

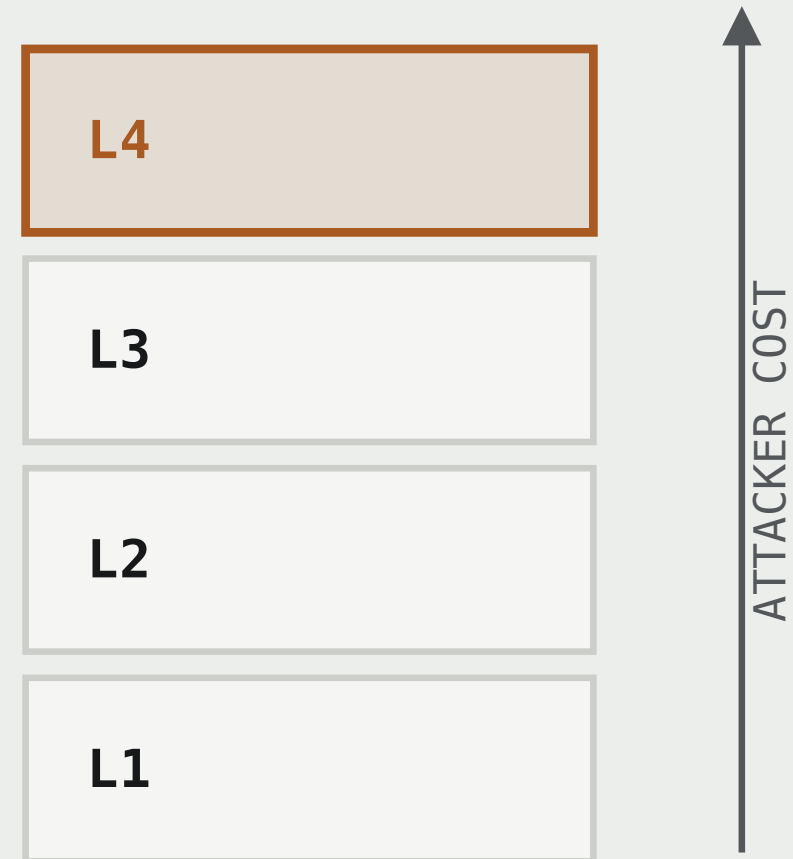
Securing Android SDKs: A Defense-in-Depth

A layered strategy for hardening an Android SDK against reverse engineering, replay, and bot-driven fraud.

35 MIN

4 LAYERS

1 THREAT MODEL



Glossary

APK	Android application package, the installable file containing compiled code and resources.
TLS	Transport Layer Security, encrypts data in transit between client and server.
Decompiler	Tools like apktool or jadx that reconstruct readable source from compiled bytecode.
ProGuard / R8	Android's built-in code shrinker and obfuscator. Renames symbols, does not encrypt strings.
DexGuard	Guardsquare's commercial obfuscator, a paid alternative that does encrypt strings.
JNI / NDK	Java Native Interface / Native Development Kit. Lets Android code call into native C/C++.
HMAC-SHA256	A keyed hash used to sign a request so tampering or forgery can be detected.
Nonce	A value used once per request, the mechanism that makes a captured request fail on replay.
TTL	Time to live. How long a credential stays valid before it must be renewed.
Attestation	Cryptographic proof, from an API like Play Integrity, that a request comes from a genuine app on a genuine device.
Click spamming	A fraud technique that fires fabricated ad clicks to falsely claim credit for real installs.



Enrique López Mañas

Android Engineer

Editor of Kotlin Weekly

kotlin.social/@eenriquelopez

An SDK Is Trust You Ship to Devices You Don't Control

A backend service runs on your infrastructure, behind your firewall, patched on your schedule. An SDK runs on someone else's device, inside someone else's app, past a review process you don't run: the moment it ships, a piece of your trust boundary lives on hardware you'll never see.

YOUR BACKEND

- Runs on infrastructure you control.
- Patched on your schedule.
- Behind your firewall, out of public view.

YOUR SDK

- Runs on hardware you'll never see.
- Patched only when the host app updates.
- Shipped in the open, inside a public binary.

