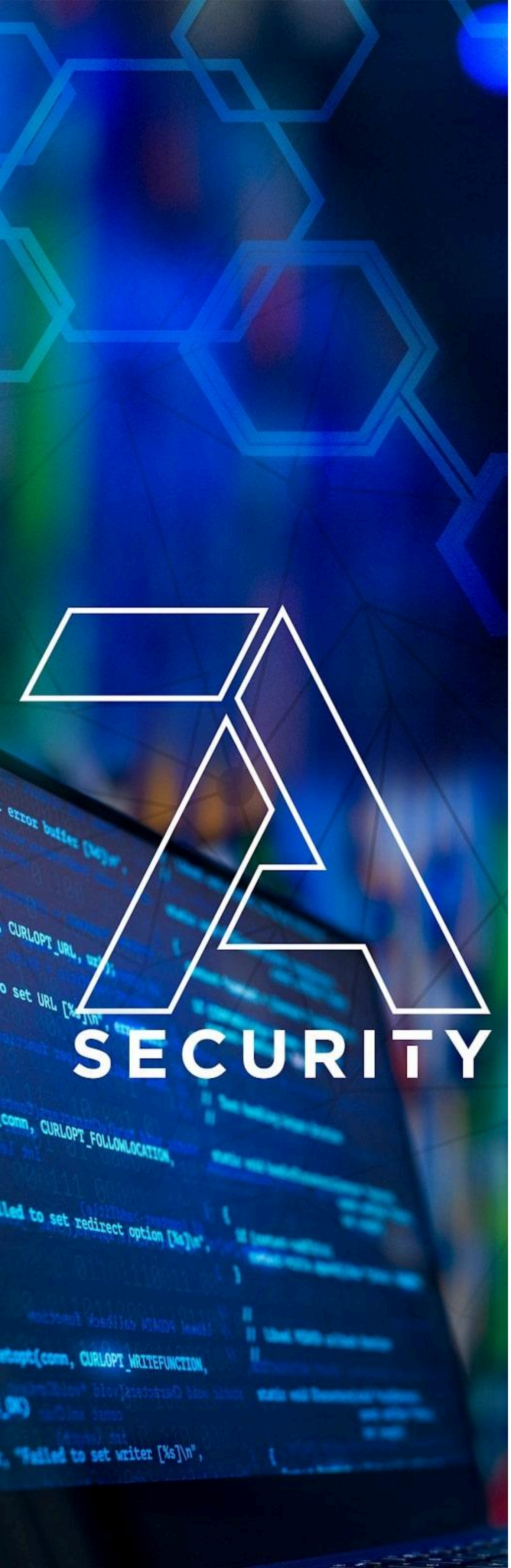

ISO/IEC 27001:2022
ISMS Certified
by Consilium Labs (IAS)

Pentest Report

Client:
Noghteha Team

Noghteha Test Targets:

Android App
Privacy Audit

7ASecurity Test Team:

- Abraham Aranguren, MSc.
- Daniel Ortiz, MSc.
- Dheeraj Joshi, BTech.
- Miroslav Štampar, PhD.

INDEX

Introduction

*“**Connect Without Internet***
Secure, decentralized messaging for when the internet goes dark.
Stay connected with Bluetooth mesh networking.”

From https://noghteha.app/en/

This document outlines the results of a whitebox security and privacy audit conducted against the Noghteha platform. The project was solicited by the Noghteha Team and executed by 7ASecurity in January 2026. The audit team dedicated 12 working days to complete this assignment. Please note that this is the first penetration test for this project. Consequently, the identification of security weaknesses was expected to be easier during this engagement, as more vulnerabilities are identified and resolved after each testing cycle.

During this iteration the goal was to review the solution as thoroughly as possible, to ensure Noghteha users can be provided with the best possible security and privacy. The methodology implemented was *whitebox*: 7ASecurity was provided with access to release and debug builds, documentation, and source code. A team of 4 senior auditors carried out all tasks required for this engagement, including preparation, delivery, documentation of findings and communication.

A number of necessary arrangements were in place by January 2026, to facilitate a straightforward commencement for 7ASecurity. In order to enable effective collaboration, information to coordinate the test was relayed through email, as well as a shared Signal group chat. The Noghteha team was helpful and responsive throughout the audit, which ensured that 7ASecurity was provided with the necessary access and information at all times, thus avoiding unnecessary delays. 7ASecurity provided regular updates regarding the audit status and its interim findings during the engagement.

This audit split the scope items into the following work packages, which are referenced in the ticket headlines as applicable:

- WP1: Whitebox Tests against Noghteha Android Application
- WP2: Noghteha Privacy Audit

The findings of the security audit can be summarized as follows:

Identified Vulnerabilities	*Hardening Recommendations*	*Total Issues*
10	*8*	*18*

Please note that the analysis of the remaining work package (WP2) is provided separately, in the following section of this report:

- WP2: Noghteha Privacy Audit

Moving forward, the scope section elaborates on the items under review, while the findings section documents the identified vulnerabilities followed by hardening recommendations with lower exploitation potential. Each finding includes a technical description, a proof-of-concept (PoC) and/or steps to reproduce if required, plus mitigation or fix advice for follow-up actions by the development team.

Finally, the report culminates with a conclusion providing detailed commentary, analysis, and guidance relating to the context, preparation, and general impressions gained throughout this test, as well as a summary of the perceived security posture of the Noghteha applications.

Scope

The following list outlines the items in scope for this project:

- **WP1: Whitebox Tests against Noghteha Android Application**
    - https://github.com/filtershekanha/Noghteha_Android
        - Noghteha Android Version: 1.0.34
- **WP2: Privacy Audit of Noghteha Android Application**
    - As above

Identified Vulnerabilities

This area of the report enumerates findings that were deemed to exhibit greater risk potential. Please note that these are offered sequentially as they were uncovered, they are not sorted by significance or impact. Each finding has a unique ID (i.e. *NOG-01-001*) for ease of reference, and offers an estimated severity in brackets alongside the title.

NOG-01-004 WP1: Lack of Biometric Authentication *(Medium)*

The Noghteha Android application does not implement password or biometric authentication to protect the chat functionality, even after being closed or left idle for an extended period. This lack of security significantly increases the risk of unauthorized access in high-threat scenarios such as device seizure at protests or physical confiscation by authorities. If a physical attacker gains access to an unlocked phone, they can fully interact with sensitive functionality, such as reading all messages, viewing contact lists, analyzing communication patterns, and accessing mesh network configurations, without bypassing any additional authentication. This exposes users to severe risks, including identification, surveillance, and potential harm, as all chat data and metadata can be accessed without further authentication steps beyond the device lock screen.

To improve security and safeguard user data, it is advised to implement biometric authentication, such as fingerprint or facial recognition, when reopening the app after inactivity. This added layer of security ensures that only the authorized user can access sensitive chat messages. For technical guidance, consult the *OWASP MASTG Android Local Authentication*[1] documentation to integrate biometric security properly.

NOG-01-005 WP1: Chat Message Access via Memory Leak *(Medium)*

Retest Notes: Fixed by Noghteha and verified by 7ASecurity.

It was found that chat message content was retained in the Noghteha Android app process memory. If an attacker obtains privileged access sufficient to capture a process memory dump (for example, via physical access with debugging enabled, root access, or local privilege escalation), chat content can be recovered from memory, increasing the impact of device compromise in high-threat environments.

To confirm this issue, the app process memory was dumped and reviewed for retained chat content. A search for a known message string identified an occurrence in memory.

[1] https://mas.owasp.org/MASTG/0x05f-Testing-Local-Authentication/

Command:
```
strings ./314572800_dump.data | grep "w00tw00t"
```

Output:
```
this is a supersecret message: w00tw00t
Message from noghteh1337: this is a supersecret message: w00tw00t
[...]
```

It is recommended to minimize retention of chat content and key material in long-lived in-memory structures and to clear buffers when the data is no longer required (for example, on chat exit, logout, or after message processing). For key material, storage in immutable objects (for example, *java.lang.String*) should be avoided where feasible, and *zeroization* should be performed for byte arrays and other mutable buffers after use. Immediate removal from memory cannot be guaranteed in a managed runtime due to nondeterministic garbage collection; therefore, retention should be minimized and debug or diagnostic caches should be bounded and cleared. For additional mitigation guidance, please see the *Testing Memory for Sensitive Data* section of the *Mobile Application Security Testing Guide (MASTG)*[2].

