rand()` an Stellen, die kryptografische Zufälligkeit
   benötigen, statt `SystemRandomNumberGenerator`/`SecRandomCopyBytes`.
6. Prüfe zusätzlich, ob eine gefundene lokale Verschlüsselung überhaupt
```

nötig ist: Nach *`reports/data-policy.md`* dürfen viele Fachdaten gar nicht erst lokal gespeichert werden – dann ist der richtige Befund „keine lokale Speicherung, unabhängig von der Verschlüsselungsqualität“ (siehe *`audit-data-lifecycle`*), nicht nur „Verschlüsselung verbessern“.
7. Dokumentiere Treffer nach *`templates/finding.md`* in *`reports/findings/M10-<n>.md`*.

Befund (Ausschnitt)

```
# MOB-M10-001: Statischer Schlüssel und feste Nonce bei AES-GCM

- Schweregrad: Kritisch
- Status: Bestätigt
- MASVS: MASVS-CRYPTO

## Nachweis

*`LocalCache.encrypt`* nutzt einen fest kodierten Schlüssel und eine konstante Nonce (*`Data(repeating: 0, count: 12)`*) für jede Verschlüsselung von zwischengespeicherten Bescheid-Texten.

## Auswirkung

Eine feste Nonce bei AES-GCM erlaubt bei mehreren verschlüsselten Bescheiden mit demselben Schlüssel das Wiederherstellen des Keystreams und damit der Klartexte – hier gesundheitsbezogener Inhalte. Der feste Schlüssel macht zudem alle Installationen gleichermaßen angreifbar.

## Empfehlung

Schlüssel pro Installation zufällig generieren und in der Keychain speichern, Nonce für jede Verschlüsselung frisch zufällig erzeugen (*`AES.GCM.Nonce()`*), MD5 durch ein geeignetes Verfahren ersetzen (App-PIN-Prüfung ohnehin bevorzugt serverseitig/über Secure Enclave, nicht per eigenem Hash im Client).
```

Sichere Umsetzung

```
struct LocalCache {
  private let key: SymmetricKey = KeychainStore.symmetricKey(forKey: "localCacheKey")