The Stakes: Revenue, Trust, and Data Integrity

Attribution and analytics SDKs sit directly in the revenue path: every install, purchase, and ad impression they report can trigger a real payout. An attacker doesn't need to breach your backend. They just need to make it believe something happened that didn't.

REVENUE

Fraudulent payouts

Forged installs and in-app events get paid out as if they were real, dollar for dollar.

DATA

Poisoned decisions

Every downstream model, dashboard, and ad-spend call inherits whatever the SDK reported, lies included.

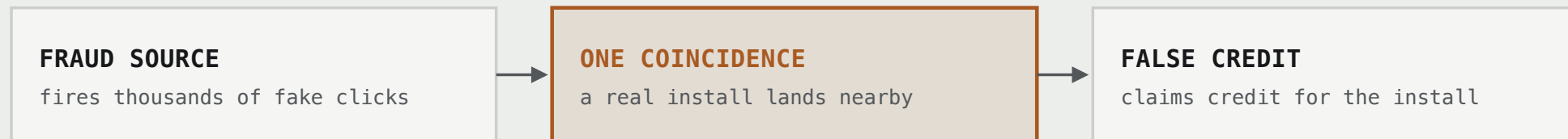
TRUST

Eroded confidence

Publishers who catch fabricated numbers stop trusting the platform that reported them.

The Report: Installs From Clicks That Never Happened

A handful of publishers flagged something odd: installs attributed to ad clicks where the click itself left no trace on the device, no ad rendered, no tap recorded, sometimes a timestamp that landed after the install had already started. The pattern had a name: click spamming, a fraud source firing a flood of fabricated click postbacks and claiming credit for whichever real installs happened to land nearby.



Why the Existing Checks Didn't Catch It

The click arrived over valid HTTPS, from a request that looked exactly like every other click. That proved the bytes weren't tampered with in transit. It proved nothing about whether an ad had actually been shown, or a person had actually tapped it.

THE CLICK REQUEST PROVED

- A valid TLS handshake.
- A well-formed, correctly structured payload.
- A device ID that looked legitimate.

THE CLICK REQUEST DID NOT PROVE

- An ad was actually rendered.
- A person actually tapped it.
- The request came from our SDK at all.

How We Fix It: Four Layers, Not One Patch

Blocking the source IP doesn't work, the next one just rotates. Rate-limiting alone doesn't work, a slow enough flood looks like normal traffic. The gap isn't a bad actor, it's structural: nothing in the request proves the click was genuine. Closing it takes more than one fix.

Over the next 35 minutes, we'll build the four layers that do, in the order we'd apply them to a real SDK.

Agenda

01	Threat model: what a decompiler actually exposes	5 min
02	Layer 1 · Obfuscation, plus native code & DexGuard	7 min
03	Layer 2 · Request signing (HMAC-SHA256)	7 min
04	Layer 3 · Device attestation, plus non-Google fallbacks	7 min
05	Layer 4 · Short-lived session tokens	6 min
06	Composing the layers, roadmap & resources	3 min

A Note on the Examples

Every code sample in this talk is written against a generic analytics SDK, a stand-in for the pattern, not a specific product. The mechanisms transfer directly: obfuscation, signing, attestation, and short-lived tokens work the same way regardless of what your SDK actually does.

Adjust the field names, the endpoints, and the specific integrity checks that matter, to your own use case.

Your SDK Ships Inside Someone Else's APK

The moment your SDK is distributed, it stops being your source code and becomes a public artifact. **apktool** or **jadx** turns a compiled `.dex`` back into readable, near-source Java in seconds: no special access required, just a copy of the app.

● JADX -D OUT/ MYAPP-RELEASE.APK

```
public final class AnalyticsConfig {  
    public static final String API_KEY =  
        "live_8f2a91cd_prod_key";  
    public static final String ENDPOINT =  
        "https://api.example.com/v1/event";  
}  
  
// four seconds after unzip. no key, no root,  
// no debugger, just a decompiler.
```

Three Ways Attackers Monetize a Readable SDK

T1

API key extraction

Pull the key straight out of the decompiled constants pool.

Call your API directly, at someone else's expense.

T2

Event injection

Reverse the wire format, then POST SDK-shaped events with no SDK involved.