NOG-01-006 WP1: DoS via Unbounded Concurrent Handshakes *(Medium)*

Retest Notes: Fixed by Noghteha and verified by 7ASecurity.

A resource exhaustion vulnerability exists in the handshake coordination subsystem regarding the handling of pending handshake sessions. While the *MeshEncryptionCoordinator* attempts to enforce a limit on concurrent handshakes (*MAX_CONCURRENT_HANDSHAKES = 10*), this limit is enforced only on outgoing initiation and is bypassed by incoming handshake requests. Furthermore, the mitigation strategy for outgoing requests introduces a secondary denial-of-service vector via coroutine exhaustion.

The application derives a *Peer ID* directly from the raw bytes of the *senderID* field in incoming packets. Since this identity is unauthenticated during the initial handshake phase, an attacker can perform a *Sybil* attack by broadcasting packets with randomly generated *senderIDs*. When processing these incoming handshake messages, the *processHandshakeMessage* function adds entries to the *pendingHandshakes* map without checking the global limit, allowing unbounded memory growth. Concurrently, when the limit is reached for outgoing requests, the *initiateHandshake* function launches a new coroutine to retry after a delay. Under a flood of requests, this results in an O(N) increase in suspended coroutines, exhausting system resources.

[2] https://mas.owasp.org/MASTG/tests/android/MASVS-STORAGE/MASTG-TEST-0011/

Affected File:
app/src/main/java/com/filtershekanha/noghteha/mesh/MeshEncryptionCoordinator.kt

Affected Code:
```
suspend fun processHandshakeMessage(routed: RoutedPacket): ByteArray? {
    val peerID = routed.peerID ?: return null
    [...]
    // Track this handshake if we haven't seen it
    if (!pendingHandshakes.containsKey(peerID)) {
        pendingHandshakes[peerID] = HandshakeState(peerID, initiatedByUs = false)
        _encryptionEvents.emit(EncryptionEvent.HandshakeStarted(peerID))
    }
    [...]
}
[...]
fun initiateHandshake(peerID: String, remoteNoisePublicKey: ByteArray? = null) {
    [...]
    // Check concurrent handshake limit
    if (pendingHandshakes.size >= MAX_CONCURRENT_HANDSHAKES) {
        Log.w(TAG, "Max concurrent handshakes reached, queueing handshake for $peerID")
        coordinatorScope.launch {
            delay(1000) // Wait and retry
            initiateHandshake(peerID, remoteNoisePublicKey)
        }
        return
    }
    [...]
}
```

It is recommended to enforce *MAX_CONCURRENT_HANDSHAKES* strictly for both incoming and outgoing handshake attempts. In *processHandshakeMessage*, the limit should be checked before creating a new *HandshakeState*; if the limit is reached, the incoming request should be dropped without creating a *pendingHandshakes* entry and a warning can be logged. The recursive coroutine pattern in *initiateHandshake* should be replaced with a bounded Channel or a single background job that drains a queue of pending requests to prevent unbounded coroutine creation. This ensures that *pendingHandshakes* never exceeds the configured limit under any traffic conditions, incoming handshake floods are dropped without allocating map entries, and outgoing handshake floods do not spawn excessive coroutines.

NOG-01-007 WP1: Unsafe Debug UI in Release Builds *(Medium)*

Retest Notes: Fixed by Noghteha and verified by 7ASecurity.

A design vulnerability exists in *DebugSettingsManager.kt* and *DebugSettingsSheet.kt* due to the inclusion of unsafe diagnostic features in release builds without adequate safeguards. The application ships with a fully functional debug interface (*DebugSettingsSheet*) and a backing singleton manager (*DebugSettingsManager*) that allows modifying critical system parameters at runtime. These settings are persisted to *SharedPreferences* via *DebugPreferenceManager*, meaning unsafe configurations survive application restarts.

The exposed functionality enables trivial DoS and increased data exposure in release builds. The *setMaxConnectionsOverall* method, accessible through a UI slider, allows the mesh connection limit to be reduced to as low as 1 without enforcing a safe minimum (for example, 3-4 peers), making it easy to accidentally or maliciously isolate the node. In addition, *setVerboseLoggingEnabled* causes decrypted packet metadata and connection details to be retained in the in-memory *debugMessageQueue*, creating a persistent cache of sensitive information on the Java heap that would otherwise be transient, thereby increasing the impact of memory-corruption issues or post-seizure forensic analysis.

Because these methods are public, unguarded, and compiled into the release build, they serve as “gadgets” that can be chained with other vulnerabilities (such as the WebView JavaScript bridge in NOG-01-009) to escalate a minor compromise into a persistent DoS or data leak.

Affected Files:
app/src/main/java/com/filtershekanha/noghteha/ui/debug/DebugSettingsManager.kt
app/src/main/java/com/filtershekanha/noghteha/ui/debug/DebugSettingsSheet.kt

Affected Code:
```
fun setMaxConnectionsOverall(value: Int) {
    val clamped = value.coerceIn(1, 32)
    DebugPreferenceManager.setMaxConnectionsOverall(clamped)
    _maxConnectionsOverall.value = clamped
    addDebugMessage(DebugMessage.SystemMessage("Max overall connections set to
$clamped"))
}
[...]
fun addDebugMessage(message: DebugMessage) {
    [...]
    debugMessageQueue.offer(message)
    [...]
}
```

It is recommended to wrap debug logic and UI initialization in *BuildConfig.DEBUG* checks to ensure that these features are disabled or unreachable in production builds. Additionally, *DebugPreferenceManager* should ignore or clear any debug overrides in release builds to prevent persistence from testing environments. This can reduce field diagnosability; if diagnostics are required, a separate, explicitly controlled mechanism should be provided.

If field diagnostics must remain available in release builds, it is recommended to gate access with explicit user authentication (e.g., device biometrics) or a dedicated unlock sequence before *DebugSettingsManager* is enabled. Safe limits should be enforced by changing *coerceIn(1, 32)* to *coerceIn(MIN_SAFE_CONNECTIONS, 32)*, and “Verbose Logging” should be automatically disabled after a short timeout (e.g., 15 minutes), with the in-memory queue bounded and cleared when the application is backgrounded.

NOG-01-008 WP1: Channel KDF Precomputation Risk *(Medium)*

Retest Notes: Fixed by Noghteha and verified by 7ASecurity.

A cryptographic weakness exists in *ChannelManager.kt* regarding the key derivation mechanism for mesh channels that can reduce resistance to offline password guessing if encrypted traffic is captured. The application utilizes a deterministic salt derived from a fixed prefix and the channel name combined with PBKDF2 to generate encryption keys. This architectural design enables "stateless" channel joining, allowing users to derive the correct key for a public channel solely from its name and password without prior network negotiation, but it fails to effectively mitigate precomputation attacks for common channel names.

Because the salt is globally predictable for any given channel name (e.g., “General”), and PBKDF2 is not memory-hard and is amenable to GPU and ASIC acceleration, a sophisticated adversary can precompute password-guessing tables for common channel names. If encrypted traffic is intercepted, the adversary can use these precomputed tables to amortize guesses across many captures, significantly reducing the attack cost to recover passwords compared to a random per-channel salt implementation. The deterministic salt does not provide per-channel randomness, which increases reliance on a memory-hard derivation function to resist GPU acceleration, which the current PBKDF2 configuration does not satisfy.

Affected File:
app/src/main/java/com/filtershekanha/noghteha/ui/ChannelManager.kt

Affected Code:

```
private fun deriveChannelKey(password: CharArray, channelName: String): SecretKeySpec {
    // PBKDF2 key derivation (same as iOS version)
    val factory = javax.crypto.SecretKeyFactory.getInstance("PBKDF2WithHmacSHA256")
    // Use application-specific prefix + channel name as salt to prevent rainbow table
attacks
    val saltPrefix = "com.filtershekanha.noghteha.channel.v1:"
    try {
        val spec = javax.crypto.spec.PBEKeySpec(
            password,
            (saltPrefix + channelName).toByteArray(),
            100000, // 100,000 iterations (same as iOS)
            256 // 256-bit key
        )
        try {
            val secretKey = factory.generateSecret(spec)
            return SecretKeySpec(secretKey.encoded, "AES")
        } finally {
            spec.clearPassword()
        }
    } finally {
        CryptoUtils.secureZero(password)
    }
}
```

It is recommended to maintain stateless channel joining while mitigating precomputation risk by migrating from PBKDF2 to a memory-hard key derivation function such as Argon2id[3]. Argon2id should be configured with mobile-appropriate parameters (e.g., 32-64 MB RAM, 1-3 iterations), targeting a derivation time of 150-300 ms on low-end devices. This can increase CPU and memory usage; the impact should be mitigated by tuning parameters and performing derivation off the UI thread. KDF versioning (e.g., v1=PBKDF2, v2=Argon2id) should be implemented to facilitate migration and support legacy channels if necessary. Alternatively, secure channels can be introduced that use a random salt generated upon creation and distributed alongside the channel name via invite links to ensure true per-channel uniqueness, at the cost of requiring an out-of-band invite for joining.