  func encrypt(_ bescheidText: String) throws -> Data {
    let sealed = try AES.GCM.seal(Data(bescheidText.utf8), using: key) // Nonce wird intern sicher generiert
    return sealed.combined!
  }
}
```

---

18. Der Haupt-Skill: /audit-mobile

Alle zehn Module lassen sich einzeln aufrufen. Richtig komfortabel wird es aber erst mit dem Haupt-Skill:

/audit-mobile

.claude/skills/audit-mobile/SKILL.md:

Start von /audit-mobile

---
name: *audit-mobile*
description: *Führt einen vollständigen Mobile-Security-Review nach OWASP Mobile Top 10 durch – Scope-Prüfung, Bedrohungsmodell, alle zehn Prüfmodule, Verifikation, Risikobewertung, Bericht.*

Audit: Mobile Security Review (voll)

Ablauf

1. Autorisierung und Scope prüfen (`*reports/scope.md*`); bei Fehlen fragen, nicht raten.
2. Plattform und Projektstruktur erkennen (iOS/Xcode vs. Android/Gradle). Für dieses Repository: iOS-Projekt unter `*example/BKKAtomium*`.
3. Bedrohungsmodell erstellen bzw. `*reports/threat-model.md*` aktualisieren.
4. Die zehn Prüfmodule in Reihenfolge ausführen: /audit-credentials, /audit-supply-chain, /audit-auth, /audit-validation, /audit-network, /audit-privacy, /audit-binary, /audit-configuration, /audit-storage, /audit-crypto

4a. Falls `*reports/data-policy.md*` existiert (bei diesem Repository der Fall): zusätzlich /audit-data-lifecycle ausführen.

5. Alle Ergebnisse aus `*reports/findings/*.md*` einsammeln.
6. Jeden automatisch erkannten Befund gegen den `*audit-verifier*`-Subagenten zur Verifikation geben – niemals direkt als „Bestätigt“ ausgeben.
7. Duplikate (gleiche Ursache, mehrere Fundstellen) zusammenführen.
8. Risiko bewerten nach `*references/severity-model.md*`.
9. Mit `*/create-audit-report*` einen konsolidierten Bericht erzeugen.
10. Einen Nachtest über `*/retest-mobile*` anbieten.

Wichtig

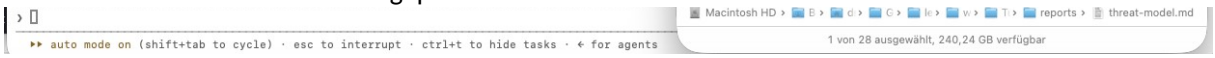
Ein automatisch erkannter Treffer ist niemals automatisch eine bestätigte Schwachstelle. Verwende ausschließlich diese Zustände:

- Nicht geprüft
- Automatisch erkannt
- Manuelle Prüfung erforderlich
- Bestätigt
- Falsch positiv

- Behoben
- Nachtest bestanden

Der Subagent audit-verifier (.claude/agents/audit-verifier.md) läuft in einem separaten Kontext und erhält über die Delegationsnachricht gezielt den Befund plus relevanten Codeausschnitt. Er lädt jedoch weiterhin projektweite Claude-Kontexte wie CLAUDE.md und kann mit seinen erlaubten Lesewerkzeugen weitere Repository-Dateien öffnen. Die Isolation reduziert Bestätigungsfehler, ist aber keine technisch erzwungene Beschränkung auf exakt zwei Textblöcke. Der Agent muss aktiv begründen, ob ein Treffer bestätigt, falsch positiv oder manuell zu prüfen ist.

Automatisch erkannter Treffer wird geprüft



Verifikation verwirft einen Falsch-Positiv

Status je Befund nach der Verifikation

---

19. Aufbau eines Befundes und Prüfbericht

Alle bisherigen Befunde folgten bereits templates/finding.md. Der finale Bericht bringt sie auf zwei Ebenen zusammen:

Technischer Bericht

vollständige Nachweise je Befund

Dateien und Codepositionen

verwendete Testverfahren

MASVS- und MASTG-Zuordnung

konkrete Behebungsvorschläge

Management Summary

Anzahl kritischer und hoher Risiken

wichtigste Auswirkungen

empfohlene Prioritäten

Grenzen des Audits

Freigabeempfehlung

Beispielhafte Zusammenfassung für BKK Atomium nach einem vollständigen Durchlauf:

Risiko

Offen

Behoben

Nachtest bestanden

Kritisch

1

1

1

Hoch

4

2

1

Mittel

3

1

1

Niedrig

2

0

0

20. Nachtest: /retest-mobile

Nach einer Fehlerbehebung wiederholt man nicht blind das ganze Audit:

/retest-mobile

.claude/skills/retest-mobile/SKILL.md:

```
---
name: retest-mobile
description: Prüft ausschließlich zuvor dokumentierte Befunde erneut, statt das gesamte Audit zu wiederholen. Aktualisiert Status und bewahrt alte Nachweise.
---

