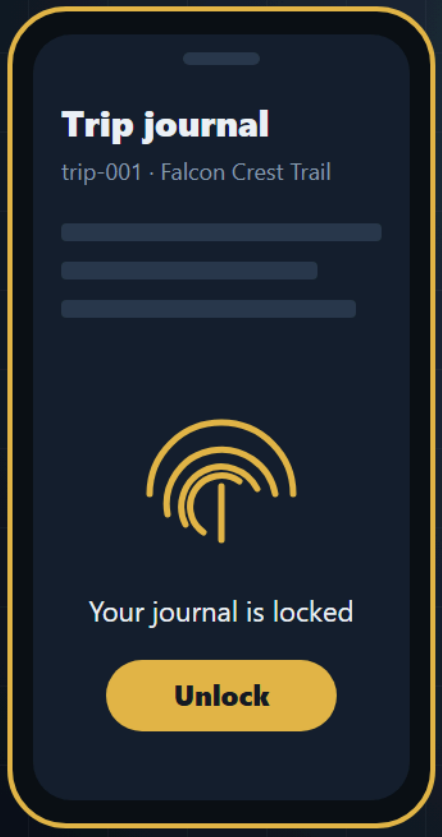
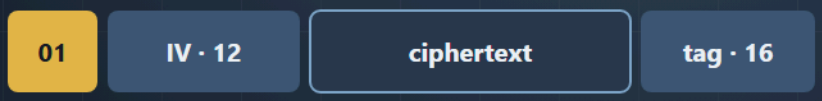



```
KeyGenParameterSpec.Builder(
  "waypoint_journal_key",
  PURPOSE_ENCRYPT or PURPOSE_DECRYPT)
.setBlockModes(BLOCK_MODE_GCM)
.setKeySize(256)
.setUserAuthenticationRequired(true)
```




```
<base-config cleartextTrafficPermitted=
  "false">
<domain>api.waypoint.example</domain>
<pin-set expiration="2027-06-30">
  primary + backup pins
```



A PRACTICAL, DEFENSIVE GUIDE

Android Security and Privacy

Harden a real Kotlin app, from threat model to security review

Permissions · Safe Components · Deep Links · WebView · Keystore
AES-GCM · Biometrics · Passkeys · TLS and Pinning · Play Integrity
Logging Hygiene · Data Safety · CI Secrets · OWASP MASVS

THREATS · STORAGE · IDENTITY · NETWORK · PRIVACY

THENEXBOOKS

Android Security and Privacy

Permissions, Secure Storage, Keystore and Biometrics, Network Security, Safe Components, and Privacy by Design

Harden a real Kotlin app step by step, from threat model to security review

First Edition

THENEXBOOKS

Android Security and Privacy
Permissions, Secure Storage, Keystore and Biometrics, Network Security, Safe Components, and Privacy by Design

First Edition, 2026

Copyright © 2026 thenexbooks (thenexbooks.gumroad.com). All rights reserved.

No part of this publication may be reproduced, distributed, or transmitted in any form or by any means without the prior written permission of the publisher, except for brief quotations in reviews and certain other non-commercial uses permitted by copyright law. Code examples may be used freely in your own projects.

The information in this book is provided "as is" without warranty of any kind. Security guidance reduces risk; it cannot guarantee that an application is free of vulnerabilities. While every precaution has been taken in its preparation, the publisher assumes no responsibility for errors or omissions, or for damages resulting from the use of the information contained herein. Android, its libraries, the Android Gradle Plugin, and Google Play policies change frequently; APIs, default behaviors, verdict labels, and policy requirements can differ between versions. Always consult the official documentation and the current Google Play policies for the versions you target, and have security-critical designs reviewed by qualified people. The content of this book is defensive: use it only to protect applications you own or are authorized to test.

Android, Google Play, Google Play Console, Play Integrity, Jetpack Compose, and Android Studio are trademarks of Google LLC. Kotlin is a trademark of the Kotlin Foundation. OWASP, MASVS, and MASTG are names of projects of the OWASP Foundation. Other product names are trademarks of their respective owners. Their use in this book is for identification only and does not imply endorsement. This book is independent and is not affiliated with, sponsored by, or endorsed by Google LLC, JetBrains, or the OWASP Foundation.