Attribution and analytics get poisoned with fiction.

T3

Bot-driven fraud

Script the same forged events across a farm of emulators.

Fictional installs turn into real payouts at scale.

Encryption Protects the Pipe, Not the Speaker

TLS and code review are necessary. Neither one answers the question that actually matters once your SDK is public.

TLS PROVES

- The bytes weren't read in transit.
- The bytes weren't altered in transit.
- You're talking to the right host.

TLS DOES NOT PROVE

- The caller is your SDK.
- The SDK is running inside a real app.
- The request hasn't been replayed.
- The caller isn't a script.

Defense in Depth, Literally in Layers

LAYER 4 · SESSION TOKENS

Short-lived and attested, raising the cost of a credential lasting.

LAYER 3 · DEVICE ATTESTATION

Play Integrity raises the cost of faking the device.

LAYER 2 · REQUEST SIGNING

HMAC-SHA256 + timestamp + nonce, raising the cost of forging.

LAYER 1 · OBFUSCATION

Renames symbols and hides constants, raising the cost of reading.

↑
ATTACKER COST →

Raise the Cost of Reading

Obfuscation doesn't make reverse engineering impossible, it makes it slow. Class and method names collapse to noise, debug metadata disappears, and string constants stop sitting in the clear.

It's the cheapest layer to add and the weakest one alone. Nothing here is *authentication*, it's friction.

Before and After, Side by Side

● BEFORE · SOURCE

```
class AnalyticsClient {  
    fun trackPurchase(sku: String, price: Double) {  
        val payload = buildEventPayload(sku, price)  
        httpClient.post(ATTRIBUTION_ENDPOINT, payload)  
    }  
}
```

● AFTER · DECOMPILED RELEASE BUILD

```
public final class a {  
    public final void a(String p0, double p1) {  
        Object v0 = a.a(p0, p1);  
        a.a.a(a.b, v0);  
    }  
}
```

● PROGUARD-RULES.PRO

```
# flatten and rename the package tree
-repackageclasses 'a'
-allowaccessmodification
-obfuscationdictionary dict/keywords.txt
-keepattributes !SourceFile,!LineNumberTable
-keep class com.example.android.AnalyticsSDK {
    public *;
}
```

● ENDPOINT.KT

```
// never a plain string constant
private val ENC = byteArrayOf(0x1a, 0x0e, 0x07, 0x1c)

fun endpoint(): String =
    String(xor(ENC, runtimeKey()))
```

ProGuard/R8 Doesn't Obfuscate Strings

Renaming classes and methods says nothing about string literals. ProGuard's own FAQ is direct about this: string encryption isn't something it does. Whatever text sits in your source, an API key, an endpoint, a token, survives verbatim in the compiled output.

```
● APKTOOL D MYAPP-RELEASE.APK
```

```
$ grep -r "const-string" smali/

const-string v1, "67a5af7f89ah3katf7m20fdj202"
// classes and methods got renamed.
// the string itself never moved.
```

Native Code Raises the Bar

Move the secret into a native library instead of a Java string, and it stops showing up in a symbol or string search entirely. Through the NDK, a C++ function can hold the value and hand it back on demand, no string ever sits in a `.java` or `.kt` file.

● NATIVE-LIB.CPP + SECRETS.KT

```
// native-lib.cpp
extern "C" JNIEXPORT jstring JNICALL
Java_com_example_android_Secrets_getApiKey(JNIEnv *env, jobject) {
    return env->NewStringUTF(decrypt(ENCRYPTED_KEY, RUNTIME_SALT));
}

// Secrets.kt
external fun getApiKey(): String
```

STILL A LAYER, NOT A LOCK

Native binaries decompile too. Tools like Hex-Rays or IDA Pro read disassembled `.so` files just as readily as jadx reads Java. This removes most casual attackers, not all of them.

DexGuard, or No Secret to Steal at All

Two paths worth knowing. DexGuard, Guardsquare's commercial obfuscator, actually encrypts string literals, something R8 explicitly doesn't do, at a real licensing cost. The more elegant fix skips storage entirely: client and server know the same function, and the client derives a value at request time instead of reading one from memory. Nothing sits on disk waiting to be extracted, because nothing is stored.