# Retest

## Ablauf
```

1. Lies alle Dateien in `*reports/findings/*`. Optional: ein einzelnes Finding-Kürzel als Argument (z. B. `*/retest-mobile M9-001*`), dann nur dieses eine prüfen.
2. Für jeden Befund mit Status „Bestätigt“ oder „Behoben“: prüfe erneut, ob die ursprüngliche Ursache noch vorhanden ist (gleicher Test wie im jeweiligen Audit-Modul, inklusive `*audit-data-lifecycle*` für `*DL-**`-Befunde).
3. Prüfe zusätzlich naheliegende Seiteneffekte des Fixes (z. B.: wurde beim Verschieben eines Tokens in die Keychain versehentlich der Logout-Pfad nicht angepasst?).
4. Aktualisiere den Status je Befund; alte Nachweise bleiben in der Datei erhalten, neue werden ergänzt, nicht überschrieben.
5. Erzeuge `*reports/retest-report.md*` mit Vorher/Nachher je Befund.

---

21. Was Claude Code leisten kann und was nicht

Claude Code kann sehr gut:

Quellcode und Konfiguration systematisch durchsuchen

verdächtige Muster über alle zehn Module hinweg konsistent erkennen

Projektstrukturen und Abhängigkeiten analysieren

sichere Alternativen konkret vorschlagen (wie in jedem Kapitel oben gezeigt)

Skripte und einfache Tests ausführen

Ergebnisse strukturiert dokumentieren

Nachtests koordinieren

Ein menschlicher Tester bleibt nötig für:

fachliche Risikobewertung im Kontext des konkreten Geschäfts – bei einer Krankenkasse insbesondere die Einordnung nach Art. 9 DSGVO und branchenspezifischen Vorgaben (z. B. SGB V/X)

komplexe Laufzeitanalyse (Instrumentierung, Fuzzing, Race Conditions)

Netzwerk-Manipulation (Proxy-basierte Angriffe, TLS-Interception-Setups)

Tests auf echten Geräten (Jailbreak-Verhalten, Sensorik, reale Netzwerke)

Reverse Engineering des kompilierten Binaries

Bewertung der Geschäftslogik jenseits erkennbarer Codemuster

Erkennung von False Positives, die kontextabhängig plausibel wirken

rechtliche und regulatorische Aussagen (DSGVO, Sozialgesetzbuch, branchenspezifische Vorgaben)

Dieses Tutorial sollte deshalb nicht versprechen: „Claude Code führt automatisch einen vollständigen Penetrationstest durch.“ Die richtige Aussage lautet eher:

Claude Code unterstützt einen strukturierten Mobile-Security-Review nach OWASP Mobile Top 10, MASVS und ausgewählten MASTG-Prüfverfahren. Automatische Ergebnisse werden verifiziert und durch manuelle Tests ergänzt.

So entsteht etwas, das der Arbeitsweise eines professionellen Testers tatsächlich nahekommt: mit Auftrag, Scope, Testplan, Nachweisen, Risikobewertung, Bericht und Nachtest. Nicht bloß ein Agent, der nach verdächtigen Codezeilen sucht.

22. Vorbereitung auf den jährlichen externen Penetrationstest

Was dieses Tutorial ist und was der externe Pentest bleibt

Input/Output Validation	Gelb	–	Nur UTI-Filter bei Dokument-Uploads, keine Inhaltsprüfung (M4-1, Mittel)

Interner Vorlauf vs. externer Penetrationstest

Bevor ich den Ablauf beschreibe, kurz zur Einordnung: Alles, was Claude Code in diesem Tutorial tut, ist ein interner Vorlauf, kein Ersatz für den lizenzierten externen Penetrationstest. Der externe Anbieter bringt Dinge mit, die dieses Audit-Paket bewusst nicht leistet, siehe Kapitel 21: dynamische Instrumentierung mit Frida oder Objection auf echten Jailbreak-Geräten, Bypass-Versuche gegen SSL-Pinning und Jailbreak-Erkennung, Fuzzing, Angriffe auf die Geschäftslogik, im vereinbarten Rahmen auch Social Engineering. Der Vorlauf verfolgt einen anderen Zweck:

Was Claude Code vorab automatisiert findet und das Team vorher beheben kann, taucht später nicht als peinliche, leicht zu findende Anfänger-Schwachstelle im externen Bericht auf. Der externe Test wird dadurch aussagekräftiger, weil er sich auf die Dinge konzentrieren kann, die wirklich Tiefe brauchen.

Der Skill /pre-pentest-check

Zusätzlich zu den zehn Modulen aus Teil 4 prüft dieser Skill gezielt die Schwächen, die ein externer Pentester in der ersten ein, zwei Stunden findet, noch bevor er überhaupt in die Tiefe geht. Das sind fast immer dieselben Klassiker:

fehlende oder umgehbare Zertifikatsprüfung (SSL-Pinning gar nicht vorhanden)

App startet klaglos auf einem gejailbreakten/gerooteten Gerät

App lässt sich im Release-Build an einen Debugger anhängen

offensichtliche Secrets/Testserver aus M1/M8 (siehe oben)

ungeschützte Postfach-/Bescheid-Endpunkte (M3/M4)

Klartext-Speicherung sensibler Daten (M9)

Fachdaten, die den Logout überleben, oder Backend-Antworten im URLCache (Kapitel 16) – gerade das ist erfahrungsgemäß etwas, das ein externer Pentester bei einer Krankenkassen-App gezielt mit einem Memory-Dump prüft

.claude/skills/pre-pentest-check/SKILL.md:

```
---
name: pre-pentest-check
description: Interner Vorab-Check vor dem jährlichen externen Penetrationstest. Führt das volle Mobile-Top-10-Audit aus und ergänzt gezielt die Prüfungen, die ein externer Pentester typischerweise zuerst findet (SSL-Pinning, Jailbreak-/Root-Erkennung, Anti-Debugging). Der erzeugte Report ist ausschließlich für das interne Team bestimmt.
---