Northbeam Labs, the Waypoint app, its team members, users, trails, trips, journal entries, server addresses, certificate pins, and tokens used as the running example in this book are fictional. Domains use reserved example names (such as `waypoint.example`), and all keys, pins, and credentials shown are placeholders that protect nothing.

Preface

A phone is the most personal computer most people own. It knows where they walked this morning, holds their private notes and photos, stays signed in to their accounts, and travels with them into cafés, airports, and taxis where it can be lost in a moment. When you ship an Android app, you are asking people to trust your code with that device. Security and privacy are not features you add at the end; they are part of what it means for an app to work.

Android gives you strong building blocks: an application sandbox, a permission system that users understand, hardware-backed key storage, biometric authentication, a declarative network security configuration, and integrity signals from Google Play. But none of them protect an app that is assembled carelessly. An exported activity that forwards intents, a token written to a log line, a WebView with a JavaScript bridge, a backup rule that copies the wrong database, or a "temporary" trust-all certificate manager can undo all of it.

This book teaches Android security and privacy the way a working team needs them: as concrete engineering decisions, each tied to a threat, implemented in Kotlin, and verified with tests and tooling. We take one realistic app, write down what could go wrong, and harden it chapter by chapter, explaining *why* each defense works, what it costs, and where its limits are.

Who This Book Is For

- **Android developers** who build apps with Kotlin, Jetpack Compose, Room, and Retrofit and want to ship them safely.
- **Tech leads and architects** preparing a security review, a penetration test, or a Google Play policy submission.
- **Developers handling sensitive data** such as location, health-adjacent notes, photos, or account credentials.
- **QA and security-minded engineers** who want to turn security requirements into automated checks.

We assume you can already build a Compose app with a ViewModel, a Room database, and a Retrofit client. We do not re-teach those basics; we focus on their security and privacy angles.

What You Will Learn

- How to build a threat model for a mobile app and use it to decide where to spend effort.
- How the Android security model works: the sandbox, UIDs, SELinux, signing, and Verified Boot, and what each one does and does not protect.
- How to request permissions respectfully and minimally, including location, the photo picker, and notifications.

- How to design safe components: exported activities, intents and PendingIntents, deep links and App Links, FileProvider, and WebView.
- How to protect data at rest: storage choices, backup and device-transfer rules, AES-GCM done right, the Android Keystore, and an encrypted journal with biometric unlock.
- How to handle identity and the network: token storage and refresh, Credential Manager and passkeys, network security configuration, TLS, and certificate pinning trade-offs.
- How to reason about hardening: R8 and its limits, tamper and root signals, and the Play Integrity API with server-side decisions.
- How to practice privacy by design: data minimization, retention, logging hygiene, the Data safety form, account deletion, CI secrets, and an OWASP MASVS-style review.

How This Book Is Organized

Part I — Foundations (Chapters 1–5) builds the threat model, tours the starter project, explains the platform security model, and covers permissions. **Part II — Secure App Components** (Chapters 6–10) covers the manifest as an attack surface, intents and PendingIntents, deep links and App Links, content providers and FileProvider, and WebView. **Part III — Protecting Data on the Device** (Chapters 11–15) covers storage and backups, practical cryptography, the Android Keystore, the encrypted journal, and biometric unlock. **Part IV — Identity and the Network** (Chapters 16–20) covers tokens, passkeys, network security configuration, certificate pinning, and API hardening. **Part V — Hardening and App Integrity** (Chapters 21–24) covers R8, tamper and root signals, Play Integrity, and logging hygiene. **Part VI — Privacy and Security Engineering** (Chapters 25–28) covers privacy by design, Google Play Data safety, security in CI, and a complete security review mapped to OWASP MASVS. The appendices provide a cheat sheet, templates and reference tables, a production security checklist, and a glossary.

Each chapter ends with exercises and a **Key Takeaways** box for quick review.

The Running Example: Waypoint

Throughout the book we harden **Waypoint**, a hiking trip planner and tracker built by Northbeam Labs, a small fictional product studio. Waypoint lets hikers browse trails, plan trips, record hikes with location samples, and keep a private journal of notes and photos for each trip; it signs users in to a backend and offers an optional subscription. That mix is exactly what makes security interesting: precise location history, private notes, account tokens, a paid entitlement, links that open the app, and a server that must not trust the client blindly. We start from a small, tested starter project and add defenses one threat at a time: an encrypted journal that only opens after biometric authentication, a validated deep-link parser, an authorization interceptor that never leaks tokens, a pinned network configuration, backup rules, redacted logs, and privacy decisions you can defend in a Play Console form. The same app appears in other thenexbooks Android titles, but this book is complete on its own.