[3] https://cheatsheetseries.owasp.org/cheatsheets/Password_Storage_Cheat_Sheet.html

NOG-01-010 WP1: DoS via Resource Exhaustion in File Sharing *(High)*

Retest Notes: Fixed by Noghteha and verified by 7ASecurity.

A DoS vulnerability exists in the file sharing stack due to inefficient memory management and quadratic-time decoding logic. This can allow an attacker-controlled peer to exhaust CPU and heap resources during file transfers, potentially causing application crashes or an unresponsive service. While the application attempts to limit file sizes to 100 MB, the implementation of the *NoghtehaFilePacket* decoder and the *FileSharingManager* logic allows for significant resource exhaustion even within those limits. Specifically, the *NoghtehaFilePacket.decode* function implements a “concatenate-on-read” strategy for file content. For every *CONTENT TLV* received, the application uses the + operator to merge the new fragment with the existing buffer.

This operation allocates a new *ByteArray* and copies the entire existing content each time, leading to $O(n^2)$ algorithmic complexity in terms of data copying. An attacker can exploit this by sending a file fragmented into thousands of small TLVs, pinning the CPU and exhausting the heap through repeated large allocations. Furthermore, the *FileSharingManager.readFileWithChecksumStreaming* function utilizes a *ByteArrayOutputStream* that allocates a full-sized buffer in RAM and then performs a redundant copy via *.toByteArray()*, doubling the peak memory footprint during file transmission.

Affected File:
app/src/main/java/com/filtershekanha/noghteha/model/NoghtehaFilePacket.kt

Affected Code:
```
fun decode(data: ByteArray): NoghtehaFilePacket? {
    [...]
    when (t) {
        [...]
        TLVType.CONTENT -> {
            val existing = contentBytes
            if (existing == null) {
                contentBytes = value
            } else {
                contentBytes = existing + value
            }
        }
        [...]
    }
}
```

Affected File:
app/src/main/java/com/filtershekanha/noghteha/model/FileSharingManager.kt

Affected Code:

```
private fun readFileWithChecksumStreaming(file: File): Pair<ByteArray, ByteArray> {
    [...]
    val content = BufferedInputStream(FileInputStream(file), CHUNK_SIZE).use {
bufferedStream ->
        DigestInputStream(bufferedStream, digest).use { digestStream ->
            ByteArrayOutputStream(fileSize.toInt().coerceAtMost(Int.MAX_VALUE)).use {
outputStream ->
                val buffer = ByteArray(CHUNK_SIZE)
                var bytesRead: Int

                while (digestStream.read(buffer).also { bytesRead = it } != -1) {
                    outputStream.write(buffer, 0, bytesRead)
                }

                outputStream.toByteArray()
            }
        }
    }
    [...]
}
```

It is recommended to refactor *NoghtehaFilePacket.decode* to parse TLV headers without immediate copying and to accumulate *CONTENT* fragments in a list so that a single allocation and copy operation is performed only after all fragments are received. It is also recommended to update the file-handling architecture to avoid *ByteArrayOutputStream* usage and redundant in-memory copies by storing a file reference (URI/path) rather than raw bytes, allowing data to be streamed directly from disk to the network socket using a fixed-size bounded buffer. This reduces peak heap usage but requires validation of referenced paths and lifecycle handling for temporary files.

Consolidate all *MAX_FILE_SIZE* constants into a single source of truth in *AppConstants.kt* and enforce this limit strictly at the start of both encoding and decoding paths. In the decoder, maintain a *totalContentLen* counter during TLV iteration to reject the packet immediately if the accumulated length exceeds the protocol maximum, ensuring that heap usage remains bounded and does not trigger an *OutOfMemoryError* (OOM) even when processing large files.

NOG-01-011 WP1: DoS via Unchecked Fragment Allocation *(Medium)*

Retest Notes: Fixed by Noghteha and verified by 7ASecurity.

A DoS vulnerability exists in the *FragmentPayload* class within the decode function. The application allocates memory for incoming fragment payloads based on the size of the received *payloadData* byte array without enforcing the strict protocol-level maximum defined in *AppConstants*. Specifically, the code uses *sliceArray* to create a new copy of the data on the heap.

Although the protocol intends for fragments to be small (*MAX_FRAGMENT_SIZE = 469* bytes), the decoder does not enforce this limit. If the upstream transport layer (e.g., TCP or a bridged connection) allows larger frames, an attacker can send malformed packets utilizing the maximum possible transport size (e.g., 1 MB). This function blindly duplicates the data into a new array. If such frames are received repeatedly, memory pressure is increased for each received frame, accelerating garbage collection (GC) thrashing and potentially degrading service stability on low-memory devices.

Affected File:
app/src/main/java/com/filtershekanha/noghteha/model/FragmentPayload.kt

Affected Code:
```
fun decode(payloadData: ByteArray): FragmentPayload? {
    if (payloadData.size < HEADER_SIZE) {
        return null
    }

    try {
        [...]
        // Extract fragment data (remaining bytes)
        val data = if (payloadData.size > HEADER_SIZE) {
            payloadData.sliceArray(HEADER_SIZE..<payloadData.size)
        } else {
            ByteArray(0)
        }

        return FragmentPayload(fragmentID, index, total, originalType, data)

    } catch (e: Exception) {
        return null
    }
}
```

It is recommended to enforce the existing protocol constants within the decoder by validating *payloadData.size* against a strict maximum before any processing. A maximum accepted size should be defined as *MAX_PACKET_SIZE = HEADER_SIZE +*

AppConstants.Fragmentation.MAX_FRAGMENT_SIZE (approximately 482 bytes), and the function should return *null* immediately if *payloadData.size* exceeds this limit to prevent the *sliceArray* allocation. Additionally, header fields (index, total) should be validated to ensure they fall within sane limits and do not trigger logic errors. The implementation must ensure that *decode()* immediately rejects packets larger than *MAX_PACKET_SIZE* without performing additional allocations.

NOG-01-012 WP1: DoS via Message Queue Flooding *(Medium)*

Retest Notes: Fixed by Noghteha and verified by 7ASecurity.

A DoS vulnerability exists in the *NostrRelayManager* class in the handling of the outgoing message queue that can cause legitimate high-priority messages to be dropped under load. The system enforces a fixed capacity limit on the message queue (*MAX_MESSAGE_QUEUE_SIZE = 1000*) but uses a first-in, first-out (FIFO) eviction policy when full. Specifically, when the queue reaches capacity, the system indiscriminately drops the oldest message (*removeFirst()*) to make room for the new one, without considering message priority or sender fairness.

If untrusted input can trigger outbound event generation, an attacker can exploit this by flooding the relay manager with low-value or "spam" events. This flood rapidly fills the queue, causing the removal of legitimate, high-priority messages (such as handshakes or direct messages) that are waiting to be processed. This allows a single attacker-controlled peer to crowd out legitimate traffic, degrading network reliability and potentially preventing new peers from joining or authenticating.

Affected File:
app/src/main/java/com/filtershekanha/noghteha/nostr/NostrRelayManager.kt

Affected Code:
```
fun sendEvent(event: NostrEvent, relayUrls: List<String>? = null) {
    val targetRelays = relayUrls ?: relaysList.map { it.url }

    // Add to queue for reliability with size limit to prevent OOM
    synchronized(messageQueueLock) {
        if (messageQueue.size >= MAX_MESSAGE_QUEUE_SIZE) {
            // PERFORMANCE: Use removeFirst() for O(1) removal from ArrayDeque
            // Previously removeAt(0) on ArrayList was O(n)
            val dropped = messageQueue.removeFirst()
            Log.w(TAG, "Message queue full (size=$MAX_MESSAGE_QUEUE_SIZE), dropping
oldest event kind=${dropped.first.kind}")
        }
        // PERFORMANCE: addLast() on ArrayDeque is O(1) amortized
        messageQueue.addLast(Pair(event, targetRelays))
        // RELIABILITY: Check and update backpressure state
        updateBackpressureState(messageQueue.size)
    }
    [...]
}
```

It is recommended to implement a “Fair Queuing” or “Quality of Service” (QoS) mechanism by using separate queues for high-priority traffic (handshakes, DMs) and low-priority traffic (gossip), ensuring that critical messages are never dropped in favor of lower-priority traffic. This can reduce throughput for low-priority traffic under load; this impact should be mitigated by applying bounded per-class queue sizes and backpressure. Alternatively, a token bucket limit per peer should be enforced to ensure that no single peer can monopolize shared buffer capacity. The system should ensure that *AUTH* events and direct messages are prioritized over read receipts and other ephemeral traffic.