WORTH REMEMBERING

"Absolute security does not exist. Security is a set of measures, piled up and combined, to slow down the inevitable."

→ WE'LL BUILD EXACTLY THAT IDEA, A VALUE DERIVED INSTEAD OF STORED, AT SESSION TOKENS

Obfuscation Delays. It Doesn't Authenticate.



THIS LAYER BUYS

Time. Casual static analysis stops finding anything useful.



THIS LAYER CANNOT STOP

A patient attacker with Frida hooking live memory, or a request being replayed once its shape is known.

→ NEXT: MAKE THE REQUEST ITSELF PROVE IT WASN'T TAMPERED WITH

Renaming Isn't Hiding

GOTCHA

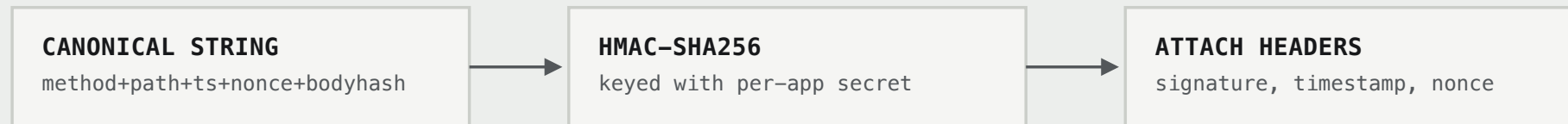
A class that implements a public SDK interface, `onEvent`, `initialize`, keeps those method names intact, because R8 won't obfuscate anything reachable from a public API surface without deliberate `-keep` exceptions. The parts of your code an integrator has to call are exactly the parts an attacker can read too.

The Threat: Replay & Fabrication

Once an attacker has seen one real request, from a network capture or reversed code, nothing stops them replaying it verbatim, or handcrafting new payloads that look identical, straight from a script with no app and no SDK involved at all.

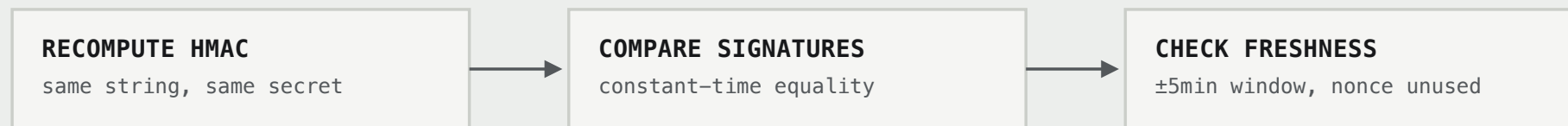
HMAC-SHA256, Timestamp, Nonce

CLIENT (SDK)



POST /v1/event · signed

SERVER



✓ **ACCEPT · 200**

× **REJECT · 401**

What Each Piece Actually Stops

DROP THIS	AND THIS HAPPENS
Timestamp	A captured request replays successfully forever. No expiry window at all.
Nonce	A request replays any number of times within the timestamp window.
Body hash	The signature no longer covers the payload. An attacker swaps in a different body and keeps a valid signature on the headers.
Secret key	There's no signature at all. Any client can construct a request that looks identical to a genuine one.

Signing a Request on Android

● REQUESTSIGNER.KT

```
fun signRequest(method: String, path: String, body: ByteArray, secret: ByteArray): Headers {
    val timestamp = Instant.now().epochSecond.toString()
    val nonce = UUID.randomUUID().toString()
    val bodyHash = sha256(body).toHex()

    val canonical = listOf(method, path, timestamp, nonce, bodyHash)
        .joinToString("\n")

    val mac = Mac.getInstance("HmacSHA256")
    mac.init(SecretKeySpec(secret, "HmacSHA256"))
    val signature = mac.doFinal(canonical.toByteArray()).toHex()

    return headersOf(
        "X-Timestamp" to timestamp,
        "X-Nonce" to nonce,
        "X-Signature" to signature
    )
}
```

The Key Still Needs a Home



THIS LAYER BUYS

Immunity to replay and to naive fabrication: a request without the secret can't produce a valid signature.



