Flick Security & Error Handling

Defense-in-depth architecture with comprehensive error prevention and recovery

SECURITY Security Architecture

Flick implements **defense-in-depth security** across multiple layers.

Security Layers

1. Input Validation & Sanitization

All user input and external data is validated and sanitized before use

2. Network Security (HTTPS Only)

All API communications use TLS encryption, no plaintext data transmission

3. Data Privacy (No PII Collection)

Anonymous user IDs, no personal information stored or transmitted

4. Firestore Security Rules

Server-side authorization prevents unauthorized data access

5. Dependency Security Scanning

Snyk and OWASP checks for known vulnerabilities in dependencies

VALIDATION Input Validation & Sanitization

1. Video ID Validation

❌ Vulnerable Code

```
// Direct use without
validation
val uri =
"https://archive.org/
  download/$videoId/$filename

// Path traversal attack
possible:
// videoId =
"../../../etc/passwd"
```

✅ Secure Code

```
// Validate video ID format
fun isValidVideoExtension(
  filename: String
): Boolean {
  if
(!filename.contains("."))
return false

  val ext = filename
    .substringAfterLast('.')
    .lowercase()

  return ext in
VALID_VIDEO_EXTENSIONS
}

// Sanitize before use
val encodedFile =
Uri.encode(
  fileName,
  "@#&=*+-_.,:!?()/~'%"
)
```

2. Video Extension Whitelist

```
val VALID_VIDEO_EXTENSIONS = setOf(
  // Modern mobile & web
  ".mp4", ".mov", ".webm", ".m4v",
```

```
  // Older mobile
  ".3gp", ".3g2",

  // Desktop formats
  ".mkv", ".avi", ".wmv", ".mpg", ".mpeg", ".flv"
)

// Reject all other extensions
if (!isValidVideoExtension(filename)) {
  throw IllegalArgumentException(
    "Invalid video file type: $filename"
  )
}
```

3. Video ID Format Validation

```
fun hasValidVideoExtension(videoId: String): Boolean {
  // DVD set videos (format: "dvdId|filename")
  if (videoId.contains("|")) {
    val filename = videoId.substringAfter("|").trim()

    // Empty filename = invalid
    if (filename.isEmpty()) return false

    // Must have extension
    if (!filename.contains(".")) return false

    val ext = filename
      .substringAfterLast('.')
      .lowercase()

    return ext in VALID_VIDEO_EXTENSIONS
  }

  // Archive.org IDs without extensions are valid
  return true
}
```

4. Firestore Data Sanitization

```
private fun safeId(raw: String): String {
  return raw
    .replace("/", "_")  // Path separators
    .replace("|", "_")  // Pipe characters
    .replace("\\s".toRegex(), "_")  // Whitespace
}

// Usage
val videoId = itemIdentifiers[currentIndex]
val safeVideoId = safeId(videoId)
RatingsRepo.like(safeVideoId)
```

⚠️ **Why This Matters:** Without sanitization, special characters in video IDs could break Firestore document paths, cause injection attacks, or corrupt data.

SECURITY Network Security

1. HTTPS-Only Communication

```
// All API endpoints use HTTPS
const val ARCHIVE_BASE_URL = "https://archive.org"
const val ARCHIVE_STREAM_URL = "https://archive.org/download"

// Retrofit enforces HTTPS
private val retrofit = Retrofit.Builder()
  .baseUrl(ARCHIVE_BASE_URL)
  .client(okHttpClient)
  .build()

// No HTTP fallback allowed
```

2. TLS/SSL Configuration

```
private val okHttpClient = OkHttpClient.Builder()
  // Use system default TLS (1.2+)
```

```
    .connectionSpecs(
      listOf(ConnectionSpec.MODERN_TLS)
    )

    // Certificate pinning (optional)
    .certificatePinner(
      CertificatePinner.Builder()
        .add("archive.org", "sha256/...")
        .build()
    )

    .build()
```

3. API Key Protection

❌ Hardcoded Keys

```
// NEVER do this!
const val API_KEY =
"sk_live_..."

// Exposed in APK
decompilation
// Committed to git history
// Visible in crash logs
```