NOG-01-015 WP1: Unauthenticated Heap Exhaustion in Wi-Fi Aware *(Medium)*

Retest Notes: Fixed by Noghteha and verified by 7ASecurity.

A DoS vulnerability exists in the *WifiAwarePeerConnection* class where packet reading logic allocates memory based on attacker-controlled input prior to authentication. The *readPacket* function reads a 4-byte integer length header and validates it against *MAX_PACKET_SIZE*, which is configured to 1 MB (1024 * 1024). Because this execution path is reached upon socket acceptance in *WifiAwareTransport* (before any peer cryptographic verification), an unauthenticated attacker within Wi-Fi Aware range can open multiple concurrent connections and flood messages declaring a 1 MB length. This can force the application to repeatedly allocate 1 MB blocks on the Java heap, driving aggressive garbage collection (GC) thrashing and potentially causing OOM crashes or an unresponsive background service.

Affected File:
app/src/main/java/com/filtershekanha/noghteha/wifi/WifiAwarePeerConnection.kt

Affected Code:

```
private const val MAX_PACKET_SIZE = 1024 * 1024 // 1 MB max
[...]
private fun readPacket(): NoghtehaPacket? {
    val input = inputStream ?: return null

    // Read 4-byte length header
    val length = input.readInt()

    // Validate length
    if (length < 0 || length > MAX_PACKET_SIZE) {
        Log.w(TAG, "Invalid packet length from $peerID: $length")
```

```
            return null
        }
        [...]
        // Read packet data
        val data = ByteArray(length)
        input.readFully(data)
        [...]
    }
```

It is recommended to separate unauthenticated framing from authenticated payload processing by applying the following controls:

- Avoid allocating a full-sized buffer immediately based on the length header by using a small fixed-size buffer (e.g., 8 KB) or a *BufferedInputStream* to identify message type.
- Defer full payload allocation (up to 1 MB) until sender identity is cryptographically verified, or until message type is confirmed as a handshake packet with a strict size limit (e.g., <1 KB).
- Implement connection-level rate limiting to drop peers that send excessive data prior to successful authentication.

NOG-01-017 WP1: Passive Device Targeting via Hardcoded BLE Secrets *(High)*

Retest Notes: Fixed by Noghteha and verified by 7ASecurity.

A privacy issue exists in the *ServiceUuidRotation* mechanism where the application attempts to prevent passive user tracking by rotating the Bluetooth Low Energy (BLE) Service UUID. Although “Device Jitter” and “Version Entropy” are used to desynchronize devices and separate app versions, the entire cryptographic derivation chain relies on a set of globally static constants, anchored by *BASE_ROTATION_SECRET*, that are hardcoded in the application binary and shared across installations.

Because the effective rotation key is derived entirely from static constants embedded in the application binary, an adversary can extract them once and precompute the valid UUID sequence for a given time window. Even if the "Device Jitter" mechanism shifts the specific bucket index used by a device, the resulting UUID is merely shifted to an adjacent value in this globally predictable sequence. Consequently, a passive adversary can positively identify an active device, regardless of its specific jitter offset, by monitoring the small set of valid UUIDs covering the current and adjacent time intervals.

Affected File:
app/src/main/java/com/filtershekanha/noghteha/mesh/ServiceUuidRotation.kt

Affected Code:
```
object ServiceUuidRotation {
    [...]
    // Prefix for HMAC input to domain-separate from other uses
    private const val HMAC_PREFIX = "noghteha-ble-uuid-v2-"
    [...]
    private val BASE_ROTATION_SECRET = byteArrayOf(
        0x4E.toByte(), 0x6F.toByte(), 0x67.toByte(), 0x68.toByte(), // "Nogh"
        0x74.toByte(), 0x65.toByte(), 0x68.toByte(), 0x61.toByte(), // "teha"
        0x2D.toByte(), 0x42.toByte(), 0x4C.toByte(), 0x45.toByte(), // "-BLE"
        0x2D.toByte(), 0x52.toByte(), 0x6F.toByte(), 0x74.toByte(), // "-Rot"
        0x61.toByte(), 0x74.toByte(), 0x69.toByte(), 0x6F.toByte(), // "atio"
        0x6E.toByte(), 0x2D.toByte(), 0x53.toByte(), 0x65.toByte(), // "n-Se"
        0x63.toByte(), 0x72.toByte(), 0x65.toByte(), 0x74.toByte(), // "cret"
        0x2D.toByte(), 0x56.toByte(), 0x32.toByte(), 0x00.toByte()  // "-V2\0"
    )

    private const val PROTOCOL_VERSION = "2.0.0"

    private val VERSION_ENTROPY: ByteArray by lazy {
        try {
            [...]
            val versionInfo = buildString {
                append("noghteha-version-entropy-")
                [...]
                append(PROTOCOL_VERSION)
            }
            MessageDigest.getInstance("SHA-256").digest(versionInfo.toByteArray(
Charsets.UTF_8))
        }
        [...]
    }

    private val EFFECTIVE_ROTATION_SECRET: ByteArray by lazy {
        try {
            val mac = Mac.getInstance(HMAC_ALGORITHM)
            val keySpec = SecretKeySpec(BASE_ROTATION_SECRET, HMAC_ALGORITHM)
            mac.init(keySpec)
            [...]
            // Add version-specific entropy
            mac.update(VERSION_ENTROPY)
            [...]
        }
        [...]
    }
    [...]
    private fun deriveUuidForBucket(bucketIndex: Long): UUID {
        return try {
            val mac = Mac.getInstance(HMAC_ALGORITHM)
            val keySpec = SecretKeySpec(EFFECTIVE_ROTATION_SECRET, HMAC_ALGORITHM)
            mac.init(keySpec)

            // Input: prefix + bucket index as string
```

```
                val input = "$HMAC_PREFIX$bucketIndex".toByteArray(Charsets.UTF_8)
                val hash = mac.doFinal(input)
                [...]
            }
            [...]
        }
    }
```

It is recommended to prioritize user control and reduced exposure by providing a “Stealth Mode” that disables global advertising in high-risk environments and limits discovery to trusted contacts using pairwise keys. This reduces the ability of a global passive adversary to monitor broadcasts for identification purposes, but it also reduces ad hoc discovery; this trade-off should be addressed by making the mode user-controlled and clearly communicating the impact on discoverability.

Where a trusted update channel is available, it is recommended to provision the rotation secret outside the application binary and support secret rotation and revocation to limit long-term predictability if a secret is extracted. This introduces operational complexity and a risk of discovery disruption during key changes; the impact should be reduced by versioning secrets and using an overlap window to allow old and new values to coexist during rollout.

Hardening Recommendations

This area of the report provides insight into less significant weaknesses that might assist adversaries in certain situations. Issues listed in this section often require another vulnerability to be exploited, need an uncommon level of access, exhibit minor risk potential on their own, and/or fail to follow information security best practices. Nevertheless, it is recommended to resolve as many of these items as possible to improve the overall security posture and protect users in edge-case scenarios.

NOG-01-001 WP1: Android: Missing Root Detection *(Info)*

Retest Notes: Fixed by Noghteha and verified by 7ASecurity.

The Android app lacks root detection, failing to alert users about security risks[4]. This can be confirmed by installing the app on a rooted device and verifying the absence of warnings.

It is recommended to implement root detection to address this issue. Since the user has root access while the app does not, detection mechanisms are inherently bypassable with sufficient skill. The RootBeer library[5] can be used to warn users about the risks of running the app on a rooted device, which, despite being bypassable, serves as an effective alert.

NOG-01-002 WP1: Android Binary Hardening Recommendations *(Info)*

It was found that a number of binaries embedded into the Android application are currently not leveraging the available compiler flags to mitigate potential memory corruption vulnerabilities. This unnecessarily puts the application more at risk for such issues.