Versions and Conventions

Examples use Kotlin and Jetpack Compose with a modern toolchain at the time of writing: Android Gradle Plugin 9.4, Gradle 9.8, Kotlin 2.4, compileSdk 37, targetSdk 36, and minSdk 26, with `androidx.biometric` 1.1.0. Android's security behavior changes from release to release, so we say which Android version introduced a behavior when it matters, and we write "recent versions" where details are still moving. Library versions that you must add yourself are not pinned in the text; take the current version from the official release notes. Code that can run on the JVM was compiled and tested with unit tests and Robolectric; code that needs a real device, such as the Android Keystore, BiometricPrompt, Credential Manager, and the Play Integrity API, is clearly marked, and its output is described as what you should see "something like".

> **TIP**
> A recommended practice or a technique that makes your work easier.

> **NOTE**
> Background information, version differences, or a useful aside.

> **WARNING**
> A common mistake or something that can cause data loss, a security weakness, or a privacy problem.

Kotlin, XML, and shell commands appear in `monospace`; long lines are wrapped to fit the page, and file captions show paths relative to the project root, such as `app/src/main/AndroidManifest.xml`. Commands are shown in the Windows form (`.\gradlew.bat test`); on macOS and Linux use `./gradlew test`. Every key, password, token, and certificate pin in this book is a placeholder for a fictional service; never copy them into a real app.

Contents

Preface

PART I — FOUNDATIONS

1 Why Mobile Security Is Different: Assets, Attackers, and the Waypoint Threat Model

2 The Waypoint Security Starter: Project Tour, Tests, and Lint

3 The Android Security Model: Sandbox, UIDs, SELinux, and Verified Boot

4 Permissions in Depth: Install-Time, Runtime, and Special Access

5 Sensitive Permissions Done Right: Location, Photos, and Notifications

PART II — SECURE APP COMPONENTS

6 Exported Components and the Manifest as Attack Surface

7 Intents and PendingIntents: Explicit Intents, Redirection, and Mutability Flags

8 Deep Links and Android App Links: Verification and Input Validation

9 Content Providers and FileProvider: Sharing Files Safely

10 WebView Safety: JavaScript, Bridges, File Access, and Custom Tabs

PART III — PROTECTING DATA ON THE DEVICE

11 Storage and Backups: Internal Storage, Scoped Storage, and Data Extraction Rules

12 Cryptography for App Developers: AES-GCM, Nonces, and Authenticated Data

13 The Android Keystore: Hardware-Backed Keys, StrongBox, and Attestation

14 Encrypting the Waypoint Journal: Envelopes, Room, Key Loss, and Rotation

15 Biometric Unlock: BiometricPrompt, Authenticators, and CryptoObject

PART IV — IDENTITY AND THE NETWORK

16 Authentication Tokens: Login, Refresh, Secure Storage, and Logout

17 Credential Manager and Passkeys

18 Network Security Configuration: TLS, Cleartext, and Trust Anchors

19 Certificate Pinning: Trade-offs, Backup Pins, and Rotation

20 Hardening API Calls: Interceptors, Token Scope, and Server-Side Trust

PART V — HARDENING AND APP INTEGRITY

21 R8, Obfuscation, and Secrets in the APK: What Shrinking Does and Does Not Protect

22 Tamper, Root, and Emulator Signals: What They Tell You and Their Limits

23 The Play Integrity API: Requests, Verdicts, and Server-Side Decisions

24 Logging Hygiene and Other Leaks: Logcat, Crash Reports, Screenshots, and the Clipboard

PART VI — PRIVACY AND SECURITY ENGINEERING

25 Privacy by Design: Data Minimization, Location, Retention, and the Privacy Dashboard

26 Google Play Data Safety, Privacy Policies, and Account Deletion

27 Security in CI and Testing: Lint, Dependencies, Secrets, and Security Tests

28 The Waypoint Security Review: OWASP MASVS and MASTG Mapping

APPENDICES

A Android Security Cheat Sheet

B Templates and Reference

C Production Security Checklist

D Glossary

PART I

Foundations

Why mobile security is different, a threat model for Waypoint, a tested starter project, the platform's own security model, and permissions requested minimally and in context: location, photos, and notifications.