THIS LAYER CANNOT STOP

The secret itself being extracted from the binary. We've hardened how a request is authenticated, not what is making it.

→ NEXT: PROVE THE REQUEST COMES FROM A GENUINE DEVICE

The Window Is a Tradeoff

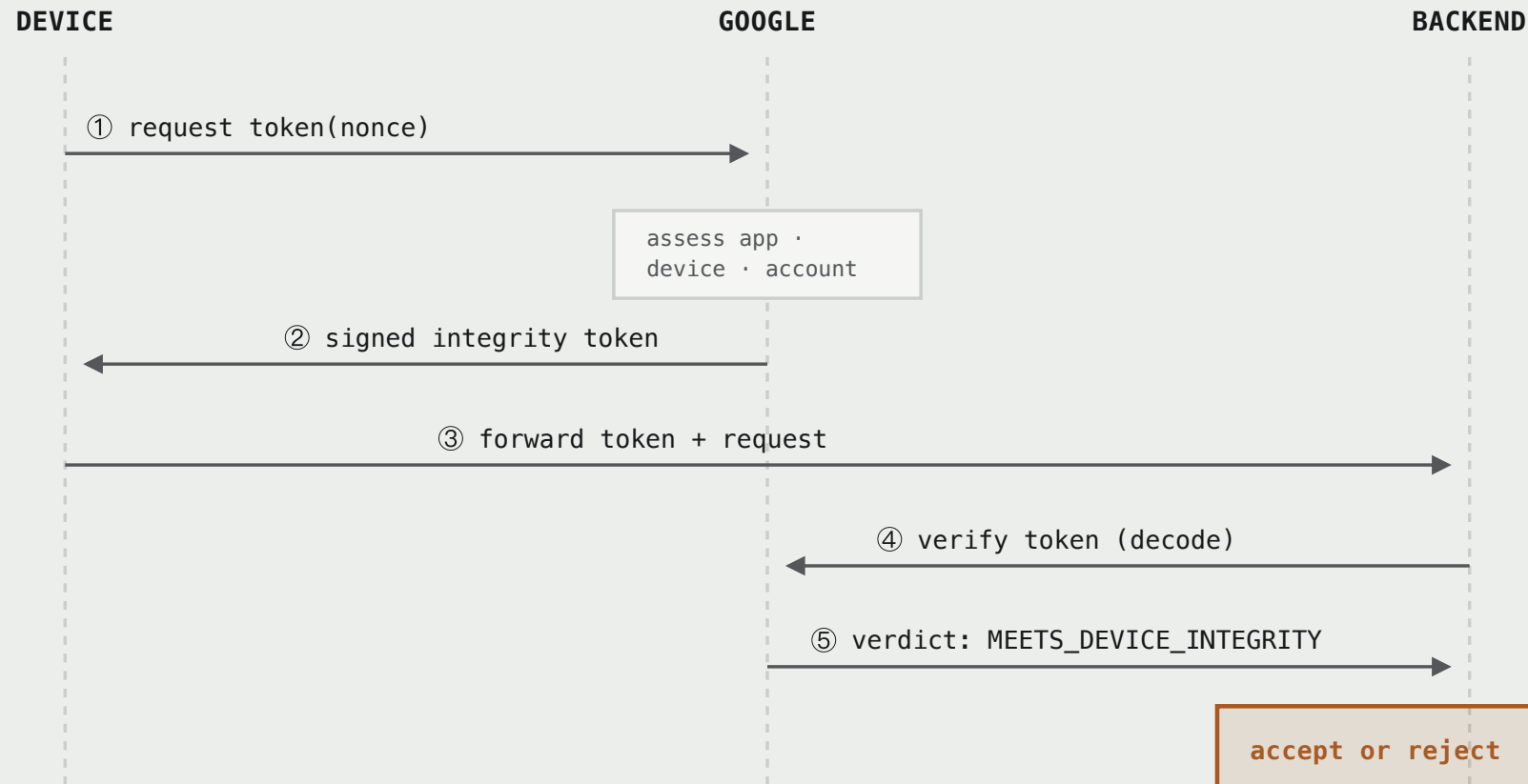
GOTCHA

A ± 5 minute timestamp window isn't a fixed rule, it's a dial. Too wide and you've reopened a replay window; too narrow and ordinary clock drift on real devices starts producing false rejections. Tune it against your own telemetry, not a default someone else picked.