✅ Secure Storage

```
// In local.properties
(gitignored)
FIREBASE_API_KEY=AIza...

// In build.gradle.kts
android {
  buildTypes {
    release {
      buildConfigField(
        "String",
        "API_KEY",
        "\"${project.properti
          ["FIREBASE_API_KEY"
      )
    }
  }
}
```

4. Request Timeout Protection

```
private val okHttpClient = OkHttpClient.Builder()
  // Prevent hanging connections
  .connectTimeout(5, TimeUnit.SECONDS)
  .readTimeout(10, TimeUnit.SECONDS)
  .callTimeout(12, TimeUnit.SECONDS)

  // Fail fast, don't wait indefinitely
  .retryOnConnectionFailure(false)

  .build()
```

SECURITY

Firestore Security Rules

Server-side authorization prevents unauthorized data access and manipulation.

User Data Protection

```
rules_version = '2';
service cloud.firestore {
  match /databases/{database}/documents {

    // Users can only read/write their own data
    match /users/{userId} {
      allow read: if request.auth != null
        && request.auth.uid == userId;

      allow write: if request.auth != null
        && request.auth.uid == userId
        && request.resource.data.keys()
          .hasOnly(['viewTimes', 'ratings',
                    'lastActive']);
    }

    // Public video stats (read-only for users)
    match /videos/{videoId} {
      allow read: if true;
```

```
      // Only Cloud Functions can write
      allow write: if false;
    }
  }
}
```

Data Validation Rules

```
// Validate view time is reasonable
match /users/{userId} {
  allow write: if request.auth.uid == userId
    && request.resource.data.viewTimes is map
    && request.resource.data.viewTimes.values()
      .every(v => v >= 0 && v <= 7200000);
    // Max 2 hours per video
}
```

✅ **Security Benefits:** Even if client-side validation is bypassed, server-side rules prevent malicious data writes, data theft, and privilege escalation.

ERROR Comprehensive Error Handling

Flick implements **graceful degradation** - errors are caught, logged, and recovered from without crashing.

1. Network Error Handling

Timeout Protection

```
suspend fun loadMetadata(videoId: String): ArchiveMetadata {
  return try {
    withTimeout(20_000) {  // 20s max
      ApiClient.service.getItemMetadata(videoId)
    }
  } catch (e: TimeoutCancellationException) {
```

```
    Timber.e("Timeout fetching $videoId")
    ArchiveMetadata(files = emptyList())
  } catch (e: Exception) {
    Timber.e(e, "Error fetching $videoId")
    ArchiveMetadata(files = emptyList())
  }
}
```

↓

Retry Logic

```
suspend fun fetchWithRetry(
  videoId: String,
  maxRetries: Int = 3
): ArchiveMetadata? {
  repeat(maxRetries) { attempt ->
    try {
      return loadMetadata(videoId)
    } catch (e: Exception) {
      if (attempt == maxRetries - 1) throw e
      delay(1000 * (attempt + 1))  // Backoff
    }
  }
  return null
}
```

↓

Fallback Strategy

```
val metadata = try {
  loadMetadata(videoId)
} catch (e: Exception) {
  Timber.e(e, "Failed to load metadata")

  // Fallback to cached data
  metadataCache[videoId]
    ?: ArchiveMetadata(files = emptyList())
}
```

2. Player Error Handling

```
player.addListener(object : Player.Listener {
  override fun onPlayerError(error: PlaybackException) {
    val idx = playerPool.entries
      .find { it.value == this@apply }?.key

    Timber.e(
      "Player $idx error: ${error.message}, "
      + "code: ${error.errorCode}"
    )

    // Handle persistent errors
    if (error.errorCode ==
        PlaybackException.ERROR_CODE_IO_BAD_HTTP_STATUS
      || error.errorCode == 2004  // Source error
      || error.errorCode == 3003) {  // Decoder error

      // Mark video as broken
      viewModel.markVideoAsBroken(idx)

      // Auto-advance if current
      if (idx == currentActiveIndex) {
        viewModel.onSwipeUp()  // Skip broken video
      }
    }
  }
})
```