1 · Assets, Attackers, and the Threat Model
2 · The Waypoint Security Starter
3 · The Android Security Model
4 · Permissions in Depth
5 · Location, Photos, and Notifications

CHAPTER 1 · PART I — FOUNDATIONS

Why Mobile Security Is Different: Assets, Attackers, and the Waypoint Threat Model

Before we write a single line of security code, we need to know what we are protecting, from whom, and where the boundaries are. In this chapter we look at what makes a phone app different from a web or server application, learn a small and practical threat-modeling vocabulary, and build the threat model for Waypoint that the rest of the book uses as its map.

A Different Kind of Battlefield

If you have written backend services, you are used to a comfortable assumption: the code runs on machines you control, the binaries never leave the data center, and an attacker has to come through the public API to reach anything. A mobile app turns this picture inside out. The code runs on a device that belongs to someone else. It can be lost on a trail, shared with a curious sibling, filled with apps of uncertain origin, or deliberately modified by its owner. You cannot log in to that device, patch it on your schedule, or even be sure which Android version it runs. Five differences shape almost every decision in this book.

1. The device is in hands you do not control

The most common "attacker" against a phone is not a nation-state; it is someone who holds the device for a few minutes while it is unlocked. Anything your app shows without asking again is visible to that person, and anything stored on the device can be copied by someone who controls it completely, such as the owner of a rooted device. Device encryption helps when the phone is powered off or locked, but it does nothing for a running, unlocked app: at that moment the operating system decrypts every file for whoever is holding it.

2. Your app binary is public

Every APK you publish can be downloaded, unpacked, and decompiled. R8 renames classes and removes unused code, but strings, network endpoints, resources, and the overall logic remain recoverable with standard tools. Anything you put in the APK, you have published: API keys in `BuildConfig`, "secret" URLs, and client-side checks that decide what a user may do are all readable and changeable. The principle starts here and returns in Chapter 21: the server, not the app, makes trust decisions.

3. The operating system mediates everything

On the positive side, Android sandboxes apps heavily. Each app runs as its own Linux user, cannot read another app's private files, and can reach other apps only through channels the system controls: intents, content providers, bound services, and broadcasts. Much of good Android security is simply *not undoing* what the platform gives you: not exporting a component by accident, not writing private data to shared storage, not turning off certificate validation to make a test pass. Chapter 3 explains this model in detail.

4. Every device and OS version is slightly different

Waypoint supports Android 8.0 (API 26) and newer. Over that range, permissions changed shape several times, storage rules were rewritten, backup gained a second configuration format, and new restrictions arrived for components and notifications. Hardware varies too: some devices have a dedicated secure element for keys, others only a trusted execution environment. Security code on Android is full of version checks and graceful fallbacks, and the tests have to reflect that.

5. Stores and policies are part of the system

On Google Play, policies become part of your requirements. Play asks you to declare what data you collect and share, restricts some sensitive permissions to apps whose core function needs them, requires an account deletion path for apps with accounts, and offers services such as the Play Integrity API and Play App Signing. Policies change over time, so we point you to the current documentation rather than quoting rules that may have moved.

> **NOTE**
>
> None of this makes mobile apps hopelessly insecure. It means the shape of the problem is different: less "keep intruders out of the building" and more "assume the building is on the street, and decide carefully what to keep inside it". The good news is that the platform does a lot of the heavy lifting, as long as we cooperate with it.

Meet Waypoint and the Northbeam Labs Team

Throughout this book we harden **Waypoint**, a hiking trip planner and tracker built by Northbeam Labs, a small fictional product studio. The application id is `com.northbeam.waypoint`. Users browse trails, plan trips, record hikes with location samples, and keep a private journal with notes and photos for each trip. An optional subscription, **Waypoint Plus**, unlocks offline trail packs and unlimited trips. The backend lives at the fictional `https://api.waypoint.example/v1/`; we never talk to a real server, and every test in the book runs against a local mock.

Four people appear in the examples. **Lena Ortiz** leads Android development and owns most of the code we change. **Sam Okafor** runs the backend and is the person we negotiate with whenever a decision belongs on the server. **Priya Natarajan** is responsible for product and design, which makes

her the voice of the user in every trade-off between friction and protection. **Theo Brandt** leads QA and turns each security decision into a test he can run on every build.