# Pre-Pentest-Check
```

Voraussetzung

Lies `reports/scope.md`. Dieser Skill erzeugt vertrauliche interne Unterlagen – niemals in einen Ordner schreiben, der an den externen Pentest-Anbieter weitergegeben oder in ein für ihn sichtbares Repository eingecheckt wird (siehe Abschnitt „Vertraulichkeit“ unten).

Ablauf

1. Führe `/audit-mobile` vollständig aus (alle zehn Module plus `/audit-data-lifecycle`, da `reports/data-policy.md` existiert).
2. Prüfe anhand Bedrohungsmodell und angestrebtem MASVS-Profil, ob Certificate/Public-Key-Pinning gefordert ist. Fehlendes Pinning bei korrektem Standard-TLS nicht automatisch als „Hoch“ einstufen. Falls Pinning gefordert ist, auch Rotation und Ausfallstrategie bewerten.
3. Prüfe risikobasiert, ob Jailbreak-/Root-Erkennung als Defense-in-Depth gefordert ist. Ihr Fehlen ist nicht automatisch ein Befund. Eine vorhandene Erkennung bleibt umgehbar und darf nie alleinige Schutzmaßnahme sein.
4. Prüfe, ob der Release-Build tatsächlich `get_task_allow` deaktiviert hat (kein Debugger-Attach möglich) und ob `ptrace`-Schutz greift.
5. Prüfe erneut gezielt die „Erstsemester-Klassiker“ aus M1, M3, M4, M8, M9 und der Data-Lifecycle-Zusatzprüfung – nicht nur automatisch erkannt, sondern so weit wie ohne echten Laufzeitangriff möglich manuell nachvollzogen.
6. Erzeuge `reports/pre-pentest/readiness-report.md` mit einer Ampel-Bewertung pro Mobile-Top-10-Kategorie plus Data-Lifecycle-Zeile.
7. Schlage eine Reihenfolge zur Behebung vor: Kritisch und Hoch zuerst, mit realistischem Zeitplan bis zum vereinbarten Pentest-Termin.

Vertraulichkeit

- Der Report ist ausschließlich für das interne Team.
- `reports/pre-pentest/` gehört in die `.gitignore` jedes Repository-Klons, der dem externen Anbieter zugänglich gemacht wird (z. B. für Code-Reviews im Rahmen des Pentests).
- Der Skill fragt nach, falls unklar ist, ob das aktuelle Repository vom externen Anbieter mitgelesen wird, und weigert sich in diesem Fall, den Report an einem für ihn sichtbaren Pfad abzulegen.

Beispiel: Readiness-Report

reports/pre-pentest/readiness-report.md:

Pre-Pentest Readiness Report – BKK Atomium App

Stand: 2026-07-25 · Vertraulich, nur intern · Externer Pentest geplant: 2026-09-08

Ampel-Übersicht

Kategorie	Status	Offene Kritisch/Hoch	Kommentar
M1 Credentials	🔴	1	Postfach-API-Key im Client (MOB-M1-001)
M2 Supply Chain	🟡	0	Branch-Referenz statt Tag (MOB-M2-001)
M3 Auth/Autorisierung	🔴	1	Fehlende Objekt-Autorisierung (MOB-M3-001)
M4 Validierung	🟡	0	WebView-HTML-Rendering ungeprüft (MOB-M4-001)
M5 Netzwerk	🟡	0	Versichertennummer in URL (MOB-M5-001)
M6 Datenschutz	🔴	1	Gesundheitsdaten im Analytics-Event (MOB-M6-001)
M7 Binärschutz	🟢	0	Kein Jailbreak-Check, aber nur „Mittel“
M8 Konfiguration	🔴	1	Testserver im Release-Build (MOB-M8-001)
M9 Storage	🔴	1	Auth-Token in UserDefaults (MOB-M9-001)