Issue 1: Binaries missing usage of -D_FORTIFY_SOURCE=2

Missing this flag means common *libc* functions are missing buffer overflow checks, so the application is more prone to memory corruption vulnerabilities. Please note that most binaries are affected, the following is a reduced list of examples for the sake of brevity.

Example binaries (from decompiled production app):
x86/libandroidx.graphics.path.so
x86_64/libandroidx.graphics.path.so

[4] https://www.bankinfosecurity.com/jailbreaking-ios-devices-risks-to-users-enterprises-a-8515
[5] https://github.com/scottyab/rootbeer

armeabi-v7a/libandroidx.graphics.path.so
arm64-v8a/libandroidx.graphics.path.so
[...]

It is recommended to compile all binaries using the *-D_FORTIFY_SOURCE=2* argument so that common insecure *libc* functions like *memcpy*, etc. are automatically protected with buffer overflow checks.

Issue 2: Binaries missing usage of Stack Canary

Some binaries do not have a stack canary value added to the stack. Stack canaries are used to detect and prevent exploits from overwriting return addresses.

Example binaries (from decompiled app):
x86/libarti_mobile_ex.so
x86_64/libarti_mobile_ex.so
armeabi-v7a/libarti_mobile_ex.so
arm64-v8a/libarti_mobile_ex.so
[...]

It is recommended to enable stack canary protections across native builds using *-fstack-protector-strong* (or *-fstack-protector-all* where the additional overhead is acceptable) to improve resilience against stack-based exploitation.

NOG-01-003 WP1: Unmaintained Android version support via minSDK level *(Info)*

A security hardening opportunity was identified in the Android platform support policy. The Android manifest sets *minSdkVersion* to 24 (Android 7.0), which allows the application to run on Android releases that are out of security support and may no longer receive OS security patches. Public support schedules indicate that Android 7.0 is end of security support. They also indicate that Android 10 (API 29) and Android 12 (API 31) have reached the end of security support on some support schedules.

Supporting end-of-life Android releases increases exposure to known vulnerabilities that may remain unpatched on those devices, including kernel privilege escalation vulnerabilities such as CVE-2019-22151[6] and task-hijacking issues such as StrandHogg 2.0 (CVE-2020-0096[7]).

Affected file:
AndroidManifest.xml

[6] https://nvd.nist.gov/vuln/detail/cve-2019-2215
[7] https://nvd.nist.gov/vuln/detail/cve-2020-0096

Affected code:
```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
   android:versionCode="35"
   android:versionName="1.0.34"
   android:compileSdkVersion="35"
   android:compileSdkVersionCodename="15"
   package="com.filtershekanha.noghteha"
   platformBuildVersionCode="35"
   platformBuildVersionName="15">
   <uses-sdk
       android:minSdkVersion="24"
       android:targetSdkVersion="35"/>
```

It is recommended to raise *minSdkVersion* to a currently supported baseline (for example, Android 13 / API 33 or later), aligned with the product device support requirements, to reduce reliance on end-of-life Android versions.

NOG-01-009 WP1: WebView JS Bridge Hardening *(Low)*

Retest Notes: Fixed by Noghteha and verified by 7ASecurity.

A security hardening opportunity was identified in *GeohashPickerActivity*. JavaScript is enabled in a WebView, and a native interface is exposed via *addJavascriptInterface*. Even if the intended content is a local asset, the presence of a JavaScript bridge increases impact if unintended JavaScript execution becomes possible in this WebView (for example, unexpected navigation, future WebView issues, or asset tampering), because bridge methods can be invoked by script within the WebView context. The exposure surface can be reduced by strictly constraining the WebView to the single expected *file:///android_asset/...* page and disabling file/universal access modes that broaden what the WebView can reach.

Affected File:
app/src/main/java/com/filtershekanha/noghteha/ui/GeohashPickerActivity.kt

Affected Code:
```
@SuppressLint("SetJavaScriptEnabled")
override fun onCreate(savedInstanceState: Bundle?) {
    [...]
    settings.javaScriptEnabled = true
    [...]
    addJavascriptInterface(object {
        @JavascriptInterface
        fun onGeohashChanged(geohash: String) {
            val sanitized = geohash.trim().lowercase()
            if (!GEOHASH_PATTERN.matches(sanitized)) {
                return
            }
            [...]
        }
    }, "Android")

    loadUrl("file:///android_asset/geohash_picker.html")
    [...]
}
```

It is recommended to keep the WebView scoped to the single expected asset page by blocking all navigations except *file:///android_asset/geohash_picker.html* and denying non-asset schemes (for example, *http:*, *https:*, and *intent:*). To reduce the blast radius of any unintended JavaScript execution, *allowFileAccessFromFileURLs* and *allowUniversalAccessFromFileURLs* should be disabled (and other WebView settings should be kept at the minimum required for functionality). The WebView should be verified to be unable to navigate to external destinations.

NOG-01-013 WP1: Privacy Gap via Unconsented Third-Party Geocoding *(Low)*

Retest Notes: Fixed by Noghteha and verified by 7ASecurity.

A privacy hardening opportunity was identified in the application location handling logic within *LocationChannelManager* and *GeohashBookmarksStore*. Reverse geocoding is performed automatically via *android.location.Geocoder* when the user location is updated and when a new geohash bookmark is added. On many Android devices, Geocoder can be serviced by a network-backed provider (often Google Play services on Google-enabled devices), which may transmit precise coordinates off-device to the configured geocoding provider. For a censorship-resistant messenger intended for high-risk environments, automatic reverse geocoding during routine operation can create avoidable metadata by generating repeated, timestamped location lookups without explicit user intent or consent.

Affected File:
app/src/main/java/com/filtershekanha/noghteha/geohash/LocationChannelManager.kt

Affected Code:
```
private fun reverseGeocodeIfNeeded(location: Location) {
    if (!Geocoder.isPresent()) {
        Log.w(TAG, "Geocoder not present on this device")
        return
    }
    [...]
    scope.launch {
        try {
            // PRIVACY NOTE: This sends coordinates to Google servers for reverse
geocoding
            Log.d(TAG, "Starting reverse geocoding (sends coords to Google)")

            @Suppress("DEPRECATION")
            val addresses = geocoder.getFromLocation(location.latitude,
location.longitude, 1)

            if (!addresses.isNullOrEmpty()) {
                [...]
            }
            [...]
        } catch (e: Exception) {
            Log.e(TAG, "Reverse geocoding failed: ${e.message}")
        }
        [...]
    }
}
```

Affected File:
app/src/main/java/com/filtershekanha/noghteha/geohash/GeohashBookmarksStore.kt

Affected Code:
```
fun resolveNameIfNeeded(geohash: String) {
    [...]
    if (!Geocoder.isPresent()) return

    resolving.add(gh)
    scope.launch {
        try {
            val geocoder = Geocoder(context, Locale.getDefault())
            val name: String? = if (gh.length <= 2) {
                [...]
                for (loc in points) {
                    try {
                        @Suppress("DEPRECATION")
                        val list = geocoder.getFromLocation(loc.latitude,
loc.longitude, 1)
                        [...]
                    }
                    [...]
                }
                [...]
            } else {
                val center = Geohash.decodeToCenter(gh)
                @Suppress("DEPRECATION")
                val list = geocoder.getFromLocation(center.first, center.second, 1)
                [...]
            }
            [...]
        } catch (e: Exception) {
            SecureLog.w(TAG, "Name resolution failed for geohash: ${e.message}")
        } finally {
            resolving.remove(gh)
        }
    }
}
```

It is recommended to gate reverse geocoding behind an explicit opt-in setting and default to no geocoding requests on initial install. When enabled, a clear notice should be shown stating that coordinates may be sent to the device geocoding provider (for example, Google on GMS devices). When disabled (or when Geocoder is unavailable), the application should fall back to displaying raw Geohash coordinates or use an offline reverse-geocoding library (e.g., one based on OpenStreetMap data) if feature parity is required. Both live location updates and bookmark name resolution should function without network geocoding when the setting is disabled.

NOG-01-014 WP1: Weaknesses in Intent Token Validation *(Low)*

Retest Notes: Fixed by Noghteha and verified by 7ASecurity.