The story begins with a request: a large hiking club wants to recommend Waypoint to its members if it passes an independent security review. Lena does not want to start by adding random "security features". She asks three questions first: what in this app is worth protecting, who might want it, and how would they get it? Those questions are the heart of threat modeling.

A Small Vocabulary for Threat Modeling

Threat modeling has a reputation for heavy process and long documents. For an app team it can be a two-hour whiteboard session that produces one table. We need only a handful of words, used consistently.

- **Asset**: something with value that we must protect. Data (journal text, tokens, location history), capabilities (calling the API as a user), and money (the Plus entitlement) are all assets.
- **Attacker** (or threat actor): a person or program that might try to get or misuse an asset. We describe attackers by what they can do, not by who they are.
- **Threat**: a specific way an attacker could harm an asset. "A malicious app on the same device calls one of our exported components to read the journal" is a threat; "hackers" is not.
- **Vulnerability**: a weakness that makes a threat possible, such as a component exported by accident or a token written to Logcat.
- **Control** (or mitigation): what we do to prevent, detect, or limit a threat. Encryption, permission checks, and server-side validation are controls.
- **Trust boundary**: a line where data or control passes between parties that trust each other differently. Every time data crosses a trust boundary, we must validate it on the receiving side.
- **Risk**: our judgment of how likely a threat is and how bad it would be. Risk is what we use to decide what to fix first and what to accept.

Two habits make these words useful: name the asset, attacker, and path in every threat, and connect every control to a test, or it will quietly disappear in a future refactor.

Waypoint's Assets

The team lists everything in Waypoint that someone could want, then groups the items until six remain. Each asset gets a short id so we can refer to it from code reviews, tickets, and the final review in Chapter 28.

ID	Asset	Why it matters
A1	Journal text and photos	Personal, sometimes about health or companions; written to be private.
A2	Auth tokens	Let anyone who holds them act as the user against the API.

A3	Location history (TrackPoints)	Reveals home, routines, and where someone is alone.
A4	Account identity (name, email)	Personal data; useful for phishing and account takeover.
A5	Waypoint Plus entitlement	Revenue; abuse costs money and bandwidth.
A6	Backend API and app integrity	A tampered client or scripted abuse harms every user.

Notice what is *not* on the list. The trail catalog and the app's code are public, so we do not spend effort hiding them. Leaving things off matters, because every control costs development time, user friction, or both.

A3 is an asset even though Waypoint collects it for a good reason: location is harmless in a single point and revealing in aggregate, since thirty days of hikes starting from the same street say where someone lives.

Who Are the Attackers?

We describe attackers by capability. Five profiles cover almost everything Waypoint faces, and you will meet the same five in most consumer apps.

1. **Someone holding a lost or stolen phone.** Low technical skill, physical access, limited time. If the phone is unlocked, they see whatever the app shows. If it is locked, they depend on the device's own protections.
2. **A malicious app on the same device.** Installed by the user, perhaps disguised as a flashlight or a game. It cannot read our private files directly, but it can send intents to our exported components, register for the same deep links, read anything we put on shared storage or the clipboard, and draw over our screens if the user grants it that special access.
3. **A network attacker.** Sitting on the same public Wi-Fi, running a rogue access point, or controlling a network path. They can observe and modify traffic. With a user-installed certificate authority on the device (for example on a managed or tricked device) they may even be able to read TLS traffic from apps that trust such certificates.
4. **A user with a rooted device or a repackaged app.** The device owner, or someone who modified our APK and redistributed it. They can read every file, change code, hook functions at runtime, and replay or forge API requests. On their own device, they ultimately win any contest that happens purely on the client.
5. **A compromised build pipeline or third-party SDK.** A leaked signing key, a poisoned dependency, a CI secret printed into a log, or an analytics SDK that collects more than we expect. This attacker does not touch the user's phone at all; they reach every user at once through our own release.

Asking what each profile can reach is more useful than ranking them by "scariness". The rooted-device user has unlimited access to their own copy of the app, which matters for A5 and A6 but not for other people's journals. The pipeline attacker is rare but catastrophic, because one mistake reaches every

install.