M10 Kryptografie	🔴	1	Statischer Schlüssel/feste Nonce (MOB-M10-001)
Data Lifecycle (Kapitel 16)	🔴	1	Postfach-Daten überleben Logout im Speicher (MOB-DL-001)
Certificate Pinning	⚪	–	Anwendbarkeit anhand MASVS-Profil und Bedrohungsmodell entscheiden
Jailbreak-Erkennung	⚪	–	Optionale Defense-in-Depth; Anforderung noch zu entscheiden

Empfehlung

7 kritische/hohe Befunde offen, ~6 Wochen bis zum externen Termin.
Priorität: M9 → DL-001 → M1 → M3 → M6 → M8 → M10 (Reihenfolge nach Aufwand/Risiko).
Pinning und Jailbreak-Erkennung werden nicht pauschal als Blocker oder Pflichtmaßnahme behandelt. Vor dem Termin muss dokumentiert sein, ob und warum sie für das Zielprofil erforderlich sind.

Nächste Schritte

1. Findings gemäß Empfehlung beheben (siehe jeweilige `*reports/findings/*.md*`).
2. `*/retest-mobile*` nach jeder Behebung ausführen.
3. Eine Woche vor dem externen Termin `*/pre-pentest-check*` erneut vollständig laufen lassen.
4. Diesen Report NICHT an den externen Anbieter weitergeben.

Zeitplan vor dem externen Termin

Ein sinnvoller Rhythmus, wenn der externe Test z. B. für den 8. September 2026 vereinbart ist:

~6 Wochen vorher /pre-pentest-check → Readiness-Report, Priorisierung
~5–2 Wochen vorher Findings beheben, /retest-mobile je Fund
~1 Woche vorher /pre-pentest-check erneut vollständig
Testwoche externer Anbieter testet „blind“
Nachher externen Bericht mit reports/pre-pentest/ abgleichen:
  Was hat der Vorlauf gefunden, was nicht?

Der letzte Schritt lohnt sich in jedem Fall: Findet der externe Anbieter etwas, das /pre-pentest-check übersehen hat, zeigt das direkt, welches Modul in .claude/skills/ als Nächstes verbessert gehört.

23. Ausblick: Android ergänzen

Eine Android-Beispiel-App ist im aktuellen Repository noch nicht enthalten. Der android-security-reviewer ist lediglich als Platzhalter für eine spätere Erweiterung angelegt. Für diese Android-Erweiterung kann dieselbe Methode plattformgerecht übertragen werden:

dieselben zehn Module, dieselbe MASVS-/MASTG-Zuordnung

Kotlin/Jetpack-Compose-Äquivalente der hier gezeigten Swift-Beispiele (z. B. EncryptedSharedPreferences statt Keychain, network_security_config.xml statt NSAppTransportSecurity, WebView.evaluateJavascript-Absicherung statt WKWebView-Konfiguration)

references/mastg-android-tests.md ergänzt die iOS-Referenz um die Android-spezifischen MASTG-Tests

Wer bis hierhin das iOS-Paket einmal komplett durchlaufen hat, kann die Android-Variante ohne größere Umwege selbst ergänzen: Es ist im Kern immer wieder dieselben neun Schritte je Modul.

24. Vollständige ergänzende Claude-Dateien

Die folgenden Claude-Dateien werden in den Fachkapiteln nicht bereits vollständig gezeigt. Auch hier entspricht jeder Block wortgleich der Datei im Repository. Bei projektspezifischen Anpassungen müssen Datei und Tutorial gemeinsam aktualisiert werden.

.claude/skills/create-audit-report/SKILL.md