A security hardening opportunity was identified in the notification intent validation mechanism. The *validateIntent* function compares the provided token to the expected HMAC using standard string comparison, which may introduce timing side-channel differences during token verification. The function also does not implement replay protection, so a captured legitimate token could be reused within the 15-minute validity window. While timestamps and HMAC signatures are validated, this can increase the risk of duplicate notification triggers or repeated actions if a valid token is obtained.

Affected File:
src/main/java/com/filtershekanha/noghteha/ui/NotificationManager.kt

Affected Code:
```
fun validateIntent(intent: Intent?, expectedIntentData: String = ""): Boolean {
   if (intent == null) return false

   // Check for token
   val token = intent.getStringExtra(EXTRA_NOTIFICATION_TOKEN) ?: return false

   // Check for timestamp
   val timestamp = intent.getLongExtra(EXTRA_TOKEN_TIMESTAMP, 0L)
   if (timestamp == 0L) return false

   // SECURITY FIX L3: Validate timestamp is within validity window
   val currentTime = System.currentTimeMillis()
   val tokenAge = currentTime - timestamp
   if (tokenAge < 0 || tokenAge > TOKEN_VALIDITY_WINDOW_MS) {
       Log.w(TAG, "Notification token expired (age: ${tokenAge}ms)")
       return false
   }

   // SECURITY FIX L3: Validate HMAC
   val expectedToken = generateSecurityToken(timestamp, expectedIntentData)
   if (token != expectedToken) { // 7asec comment: non constant-time comparison
       Log.w(TAG, "Notification token HMAC mismatch")
       return false
   }

   return true // 7asec comment: No replay protection
}
```

It is recommended to implement constant-time comparison for HMAC validation and add replay protection through token tracking. The string comparison should be replaced with a constant-time byte comparison (for example, *MessageDigest.isEqual()*[8] on decoded HMAC bytes). To reduce replay risk, a synchronized in-memory set can be used to track a composite key (timestamp, intent data, token); if the composite key has already been seen, the intent should be rejected, and successful validations should be recorded after HMAC verification. Periodic cleanup should be implemented to remove entries older than the validity window or cap the set size (for example, 1000 tokens). If replay protection must remain optional, at minimum the constant-time comparison should be implemented to reduce timing side-channel risk.

NOG-01-016 WP1: Weaknesses in DeepLink URL Validation *(Info)*

Retest Notes: Fixed by Noghteha and verified by 7ASecurity.

A security hardening opportunity was identified in the deep link sanitization mechanism. The current function relies on character blacklisting to remove a small set of characters from deep link URLs (for example, <, >, ", and '), but this approach is not sufficient for URL validation and can be bypassed through URL encoding, Unicode alternatives, or protocol-based payloads. Several characters and URL features that frequently matter for deep link handling are not addressed (for example, %, #, and ;). More importantly, URL components such as scheme, host, and path are not validated, so if this function is later used as a security control, unsafe schemes (for example, *javascript:*, *file:*, *data:*, or *intent:*), open redirects, path traversal sequences (..), or unintended intent resolution could be allowed depending on downstream usage. This may increase the risk of phishing, local resource exposure, or unintended navigation when an attacker-controlled URL is processed.

[8] https://docs.oracle.com/javase/8/docs/api/java/security/MessageDigest.html

This finding is cataloged as informative because the function is not currently in use but may be utilized in the future.

Affected File:
src/main/java/com/filtershekanha/noghteha/navigation/NavRoutes.kt

Affected Code:
```
// Characters that could be used for injection attacks
private val DANGEROUS_CHARS = setOf('<', '>', '"', '\'', '\\', '\n', '\r',
'\t', '\u0000')
[...]
fun sanitizeString(input: String?, maxLength: Int = 256): String? {
   if (input.isNullOrBlank()) return null

   return input
       .trim()
       .filter { it !in DANGEROUS_CHARS }
       .take(maxLength)
       .ifBlank { null }
}
```

It is recommended to replace character blacklisting with allowlist-based URL validation using Android URI parsing. Explicit allowlists should be defined for permitted schemes (for example, https and approved app schemes) and permitted hostnames for HTTPS URLs. The URL should be parsed using *Uri.parse()* with malformed inputs rejected, the scheme should be validated using case-insensitive comparison, and HTTPS hostnames should be checked against an allowed domain list. Inputs containing path traversal sequences (..) should be rejected, and a length limit (for example, 2048 characters) should be enforced to reduce DoS risk. Rejected URLs can be logged for monitoring (in a privacy-safe manner). This approach ensures only explicitly permitted URL patterns are accepted and reduces bypass techniques.

NOG-01-018 WP1: Weaknesses in Android Keystore Configuration *(Medium)*

Retest Notes: Fixed by Noghteha and verified by 7ASecurity.

A security hardening opportunity was identified in the Android Keystore implementation used for encryption key management. The current configuration does not request StrongBox-backed storage and does not bind key use to user authentication, which weakens protection in device-seizure and coerced-unlock scenarios. StrongBox-backed key storage is not requested, so keys can be generated outside StrongBox even on devices that support it. User authentication is not bound to cryptographic operations, so key use is not gated by biometric or device-credential verification, and no authentication validity window is enforced. If *setUnlockedDeviceRequired(true)* is used, it only restricts key use while the device is locked and does not enforce per-operation user authentication. As a result, the risk of data exposure is increased when an attacker obtains access to an unlocked device or coerces device unlock.

Affected File:
app/src/main/java/com/filtershekanha/noghteha/db/NoghtehaDatabase.kt

Affected Code:
```
private fun createMasterKey() {
   val keyGenerator = KeyGenerator.getInstance(
       KeyProperties.KEY_ALGORITHM_AES,
       ANDROID_KEYSTORE
   )

   val builder = KeyGenParameterSpec.Builder(
       KEYSTORE_ALIAS,
       KeyProperties.PURPOSE_ENCRYPT or KeyProperties.PURPOSE_DECRYPT
   )
       .setBlockModes(KeyProperties.BLOCK_MODE_GCM)
       .setEncryptionPaddings(KeyProperties.ENCRYPTION_PADDING_NONE)
       .setKeySize(256)
       .setRandomizedEncryptionRequired(true)
       // 7asec comments
       // Missing: User authentication requirement
       // Missing: Biometric enrollment invalidation
       // Missing: StrongBox hardware backing
       // Missing: Authentication timeout
```

It is recommended to configure Android Keystore to prefer hardware-backed storage, require per-operation authentication, and invalidate keys on biometric enrollment changes. Where available, StrongBox can be requested via *setIsStrongBoxBacked(true)*, with a fallback key configuration when StrongBox is unavailable. On Android 11+ (API 30+), *setUserAuthenticationParameters(30, AUTH_BIOMETRIC_STRONG | AUTH_DEVICE_CREDENTIAL)* can be used to enforce a 30-second authentication validity window, and *setInvalidatedByBiometricEnrollment(true)* can be used to invalidate keys on biometric enrollment changes. *UserNotAuthenticatedException* can be handled by prompting biometric or device-credential authentication via BiometricPrompt with a *CryptoObject. setUnlockedDeviceRequired(true)* can be used as a supplemental control to prevent key use while the device is locked, but it should not be treated as a substitute for per-operation authentication.

WP2: Noghteha Privacy Audit

This section presents the analysis results addressing ten privacy-related questions. For this portion of the engagement, 7ASecurity utilizes the following classification to specify the certainty level of findings. As the research is based on documentation, source code, and sample configuration analysis, classification is necessary to indicate the confidence level of each discovery:

- *Proven*: Source code and runtime activity clearly confirm the finding as fact
- *Evident*: Source code strongly suggests a privacy concern, but this could not be proven at runtime
- *Assumed*: Indications of a potential privacy concern were found but a broader context remains unknown.
- *Unclear*: Initial suspicion was not confirmed. No privacy concern can be assumed.

Each ticket summarizes the 7ASecurity attempts to answer relevant questions cited at the beginning of each section.

NOG-01-Q01: Files & Information Gathered by Noghteha *(Proven)*

Q1: What files/information are gathered by the Noghteha app?