Cryptographic Proof of a Genuine App + Device

The Play Integrity API lets a device ask Google to vouch, cryptographically, for three things at once: the app binary matches what you published, the device passes Google's integrity checks, and the Play account behind it is in good standing.

The Attestation Round-Trip



Reading the Verdict

The response isn't a single yes or no, and it isn't three independent judgments either. It's several fields that cascade, and getting a field name wrong means the check silently does nothing.

APP INTEGRITY

Does this match Play?

`appIntegrity.appRecognitionVerdict:`
PLAY_RECOGNIZED,
UNRECOGNIZED_VERSION, or
UNEVALUATED.

DEVICE INTEGRITY

Is this device trustworthy?

`deviceRecognitionVerdict` is an array, not a tier. A strong device returns MEETS_BASIC_INTEGRITY, MEETS_DEVICE_INTEGRITY, and MEETS_STRONG_INTEGRITY together. Check with `contains`, not `equals`.

LICENSING

Did this install come from Play?

`accountDetails.appLicensingVerdict:`
LICENSED, UNLICENSED, or UNEVALUATED.
An entitlement check, not a statement about account trust.

THEY CASCADE

If the device isn't trustworthy enough, app integrity and licensing both come back UNEVALUATED instead of a real answer, treat that as no answer, not as unlicensed. Two more fields worth knowing: `environmentDetails` covers Play Protect state, and `requestDetails` validates the nonce, check that one first. A reasonable production baseline: PLAY_RECOGNIZED, LICENSED, and a device list containing MEETS_DEVICE_INTEGRITY, only after the nonce validates.

Requesting an Integrity Token

● ATTESTATION.KT

```
val integrityManager = IntegrityManagerFactory.create(context)
val nonce = generateRequestNonce()

integrityManager.requestIntegrityToken(
    IntegrityTokenRequest.builder()
        .setNonce(nonce)
        .setCloudProjectNumber(CLOUD_PROJECT_NUMBER)
        .build()
).addOnSuccessListener { response ->
    val token = response.token() // opaque, verify server-side
    sessionApi.bootstrap(token, nonce)
}
```

What It Proves, and What It Doesn't



THIS LAYER BUYS

A genuine app on a genuine, unmodified device, proven at the moment the token was requested.



THIS LAYER CANNOT STOP

Nothing, but it's too expensive in latency and rate limits to run on every single event. It's built to gate something longer-lived, not to sign each call.

→ NEXT: SPEND THAT PROOF ONCE, AS A SESSION

No Play Services, No Attestation

GOTCHA

Play Integrity has no meaning on a device without Google Play Services. Any SDK shipping to markets with significant GMS-less Android usage needs a real fallback path, not a hard dependency that silently fails closed, or worse, silently fails open.

Non-Google Devices Need a Different Answer

Play Integrity assumes Google Play Services. That assumption breaks in markets where it doesn't ship: Huawei devices since 2019 run HMS Core instead of GMS, most Android devices sold in China have no Google services at all, and Amazon's Fire OS devices never had them to begin with. None of that traffic is fake, it's just unattestable through Google's API.

Alternatives Worth Knowing

HUAWEI

Safety Detect

Part of HMS Core, the closest direct equivalent to Play Integrity. Covers Huawei hardware only.

SAMSUNG

Knox Attestation

Hardware-backed attestation on Knox-enabled devices. Narrower scope, Samsung hardware only.

EVERYONE ELSE

No direct equivalent

Fall back to weaker signals: device fingerprinting, root and emulator heuristics, a wider server-side margin instead of a hard gate.

WORTH PLANNING FOR

There's no single API that covers every device. If your user base includes these markets, building the fallback path is not optional.