3. Coroutine Exception Handling

```
// Global coroutine exception handler
val handler = CoroutineExceptionHandler { _, exception ->
  Timber.e(exception, "Uncaught coroutine exception")

  // Log to Crashlytics
  FirebaseCrashlytics.getInstance()
    .recordException(exception)
}

// Use in ViewModel
```

```
viewModelScope.launch(handler) {
  // Coroutine code
}
```

4. Null Safety & Bounds Checking

❌ Crash-Prone Code

```
// NullPointerException risk
val videoId = itemIdentifiers[index]
val metadata = loadMetadata(videoId)
val title = metadata.title

// IndexOutOfBoundsException risk
val nextVideo = itemIdentifiers[index + 1]
```

✅ Safe Code

```
// Safe navigation
val videoId = itemIdentifiers
  .getOrNull(index) ?: return

val metadata = loadMetadata(videoId)
val title = metadata?.title
  ?: "Unknown"

// Bounds checking
if (index + 1 in itemIdentifiers.indices) {
  val nextVideo = itemIdentifiers[index + 1]
}
```

5. State Validation

```
fun onSwipeUp() {
  // Validate state before proceeding
  if (itemIdentifiers.isEmpty()) {
    Timber.e("Cannot swipe - no videos")
    return
  }

  if (currentIndex >= itemIdentifiers.size) {
    Timber.e("Invalid index: $currentIndex")
```

```
    return
  }

  // Safe to proceed
  proceedToNextVideo()
}
```

CRASHLYTICS Firebase Crashlytics Integration

Automatic crash reporting with custom logging for **proactive issue detection**.

1. Setup & Initialization

```
// In build.gradle.kts
plugins {
  id("com.google.gms.google-services")
  id("com.google.firebase.crashlytics")
}

dependencies {
  implementation(platform(
    "com.google.firebase:firebase-bom:32.7.0"
  ))
  implementation(
    "com.google.firebase:firebase-crashlytics-ktx"
  )
  implementation(
    "com.google.firebase:firebase-analytics-ktx"
  )
}
```

```
// In Application class
class FlickApplication : Application() {
  override fun onCreate() {
    super.onCreate()

    // Initialize Crashlytics
```

```
    FirebaseCrashlytics.getInstance()
      .setCrashlyticsCollectionEnabled(!DEBUG_MODE)

    // Setup Timber with Crashlytics
    if (DEBUG_MODE) {
      Timber.plant(Timber.DebugTree())
    } else {
      Timber.plant(CrashlyticsTree())
    }
  }
}
```

2. Custom Crashlytics Tree

```
class CrashlyticsTree : Timber.Tree() {
  override fun log(
    priority: Int,
    tag: String?,
    message: String,
    t: Throwable?
  ) {
    // Only log warnings and errors
    if (priority < Log.WARN) return

    val crashlytics = FirebaseCrashlytics.getInstance()

    // Add custom log
    crashlytics.log("$tag: $message")

    // Record exception if present
    t?.let {
      crashlytics.recordException(it)
    }
  }
}
```

3. Custom Keys for Context

```
// Track important state
fun updateCrashlyticsContext() {
```

```
  val crashlytics = FirebaseCrashlytics.getInstance()

  crashlytics.setCustomKey("current_index", currentIndex)
  crashlytics.setCustomKey("video_id",
    itemIdentifiers.getOrNull(currentIndex) ?: "none")
  crashlytics.setCustomKey("pool_size",
    playerPoolManager?.playerPool?.size ?: 0)
  crashlytics.setCustomKey("queue_size",
    predeterminedQueue.size)
  crashlytics.setCustomKey("sideways_chain",
    sidewaysHistory.isNotEmpty())
}

// Call on state changes
fun onVideoAdvanced(newIndex: Int) {
  currentIndex = newIndex
  updateCrashlyticsContext()
}
```