MITRE ATT&CK framework[9] mapping:

- T1005 Data from Local System[10]
- T1517 Access Notifications[11]

Based on privacy documentation and source code review, the following information is gathered by the Noghteha application:

Data collected and stored locally:

1. **Cryptographic identity keys:** Encryption key pairs are generated and stored on-device and are not transmitted to servers.
2. **Messages**: End-to-end encrypted message content is stored locally and is deleted after reading by default.
3. **Shared media files**: Images, voice notes, and documents shared in conversations are stored in application-private encrypted directories.
4. **Location data (opt-in only)**: When location-based channels are enabled, the following are used:
   - Geohash approximations (privacy-preserving grid cells)

[9] https://attack.mitre.org
[10] https://attack.mitre.org/techniques/T1005/
[11] https://attack.mitre.org/techniques/T1517/

- ○ Reverse-geocoded location names (city, region, country)

Minimal data collection is consistent with the privacy-focused design (no phone numbers, email addresses, contact lists, or message metadata). However, a critical privacy disclosure gap exists: precise GPS coordinates are transmitted to Google for reverse geocoding (NOG-01-013), despite privacy policy claims to the contrary. It is recommended to update documentation and implement explicit user consent mechanisms for third-party location data sharing.

NOG-01-Q02: Where & How Noghteha Transmits Data *(Unclear)*

Q2: Where and how are the files/information gathered transmitted?

MITRE ATT&CK framework[12] mapping:

- T1041 Exfiltration Over C2 Channel[13]
- T1048 Exfiltration Over Alternative Protocol[14]
- T1071 Application Layer Protocol[15]

Based on documentation and source code review, the Noghteha mobile application appears to transmit data through multiple decentralized channels designed for privacy preservation. Messages are routed peer-to-peer via Bluetooth Low Energy (BLE) and Wi-Fi Direct mesh networking, with end-to-end encryption using the Noise protocol. When Internet connectivity is available, Tor (via the Arti library) may be used for anonymous routing, and the Nostr decentralized relay network may be used for extended message delivery. Message content remains encrypted during transmission.

When the user opts into location-based channels, location data is converted to geohash approximations (privacy-preserving grid cells) before being transmitted to peers via the mesh network. A critical discrepancy is indicated by the reverse-geocoding implementation: precise GPS coordinates (latitude and longitude) appear to be transmitted to Google geocoding servers via HTTPS to obtain human-readable location names. This appears to occur without explicit user disclosure or consent and contradicts the privacy policy statement that *"your exact position will never be saved or sent"*[16].

User encryption keys are generated and stored locally in the Android Keystore and are not transmitted to servers. No centralized servers are maintained for message storage; messages are transmitted directly between devices or through decentralized Tor/Nostr networks.

12 https://attack.mitre.org
13 https://attack.mitre.org/techniques/T1041/
14 https://attack.mitre.org/techniques/T1048/
15 https://attack.mitre.org/techniques/T1071/
16 https://noghteha.app/en/privacy

Media files (images, voice notes, documents) shared through the application are stored in application-private encrypted directories and are transmitted peer-to-peer with end-to-end encryption when shared in conversations.

Network metadata that may be observable during transmission includes:

- **Mesh network:** Bluetooth MAC addresses, Wi-Fi SSID/BSSID, timing patterns, peer proximity information
- **Tor network:** Entry guard may observe user IP (not destination); exit node may observe destination (not source)
- **Nostr relays:** Public keys (pseudonymous identifiers), connection timestamps, subscription patterns, IP addresses (unless routed via Tor)
- **Google geocoding:** Precise GPS coordinates, device IP address, request timestamps, device fingerprinting data (User-Agent, platform details)

The ProGuard configuration indicates that logging is stripped from release builds to reduce information exposure through logcat. An emergency "Panic Mode" feature is included that irreversibly deletes encryption keys, messages, and settings when activated.

In summary, data transmission occurs through the following channels:

- **Message content:** Peer-to-peer mesh (BLE/WiFi), Tor network, Nostr relays—all with Noise protocol end-to-end encryption
- **Geohash approximations:** Transmitted to peers via mesh/Tor/Nostr for location-based channel discovery (opt-in)
- **Precise GPS coordinates:** Appear to be transmitted to Google geocoding servers via HTTPS (undisclosed, requires user consent)
- **Media files:** Peer-to-peer transmission with end-to-end encryption when shared
- **Encryption keys:** Never transmitted—stored locally in Android Keystore only

Overall, the transmission architecture follows privacy-preserving principles with strong encryption and decentralized routing, except for the undisclosed Google geocoding integration which requires immediate remediation to align with privacy policy claims and regulatory compliance requirements.

NOG-01-Q03: Insecure PII Storage Analysis of Noghteha *(Unclear)*

Q3: Is sensitive PII such as audio, pictures or data insecurely stored or easily retrievable from the Noghteha app?

MITRE ATT&CK framework[17] mapping:

- T1552 Unsecured Credentials[18]
- T1005 Data from Local System[19]
- T1074 Data Staged[20]

No evidence was identified to suggest that sensitive Personally Identifiable Information (PII) is stored insecurely or is readily retrievable from the Noghteha application. A security-focused architecture and encryption-at-rest controls are implemented (including SQLCipher, Android Keystore, and application-private storage). Data is not easily retrievable without the required device unlock credentials. Defense-in-depth could be improved by requiring per-operation authentication for Keystore usage to better address scenarios involving coerced device unlocks. The current security posture is stronger than typical Android messaging applications and is aligned with the stated threat model (activists operating in surveillance environments).

NOG-01-Q04: Analysis of Potential Noghteha User Tracking *(Unclear)*

Q4: Does the Noghteha app implement any sort of user tracking function via location or other means?

MITRE ATT&CK framework[21] mapping:

- T1082 System Information Discovery[22]
- T1016 System Network Configuration Discovery[23]

The Noghteha Android application requests location permissions for geohash-based chat channel discovery. Opt-in location functionality is implemented to allow location-based public chat rooms to be joined using geohash technology.

Affected File:

AndroidManifest.xml

Affected Code:

```
<uses-permission android:name="android.permission.ACCESS_COARSE_LOCATION" />
```

[17] https://attack.mitre.org
[18] https://attack.mitre.org/techniques/T1552/
[19] https://attack.mitre.org/techniques/T1005/
[20] https://attack.mitre.org/techniques/T1074/
[21] https://attack.mitre.org
[22] https://attack.mitre.org/techniques/T1082/
[23] https://attack.mitre.org/techniques/T1016/

```
<uses-permission android:name="android.permission.ACCESS_FINE_LOCATION" />
```

Based on source code and documentation review, the following characteristics were identified:

- **Core functionality:** Location-based chat channel discovery is implemented through geohash grid cells.
- **Privacy protocol:** Coordinates are converted into geohash identifiers rather than being transmitted as raw latitude/longitude values.
- **User configuration:** The feature is opt-in and can be disabled in application settings.
- **Data transmission policy (documented):** Transmission of raw location coordinates to external servers is stated to be prohibited.

Although location functionality is present, no evidence was identified to suggest that it is used for user tracking rather than application functionality.

NOG-01-Q05: Potential Noghteha Crypto Weakening *(Unclear)*

Q5: Does the Noghteha app intentionally weaken cryptographic procedures to ensure third-party decryption?

MITRE ATT&CK framework[24] mapping:

- T1600 Weaken Encryption[25]

Several minor cryptographic weaknesses were identified during the assessment, as documented in NOG-01-008 and NOG-01-013. These weaknesses did not appear to have been intentionally introduced to enable third-party decryption.

[24] https://attack.mitre.org
[25] https://attack.mitre.org/techniques/T1600/

NOG-01-Q06: Insecure SD Card Usage by Noghteha *(Unclear)*

Q6: Is data dumped in the SD Card from where it could be retrieved later without even entering the PIN to unlock the device?
MITRE ATT&CK framework[26] mapping:

- T1005 Data from Local System[27]
- T1074 Data Staged[28]
- T1407 Access Sensitive Data in Device Storage[29]

No evidence was identified to suggest that application data is written to the SD card. Sensitive content (messages, voice notes, image files, encryption keys) is stored in application-private internal storage, which requires device unlocking and is protected by Android full-disk encryption. The *READ_EXTERNAL_STORAGE[30]* permission is used solely to allow users to select media files to share and is not used to write application data externally.

No additional action is required regarding external storage security.

NOG-01-Q07: Potential for RCE in Noghteha *(Unclear)*

Q7: Does the Noghteha app contain vulnerabilities or shell commands that could lead to RCE in any way?
MITRE ATT&CK framework[31] mapping:

- T1203 Exploitation for Client Execution[32]
- T1059 Command and Scripting Interpreter[33]
- T1068 Exploitation for Privilege Escalation[34]

No vulnerabilities that could lead to direct or indirect remote code execution (RCE) were identified during this engagement.