> **WARNING**
>
> Do not design against a mythical attacker who can do anything. If you assume that, every control looks pointless and the team gives up. Design against the realistic profiles above, state clearly which ones a control addresses, and write down the residual risk you accept. "Biometric unlock protects the journal from someone who picks up an unlocked phone; it does not protect it from the owner of a rooted device" is a perfectly good engineering statement.

Data Flows and Trust Boundaries

With assets and attackers in hand, the team sketches how data moves. A threat model does not need a formal diagram tool; a box-and-arrow sketch that fits on one screen is enough, as long as it shows where data crosses from one level of trust to another.

Waypoint data flow (simplified)



Four trust boundaries stand out:

- **B1, other apps to Waypoint.** Anything that arrives through an intent, a deep link, a shared file, or the clipboard comes from code we do not control. We validate it as hostile input. Chapters 6 to 10 live here.

- **B2, Waypoint to the Keystore.** The Android Keystore holds keys outside our process, in a trusted execution environment or a secure element when the hardware has one. Our code asks the Keystore to encrypt or decrypt; it never sees the key bytes. Chapters 13 to 15 live here.
- **B3, device to backend.** Every byte crosses networks we do not trust. TLS protects it in transit, and the server must treat every request as potentially forged, because a modified client can send anything. Chapters 16 to 20 and 23 live here.
- **B4, build pipeline to every device.** What we sign and ship becomes trusted code on millions of devices, or on a few hundred in Waypoint's case. Protecting signing keys and dependencies is a security control even though it never runs on a phone. Chapter 27 lives here.

There is also a boundary that is easy to forget: between the app and the user's own attention. Notifications on the lock screen, the Recents thumbnail, and screenshots shared in a support ticket all move data out of the app through the user's screen. Chapter 24 deals with those leaks.

STRIDE in Plain Words

To find threats systematically, many teams walk each part of the diagram through STRIDE, a classic checklist of six threat categories. We use it as a prompt, not as a ritual. For every box and arrow, ask the six questions in the table below and write down anything that sounds plausible.

Letter	Question	Waypoint example	Typical control
S, Spoofing	Can someone pretend to be someone else?	A stolen refresh token lets an attacker act as Alex.	Token storage, short lifetimes, revocation
T, Tampering	Can someone change data or code?	A journal ciphertext is swapped between rows in the database.	Authenticated encryption (AES-GCM with AAD)
R, Repudiation	Can someone deny doing something?	A user disputes a Plus purchase they made.	Server-side records and audit logs
I, Information disclosure	Can data leak to the wrong party?	A token appears in Logcat or a crash report.	Redaction, logging hygiene
D, Denial of service	Can someone make it stop working?	A script floods the trip upload endpoint.	Rate limiting on the server
E, Elevation of privilege	Can someone gain abilities they should not have?	Another app uses an exported activity to open the journal.	Export review, permission checks

Some categories land mostly on the server: repudiation and denial of service are real for Waypoint, but their controls belong to Sam's backend. Being honest about this split stops the team from solving a server problem with client code that an attacker can simply remove.

The Waypoint Threat Model

After a whiteboard session with STRIDE and the data-flow sketch, the team keeps ten threats. Each has an id, a one-line description, and the main defense with the chapters where we build it. This table is the map for the whole book; whenever a chapter adds a control, it names the threat it addresses.

ID	Threat	Main defense (chapter)
T1	Unlocked stolen phone shows the journal	Biometric unlock + auth-bound key (13–15)
T2	Backup or device transfer copies the journal	Backup rules + Keystore keys (11, 14)
T3	Malicious app calls an exported component	Export review, explicit intents, PendingIntent flags (6–7)
T4	Crafted link opens a deep link with bad input	App Links + `DeepLinkParser` (8)
T5	WebView runs attacker JavaScript with a bridge	WebView hardening, Custom Tabs (10)
T6	Network attacker intercepts API traffic	TLS, network security config, pinning (18–19)
T7	Token leaks to logs or third-party hosts	Token store, `AuthInterceptor`, `Redactor` (16, 20, 24)
T8	Repackaged app or rooted device abuses Plus/API	R8, integrity signals, Play Integrity, server checks (21–23)
T9	Location history is over-collected or exposed	Coarse-first permissions, retention, Data safety (5, 25–26)
T10	Signing keys or secrets leak from repo/CI	Secret management, CI hardening (27)

Let us walk through the reasoning behind each entry, because the reasoning is more valuable than the table.