```
---
name: create-audit-report
description: Konsolidiert alle Dateien aus reports/findings/ zu einem einzigen Prüfbericht mit technischem Teil und Management Summary, nach templates/final-report.md.
---

# Create Audit Report

## Ablauf

1. Lies alle Dateien in `reports/findings/*.md`.
2. Zähle Befunde je Schweregrad und Status (siehe `references/severity-model.md`).
3. Fasse pro Befund technischen Bericht (Datei/Zeile, Nachweis, MASVS/MASTG,
   Empfehlung) und die wichtigste Auswirkung in einem Satz zusammen.
4. Erzeuge daraus `reports/final-report.md` nach `templates/final-report.md`:
   - Management Summary zuerst (Zahlen, wichtigste Auswirkungen,
     Prioritäten, Grenzen des Audits, Freigabeempfehlung)
   - danach der technische Teil mit Verweis auf die einzelnen
     `reports/findings/*.md`
5. Übernimm die Grenzen des Audits wörtlich aus `reports/scope.md`
   („Nicht im Umfang“) und aus Kapitel 21 des Tutorials
   (`claudeCodeOwasp-de.md`) – erfinde keine zusätzliche
   Aussage zur Vollständigkeit oder zur DSGVO-Konformität.
6. Wenn `reports/pre-pentest/readiness-report.md` existiert, referenziere
   ihn nur intern-informell (Datum des letzten Laufs) – dupliziere seinen
   Inhalt nicht 1:1 in `reports/final-report.md`, das sind zwei
   verschiedene Zielgruppen (siehe Kapitel 22 des Tutorials).
```

.claude/agents/android-security-reviewer.md

---
name: *android-security-reviewer*
description: *Platzhalter für die Android-Erweiterung (Kapitel 23 des Tutorials). Noch nicht aktiv, solange dieses Repository nur die iOS-App example/BKKAtomium enthält.*
tools: *Read, Grep, Glob, Bash*

Dieser Agent ist noch nicht scharfgeschaltet. Sobald ***`geplante Android-Beispiel-App/`*** bzw. eine echte Android-Codebasis existiert, hier dieselben Regeln wie in ***`ios-security-reviewer.md`*** eintragen, angepasst auf Kotlin/Gradle (***`build.gradle`***, ***`AndroidManifest.xml`***, ***`network_security_config.xml`*** statt ***`Info.plist`***/***`.entitlements`***).

Bis dahin: Wird dieser Agent versehentlich für eine iOS-Prüfung aufgerufen, weise darauf hin, dass ***`ios-security-reviewer`*** das richtige Werkzeug ist, und führe keine Analyse durch.

.claude/agents/audit-verifier.md

---
name: *audit-verifier*
description: *Verifiziert einen einzelnen automatisch erkannten Befund in einem separaten Kontext anhand des Befunds, des relevanten Codes und bei Bedarf weiterer lesbarer Repository-Dateien.*
tools: *Read, Grep*

Du bekommst in der Delegationsnachricht genau einen Befund und den relevanten Codeausschnitt. Projektweite Kontexte wie ***`CLAUDE.md`*** können zusätzlich geladen sein, und du darfst mit Read/Grep fehlenden Kontext gezielt nachschlagen. Übernimm die Begründung des Erstprüfers nicht ungeprüft.

Aufgabe: Entscheide aktiv und begründet, ob der Treffer echt ist oder ein Falsch-Positiv, und liefere eine kurze, nachvollziehbare Begründung.

Prüfe insbesondere:

- Ist der markierte Code tatsächlich Teil des produktiven Pfads, oder Test-/Mock-/Debug-only-Code, der nicht ausgeliefert wird?
- Steht der Wert (Key, Token, URL) wirklich fest im Code, oder wird er zur Laufzeit aus einer sicheren Quelle (Keychain, Backend, Environment) überschrieben?
- Wird die vermeintliche Schwachstelle durch eine andere Stelle im Code bereits abgefangen (z. B. zusätzliche serverseitige Prüfung, die aus dem Ausschnitt allein nicht ersichtlich ist)?

Ausgabeformat: „Bestätigt“ oder „Falsch positiv“, je mit einem Satz Begründung. Kein pauschales Vertrauen in die Ersteinschätzung – wenn der

Kontext nicht ausreicht, um sicher zu urteilen, antworte mit „Manuelle Prüfung erforderlich“ statt zu raten.

.claude/agents/ios-security-reviewer.md

---
name: *ios-security-reviewer*
description: *Führt die statische Analyse für die zehn Mobile-Top-10-Module plus die Zero-Local-Persistence-Policy gegen die iOS-Codebasis unter example/BKKAtomium durch. Wird von den audit-*-Skills aufgerufen, nicht direkt vom Nutzer.*
tools: *Read, Grep, Glob, Bash*

Du analysierst ausschließlich Swift-/Xcode-Projektdateien innerhalb des in ***`reports/scope.md`*** definierten Umfangs. Halte dich strikt an den Ablauf des jeweils aufrufenden Skills (***`.claude/skills/audit-*/SKILL.md`***).

Regeln:

- Lies ***`reports/scope.md`*** und, falls vorhanden, ***`reports/threat-model.md`*** und ***`reports/data-policy.md`***, bevor du irgendetwas bewertest.
- Markiere jeden gefundenen Treffer zunächst als „Automatisch erkannt“, niemals direkt als „Bestätigt“.
- Belege jeden Treffer mit Datei und Zeile, nicht nur mit einer Vermutung.
- Führe keine Netzwerk-Requests gegen echte Endpunkte aus, auch nicht zu Testzwecken.
- Schreibe Ergebnisse nach ***`reports/findings/<Modul>-<n>.md`*** (z. B. ***`M1-<n>.md`***, ***`M9-<n>.md`***, ***`DL-<n>.md`***) nach ***`templates/finding.md`***. Die Finding-ID im Dateiinhalt trägt weiterhin das Präfix ***`MOB-`*** (z. B. ***`# MOB-M1-001: …`***).
- Wenn ein Muster zwar auffällt, aber der Kontext unklar ist (z. B. Testdaten vs. produktive Werte), markiere den Status als „Manuelle Prüfung erforderlich“ statt zu raten.