4. Non-Fatal Exception Logging

```
// Log non-fatal errors for analysis
try {
  buildVideoUri(videoId)
} catch (e: Exception) {
  // Log to Crashlytics but don't crash
  FirebaseCrashlytics.getInstance()
    .recordException(e)

  Timber.e(e, "Non-fatal: Failed to build URI")

  // Use fallback
  buildFallbackUri(videoId)
}
```

5. User Identification (Anonymous)

```
// Set anonymous user ID
val userId = UUID.randomUUID().toString()
```

```
FirebaseCrashlytics.getInstance()
  .setUserId(userId)

// Track user cohorts
FirebaseCrashlytics.getInstance()
  .setCustomKey("app_version", BuildConfig.VERSION_NAME)
  .setCustomKey("android_version", Build.VERSION.SDK_INT)
  .setCustomKey("device_model", Build.MODEL)
```

✅ **Benefits:** Crashlytics provides detailed crash reports with full stack traces, custom context, breadcrumbs, and device info - enabling rapid bug fixes.

🛡️ Error Prevention Strategies

1. Kotlin Null Safety

Leverage Kotlin's type system to eliminate NullPointerExceptions

```
// Non-null by default
var currentIndex: Int = 0

// Explicit nullable
var videoId: String? = null

// Safe calls
val title = metadata?.title

// Elvis operator
val name = title ?:
"Unknown"

// Safe casts
```

2. Immutable Data Structures

Prevent accidental state mutations

```
// Immutable list (read-
only)
val queue: List<Int> =
_queue.toList()

// Immutable map
val mapping: Map<String,
String> =
  _mapping.toMap()

// Data class (immutable by
default)
data class VideoState(
  val index: Int,
  val isPlaying: Boolean
)
```

```
val player = obj as?
ExoPlayer
```

3. Sealed Classes for State

Exhaustive when expressions prevent missing cases

```
sealed class UiEvent {
  object ShowToast :
UiEvent()
  object ShowError :
UiEvent()
  data class Navigate(
    val route: String
  ) : UiEvent()
}

// Compiler enforces all
cases
when (event) {
  is UiEvent.ShowToast ->
{}
  is UiEvent.ShowError ->
{}
  is UiEvent.Navigate -> {}
  // No 'else' needed -
exhaustive
}
```

4. Resource Cleanup

Prevent memory leaks with lifecycle-aware cleanup

```
DisposableEffect(player) {
  val listener = object :
Player.Listener {
    override fun
onIsPlayingChanged(
      isPlaying: Boolean
    ) {
      // Handle state
    }
  }

  player.addListener(listener

  onDispose {
    // Cleanup
    player.removeListener(lis
  }
}
```

5. Type-Safe Configuration

Use enums instead of magic strings/numbers

6. Defensive Copying

Prevent external mutations of internal state

```
enum class OriginEnum {
  RANDOM,
  RECOMMENDED,
  TOP_PERCENTILE,
  SIDEWAYS
}

// Type-safe usage
val origin: OriginEnum =
OriginEnum.RANDOM

// Instead of error-prone
strings
// val origin =
"random"  // Typo risk
```

```
// Internal mutable state
private val _queue =
mutableListOf<Int>()

// Public immutable copy
val queue: List<Int>
  get() = _queue.toList()

// Caller can't modify
internal state
val q = viewModel.queue
q.add(5)  // Compile error!
```

✅ Security & Error Handling Checklist

- ✓ All user input validated and sanitized
- ✓ HTTPS-only network communication
- ✓ Firestore security rules enforced server-side
- ✓ No API keys hardcoded in source
- ✓ All network calls wrapped in try-catch
- ✓ Player errors handled with auto-recovery
- ✓ Null safety with safe navigation operators
- ✓ Bounds checking on all array access
- ✓ Crashlytics integrated for crash reporting
- ✓ Resource cleanup in onDispose/onCleared

- ✓ Timeout protection on all async operations
- ✓ Dependency security scanning (Snyk, OWASP)
- ✓ No PII collected or transmitted
- ✓ Graceful degradation on errors
- ✓ ProGuard/R8 obfuscation in release builds

Security-First Design | Zero-Crash Goal