T1 and T2: the journal at rest

T1 covers the most likely attacker of all: someone holding Alex's phone while it is unlocked. Device encryption does not help, so the journal needs its own gate. A biometric prompt alone is a UI gate that a bug could skip; binding the decryption key to user authentication in the Keystore makes the key itself unusable until the user proves presence. T2 covers copies made by cloud backup or device transfer. Waypoint excludes the journal from backups and keeps its key in the Keystore, which never backs up keys, so even a copied database would be unreadable ciphertext.

T3, T4, and T5: other apps and hostile input

These three live on boundary B1. An exported component is an entry point any app can call (T3). A deep link can be triggered by any page or app, and custom schemes such as `waypoint://` can be claimed by any app (T4). A WebView that runs JavaScript with a Kotlin bridge gives every page it

loads access to that bridge (T5). The defenses share a pattern: expose as little as possible, verify the source when the platform can, and validate every field as hostile.

T6 and T7: tokens and traffic

Auth tokens (A2) act as the user's password for the API while they are valid. T6 covers the network path: the platform's TLS already protects Waypoint well, and the network security configuration stops us from weakening it by accident. Pinning adds protection against mis-issued or user-installed certificates at an operational cost we discuss in Chapter 19. T7 covers the quieter leaks: a token in a debug log, attached to a request for a third-party image host, or stored in plain preferences. Most token leaks look like T7, not like a dramatic interception.

T8, T9, and T10: integrity, privacy, and the pipeline

T8 accepts a hard truth: on a device the attacker controls, the client eventually does what the attacker wants. R8 raises the cost of reading the code, and integrity signals give the server evidence, but the decision to serve a Plus offline trail pack lives on the server. The app never bans anyone on its own, because local checks produce false positives and can be removed by exactly the people they target. T9 is caused not by an outsider but by us collecting too much or keeping it too long; data you never collect cannot leak. T10 targets boundary B4: a signing key in a repository, a password in a CI log, or an unreviewed dependency update can compromise every install at once.

Rating Risk Without Pretending to Be Precise

Some threat-modeling methods produce numeric scores with many decimal places. For an app team, a simple qualitative rating is more honest: for each threat, estimate likelihood (low, medium, high) and impact (low, medium, high), and discuss the result. The numbers are not data; they are a structured way to argue about priorities.

Threat card template (one per threat, kept next to the code)

```
ID:          T1
Threat:      Unlocked stolen phone shows the journal
Asset(s):    A1 journal text and photos
Attacker:    someone holding a lost/stolen phone (unlocked)
Boundary:    user attention / device
Likelihood:  medium   (phones are lost; many are unlocked at the time)
Impact:      high     (private notes, sometimes health details)
Controls:    biometric unlock (Ch 15), auth-bound journal key (Ch 13),
             FLAG_SECURE on the journal screen (Ch 24)
Tests:       JournalViewModelTest.staysLocked_andReadsNothing_untilUnlocked
             JournalScreenTest.lockedState_showsNoEntryText_andUnlockCallsBack
Residual:    a rooted device owned by the attacker; shoulder surfing
             while the user reads the journal
Owner:       Lena Ortiz      Reviewed: (date of last review)
```

The "Tests" line is the most important one on the card. It turns a paragraph in a document into something Theo can run on every build. In the next chapter you will see that the starter project already contains tests that support several cards: the journal stays locked and reads nothing until unlocked, ciphertext copied to another row is hidden, only one component is exported, and the network configuration forbids cleartext.

> **TIP**
>
> Keep the threat cards in the repository, for example in a `docs/security/` folder, rather than in a slide deck. Then a pull request that changes the manifest, the network configuration, or the token code can update the matching card in the same review, and the threat model stays as current as the code.

Security Is a Set of Trade-offs

Every control has a cost, and the team has to pay it in some currency: development time, user friction, battery, support tickets, or operational risk. A threat model makes those costs visible so the team can decide on purpose. Here are four decisions Northbeam Labs makes in this book, with the trade-off spelled out.