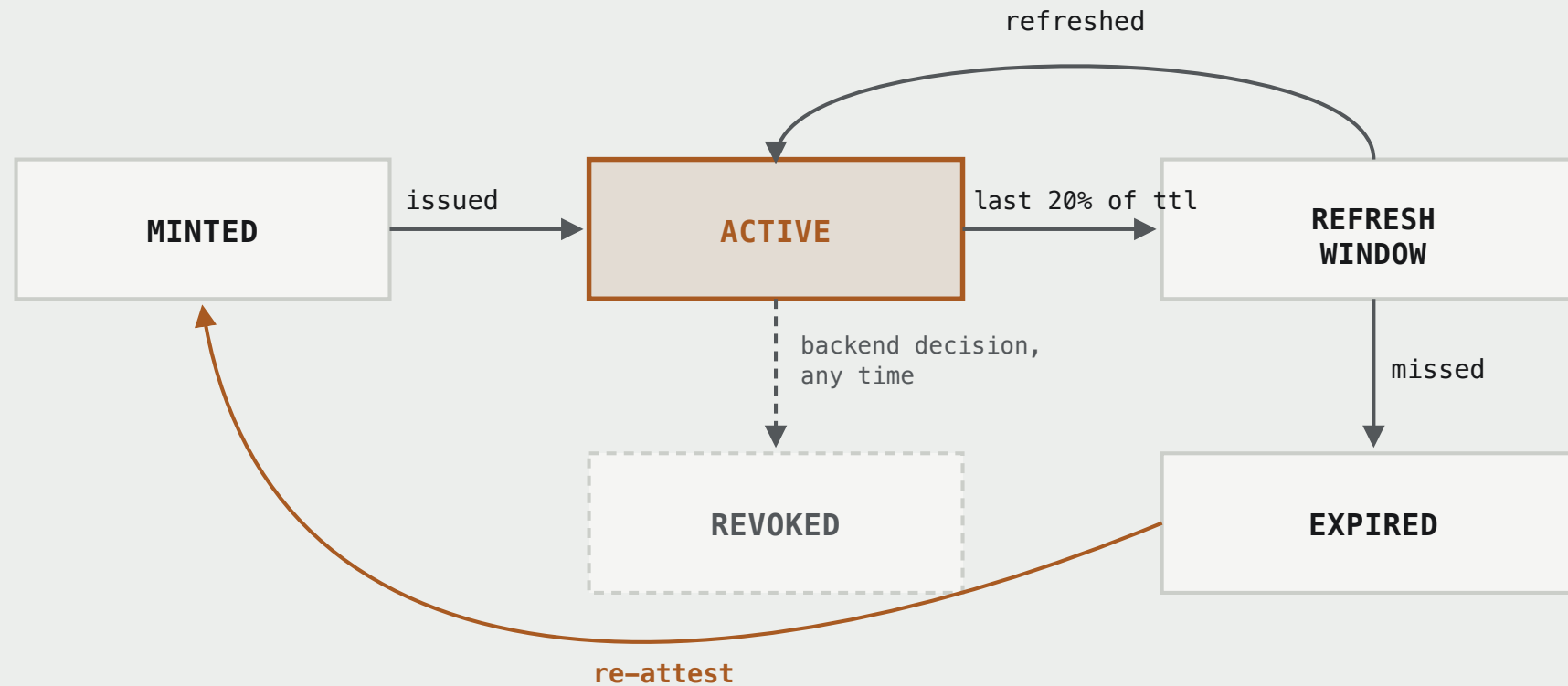
The Shift: Static Keys → Short-Lived Tokens

A static API key baked into the SDK is a permanent liability: once extracted, it's valid until someone manually rotates it, if ever. The fix isn't a stronger key. It's shipping no long-lived credential at all.

The Attested Handshake, at Init



The Token's Lifecycle



Bootstrapping a Session

● SESSIONBOOTSTRAP.KT

```
suspend fun bootstrapSession(): Session {  
    val nonce = generateRequestNonce()  
    val attestation = integrity.requestIntegrityToken(nonce)  
  
    val session = authApi.post("/v1/session") {  
        body = SessionRequest(attestation, nonce)  
    }  
    // session.token is the HMAC key for every call until it expires  
    tokenStore.save(session.token, session.expiresAt)  
    scheduleRefresh(before = session.expiresAt)  
    return session  
}
```

Eliminating Long-Lived Credential Compromise

Nothing permanent to extract

There's no static secret sitting in the binary anymore, only a token minted after proof.