[26] https://attack.mitre.org
[27] https://attack.mitre.org/techniques/T1005/
[28] https://attack.mitre.org/techniques/T1074/
[29] https://attack.mitre.org/techniques/T1407/
[30] https://developer.android.com/[...]/Manifest.permission#READ_EXTERNAL_STORAGE
[31] https://attack.mitre.org
[32] https://attack.mitre.org/techniques/T1203/
[33] https://attack.mitre.org/techniques/T1059/
[34] https://attack.mitre.org/techniques/T1068/

NOG-01-Q08: Potential Noghteha Backdoors *(Unclear)*

Q8: Does the Noghteha app have any kind of backdoor?
MITRE ATT&CK framework[35] mapping:

- T1055 Process Injection[36]
- T1505 Server Software Component[37]
- T1556 Modify Authentication Process[38]

No indications of process or command execution calls typically associated with backdoors or malware were identified. Based on this assessment, no action is required to improve the privacy posture in this regard.

NOG-01-Q09: Noghteha Attempts to Gain Root Access *(Unclear)*

Q9: Does the Noghteha app attempt to gain root access through public Android vulnerabilities or in other ways?
MITRE ATT&CK framework[39] mapping:

- T1068 Exploitation for Privilege Escalation[40]
- T1548 Abuse Elevation Control Mechanism[41]
- T1404 Exploit OS Vulnerability[42]

No evidence was identified to suggest that Noghteha client components (Android library and C++ client) attempt to exploit platform-specific vulnerabilities to obtain elevated privileges. Based on this assessment, no action is required to improve the privacy posture in this regard.

[35] https://attack.mitre.org
[36] https://attack.mitre.org/techniques/T1055/
[37] https://attack.mitre.org/techniques/T1505/
[38] https://attack.mitre.org/techniques/T1556/
[39] https://attack.mitre.org
[40] https://attack.mitre.org/techniques/T1068/
[41] https://attack.mitre.org/techniques/T1548/
[42] https://attack.mitre.org/techniques/T1404/

NOG-01-Q10: Potential Noghteha Usage of Obfuscation *(Proven)*

Q10: Does the Noghteha app use obfuscation techniques to hide code and if yes for which files and directories?

MITRE ATT&CK framework[43] mapping:

- T1027 Obfuscated Files or Information[44]
- T1406 Obfuscated Files or Information[45]

Evidence of obfuscation was identified in the Noghteha codebase. ProGuard/R8[46] obfuscation is used for release builds as a security hardening measure, given the threat model involving device seizure scenarios.

Obfuscation applies to the Android application by default, with exceptions defined through *-keep* rules. The following components remain unobfuscated to preserve functionality:

- Protocol and cryptography classes (*com.filtershekanha.noghteha.protocol.***, *com.filtershekanha.noghteha.crypto.***)
- BouncyCastle cryptography library
- Identity management classes (*com.filtershekanha.noghteha.identity.***)
- Nostr protocol implementation (*com.filtershekanha.noghteha.nostr.***)
- Mesh networking packages (*com.filtershekanha.noghteha.mesh.***, *com.filtershekanha.noghteha.wifi.***, *com.filtershekanha.noghteha.transport.***)
- Database entities and DAOs (*com.filtershekanha.noghteha.db.***)
- Dependency injection annotations (Hilt/Dagger)
- Native library bindings (Arti/Tor, SQLCipher)

All other application code is obfuscated, including method names, field names, and internal implementation details. Logging statements are stripped from release builds to reduce information exposure through logcat.

43 https://attack.mitre.org
44 https://attack.mitre.org/techniques/T1027/
45 https://attack.mitre.org/techniques/T1406/
46 https://github.com/Guardsquare/proguard

Conclusion

Despite the number of findings identified during this assessment, the Noghteha solution defended itself well against a broad range of attack vectors and demonstrated a clear security and privacy-focused architectural intent. The platform is expected to become increasingly difficult to attack as additional cycles of security testing and subsequent hardening continue.

The Noghteha application provided a number of positive impressions during this assignment that must be mentioned here:

- The Noise Protocol Framework (XX pattern) was selected for encrypted transport, providing a strong foundation for forward secrecy and mutual authentication.
- Kotlin is used extensively, which reduces exposure to classic native memory corruption issues common in C/C++ mesh stacks.
- The architecture reflects a privacy-first intent, including localized mesh networking and BLE UUID rotation, even though specific details require hardening (NOG-01-017).
- The source code is well documented, which materially improves reviewability and reduces implementation ambiguity.
- Sensitive key material is explicitly zeroed in multiple places (for example, *CryptoUtils.secureZero* and *java.util.Arrays.fill()*), indicating strong engineering hygiene.
- Numerous defensive checks and hardening strategies are already present throughout the codebase, reducing the likelihood of trivial implementation flaws.
- Release build hardening controls (for example, obfuscation and log reduction) are present, which increases attacker cost during reverse engineering and seizure scenarios.

The security of the Noghteha solution will improve with a focus on the following areas:

- **DoS Resistance:** Eliminate algorithmic and allocation-driven DoS vectors by removing iterative byte-array concatenation and enforcing strict size limits before allocation or decoding (NOG-01-010, NOG-01-011, NOG-01-015).
- **Resource Bounding:** Apply fail-fast validation before allocating memory, adding map entries, or launching coroutines; enforce caps for concurrent handshakes and outbound buffering, and prevent a single peer from monopolizing queues under flood conditions (NOG-01-006, NOG-01-012, NOG-01-015).
- **Harden Release Builds:** Strip or make unreachable debug/administrative functionality in production builds, and ensure any required field diagnostics are explicitly gated and cannot persist unsafe configurations (NOG-01-007).

- **Eliminate Globally Hardcoded Secrets:** Remove static BLE rotation secrets embedded in the binary to reduce long-term predictability and passive device targeting risk (NOG-01-017).
- **Keystore Hardening:** Prefer hardware-backed key storage where available and bind key use to user authentication with an appropriate validity window to strengthen seizure and coerced-unlock resilience (NOG-01-018).
- **Modernize Cryptographic Primitives:** Replace PBKDF2 with predictable salts with a memory-hard KDF (for example, *Argon2id*) and versioning to reduce GPU-accelerated precomputation and offline guessing feasibility for common channel names (NOG-01-008).
- **Intent Validation:** Strengthen intent token handling with constant-time comparison and replay resistance to reduce spoofing and reuse risk (NOG-01-014).
- **DeepLink Validation:** Replace character filtering with allowlist-based parsing and strict scheme/host validation to reduce injection and phishing risk in downstream usage (NOG-01-016).
- **Platform Baseline:** Raise the minimum supported Android baseline to reduce reliance on end-of-life OS versions and legacy security limitations (NOG-01-003).
- **Binary Hardening:** Enable standard native hardening controls (stack canaries, FORTIFY, RELRO/PIE as applicable) across shipped libraries to improve exploitation resistance (NOG-01-002).
- **Re-auth Controls:** Require biometric or device-credential re-authentication for sensitive operations after inactivity to reduce exposure on seized or shared devices (NOG-01-004).
- **Memory Residue:** Reduce plaintext chat persistence in memory where feasible and ensure sensitive caches are bounded and cleared on view dismissal/backgrounding (NOG-01-005).
- **Geocoding Consent:** Gate network-backed reverse geocoding behind explicit opt-in to avoid leaking precise coordinates to third-party providers during routine operation (NOG-01-013).

It is advised to address all issues identified in this report, including informational and low severity tickets where possible. This will not just strengthen the security posture of the application significantly, but also reduce the number of tickets in future audits.

Once all issues in this report are addressed and verified, a more thorough review, ideally including another source code audit, is highly recommended to ensure adequate security coverage of the platform.

Please note that future audits should ideally allow for a greater budget so that test teams are able to deep dive into more complex attack scenarios. Some examples of this could be third party integrations, complex features that require to exercise all the application logic for full visibility, authentication flows, challenge-response mechanisms implemented, subtle vulnerabilities, logic bugs and complex vulnerabilities derived from the inner workings of dependencies in the context of the application. Additionally, the scope could perhaps be extended to include other internet-facing Noghteha resources.

It is suggested to test the application regularly, at least once a year or when substantial changes are going to be deployed, to make sure new features do not introduce undesired security vulnerabilities. This proven strategy will reduce the number of security issues consistently and make the application highly resilient against online attacks over time.

7ASecurity would like to take this opportunity to sincerely thank Nariman Gharib and the rest of the Noghteha team, for their exemplary assistance and support throughout this audit.