- **How often to ask for biometrics.** Asking on every journal open protects best against T1 but annoys a hiker who wants to jot down a note with cold fingers. Waypoint unlocks a 30-second window after a successful authentication: short enough that a phone set down on a table locks itself quickly, long enough to read and add a note. Priya pushed for usability, Lena for protection, and the 30 seconds is their written compromise.
- **Whether to pin certificates.** Pinning defends T6 against mis-issued or user-installed certificates but can lock every installed app out of the API if the server's key changes unexpectedly. Waypoint pins with a backup key and an expiration date, and Chapter 19 shows the rotation runbook that makes it safe. Many teams reasonably choose not to pin at all; we explain both sides.
- **What to do on a rooted device.** Blocking every rooted device would stop some abuse of T8 and would also lock out enthusiasts who simply like custom ROMs. Waypoint reports signals to the server and lets the server restrict only the one feature that costs real money (Plus offline packs), and even then it flags the account for review instead of silently banning it.
- **How much location to keep.** Keeping every track point forever would enable nice "your year in hiking" features, and it would also turn the database into a detailed record of where every user goes. Waypoint keeps raw points for 30 days after a trip and then keeps only the summary.

None of these choices is "the secure one" in the abstract. Each is a reasonable answer for Waypoint, written down with its reasoning so a future team member can change it consciously instead of accidentally. Honesty about limits belongs in the same place: we never call a control "unhackable"; we say what it does and does not do.

Our Defensive Stance

This is a book about protecting apps, not attacking them. We explain an attack only as far as you need to understand why a defense works. We do not provide exploit code, instructions for attacking other people's apps, or walkthroughs of root, hooking, or bypass tooling. Testing your *own* app is encouraged throughout: lint with security checks promoted to errors, unit tests for parsers and encryption, merged-manifest inspection, a local proxy in front of your own debug build, and a look inside your own release APK. Running these checks first turns a stressful review into a confirmation.

> **NOTE**
>
> If you are responsible for a security review or a penetration-test report, the structure of this book maps naturally onto the OWASP Mobile Application Security Verification Standard (MASVS) and its testing guide (MASTG). Chapter 28 connects every Waypoint control to those categories; you do not need to know MASVS beforehand.

How This Book Is Organized Around the Threat Table

Each part of the book attacks a group of threats from the table. The parts build on each other, but every chapter names the threat it addresses, so you can also jump straight to the one your review report or Play policy email is about.

Part	Focus	Threats
I. Foundations (Ch 1–5)	Threat model, the starter project, the platform security model, permissions	All; T9 (Ch 5)
II. Secure App Components (Ch 6–10)	Exported components, intents, deep links, providers, WebView	T3, T4, T5
III. Protecting Data on the Device (Ch 11–15)	Storage, backups, AES-GCM, Keystore, the encrypted journal, biometrics	T1, T2
IV. Identity and the Network (Ch 16–20)	Tokens, passkeys, TLS configuration, pinning, API calls	T6, T7
V. Hardening and App Integrity (Ch 21–24)	R8, integrity signals, Play Integrity, logging and other leaks	T7, T8
VI. Privacy and Security Engineering (Ch 25–28)	Privacy by design, Data safety, CI, the final review	T9, T10, all

The running code lives in the **Waypoint security starter**, which builds and tests entirely on your computer without an emulator. Device-only features such as the Keystore, the biometric prompt, and the Play Integrity API are always labeled as such, with output described as what you should see something like.

Keeping the Threat Model Alive

A threat model written once and filed away is a historical document. The team agrees on four triggers for revisiting Waypoint's model: a **new asset** (photo attachments, sharing, payments), a **new boundary** (an exported component, an SDK, a backend host, a WebView), a **platform or policy change** that alters what the OS does or what the store requires, and an **incident or review finding**, which is evidence that a card or a test is missing.

Theo adds one habit: any pull request that touches the manifest, the network configuration, the security module, or token handling gets a review comment naming the threat id it affects. That keeps the ids alive and makes the final review in Chapter 28 much faster.

Try It Yourself

1. Take an app you work on and list its assets. Aim for five to eight, then remove anything that is already public. Give each one an id like A1, A2, and so on.
2. Draw a data-flow sketch for that app in plain text, like the Waypoint diagram above. Mark every trust boundary and give it a short name.
3. Walk one boundary through the six STRIDE questions. Write each plausible threat as a sentence that names an asset, an attacker, and a path.
4. Write one threat card using the template in this chapter. Fill in the "Tests" line with a test that exists today, or write "none" and open a ticket to add one.
5. Pick one trade-off in your app, such as how often users must re-authenticate, and write down the current choice and the reasoning in two or three sentences. Share it with whoever owns product decisions and see whether they agree.