A leaked token dies in minutes

A 15-minute TTL turns a catastrophic leak into a non-event.

Revocation is a backend decision

No app release required to cut off a compromised credential.

Every session traces to a real device

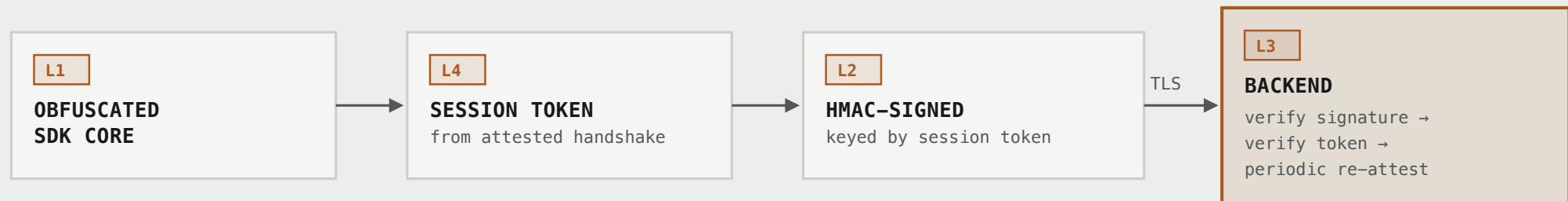
You can't get a token without first passing attestation.

Refresh Before It Dies, Not After

GOTCHA

A 15 minute TTL sounds generous until a user backgrounds the app for an hour and the very next foreground event needs a token that already expired. Refresh proactively, on a timer before expiry, not reactively after a request has already failed.

All Four Layers, One Request



Every outgoing call is obfuscated at rest, authorized by an attested token, and signed in transit: three independent checks, not one.

Which Layer Stops Which Attack

ATTACK	OBFUSCATION	SIGNING	ATTESTATION	TOKENS
Static analysis / decompilation	●	—	—	—
Static secret extraction	◐	—	—	●
Replay attack	—	●	—	◐
Fabricated request, no real client	—	◐	●	◐
Rooted device / emulator farm	—	—	●	◐
Long-lived credential leak	—	—	—	●

● mitigates ◐ partially mitigates — no coverage. No single layer clears the board: that's the argument for all four.

Sequencing This in a Real SDK

1

Ship obfuscation

No server changes. Immediate protection against casual decompilation.

2

Add request signing on the client

Distribute a shared secret through your existing SDK config channel.

3

Add server-side signature verification

Reject unsigned and stale requests before they touch business logic.

4

Integrate device attestation

Android and GMS only. Gate it at SDK init, not per event.

5

Cut over to session tokens

Run both paths in parallel, watch adoption, then sunset the static key.

6

Re-tune thresholds against real telemetry

Timestamp windows and token TTLs are dials, not defaults. Revisit them once traffic tells you something.

Key Takeaways

- 01 **Obfuscation buys time. It doesn't buy trust.**
 - 02 **Signing proves a request wasn't tampered with, not that it came from a real client.**
 - 03 **Attestation proves the client is real, once, at a cost too high to spend on every call.**
 - 04 **Session tokens turn that one proof into a currency safe enough to spend for the whole session.**
-

Tools & Further Reading

TOOLS

apktool, jadx, Frida, R8 (developer.android.com/build/shrink-code), DexGuard (guardsquare.com/dexguard)

PLATFORM

Play Integrity API (developer.android.com/google/play/integrity), Huawei Safety Detect (developer.huawei.com)

STANDARDS

OWASP MASVS, the Mobile Application Security Verification Standard (owasp.org/www-project-mobile-app-security)

READING

"A follow-up on how to store tokens securely in Android," Medium, Google Developer Experts

Run It Yourself

REPO	<code>github.com/kikoso/security-demo</code>
RUN	<code>./gradlew run</code> — no Android SDK, no emulator, no device
COVERS	Obfuscated string decode, HMAC request signing with replay rejection, an attested session-token handshake over a real localhost HTTP server
REFERENCE	The real Android Play Integrity client, uncompiled, alongside the JVM mock it stands in for

Questions?

Four layers, one request, no single point of trust.