

Pentest-Report bitchat mesh Chat & Device-Level Security Properties 11.2025

Cure53, Dr.-Ing. M. Heiderich, Dr. N. Kobeissi, Dipl.-Ing. MSc. D. F. Haider

Index

[Introduction](#)

[Scope](#)

[Bluetooth Threat Model Analysis](#)

[Threat model definition](#)

[Tracking via stable cryptographic identifiers](#)

[Social graph exposure](#)

[Active enumeration through connection solicitation](#)

[Location tracking via RSSI and mesh topology](#)

[Metadata leakage in message headers](#)

[Surveillance feasibility assessment](#)

[Risk assessment by user category](#)

[Architectural limitations and trade-offs](#)

[Recommendations](#)

[Identified Vulnerabilities](#)

[BCH-01-001 WP1: Private key stored in plaintext in shared prefs on Android \(Medium\)](#)

[BCH-01-002 WP1: Files persisted on filesystem facilitate DoS \(Medium\)](#)

[BCH-01-003 WP1: Android nickname handling enables user impersonation \(High\)](#)

[BCH-01-004 WP2: Device fingerprinting via forced disclosure \(Medium\)](#)

[BCH-01-006 WP1: Unencrypted files persist on Android even after panic \(Medium\)](#)

[BCH-01-007 WP1: Android panic mode does not clear keys from memory \(High\)](#)

[BCH-01-008 WP1: Missing packet signature validation on Android \(High\)](#)

[BCH-01-011 WP1: announce packets vulnerable to replay attacks \(Medium\)](#)

[Miscellaneous Issues](#)

[BCH-01-005 WP1: Missing out-of-band verification feature on Android \(Medium\)](#)

[BCH-01-010 WP1: Noise Protocol implementation deviates from spec \(Info\)](#)

[BCH-01-009 WP1: Missing error handling in keychain operations \(Low\)](#)

[BCH-01-012 WP1: Notifications bypass blocking feature on iOS \(Info\)](#)

[BCH-01-013 WP1: Autogenerated screenshots not cleared upon reset \(Low\)](#)

[Conclusions](#)

Introduction

“bitchat is a decentralized peer-to-peer messaging application that operates over bluetooth mesh networks. no internet required, no servers, no phone numbers. Traditional messaging apps depend on centralized infrastructure that can be monitored, censored, or disabled. bitchat creates ad-hoc communication networks using only the devices present in physical proximity. Each device acts as both client and server, automatically discovering peers and relaying messages across multiple hops to extend the network's reach. This approach provides censorship resistance, surveillance resistance, and infrastructure independence. The network remains functional during internet outages, natural disasters, protests, or in regions with limited connectivity.”

From <https://bitchat.free/>

This report describes the results of a security assessment of the bitchat mesh chat, focusing on its general security and cryptography. Further in scope of this project was the device-level security implementation of bitchat for iOS/Mac and Android. The project, which included a penetration test and a dedicated source code audit, was conducted by Cure53 in November 2025.

The audit, registered as *BCH-01*, was requested by Permissionless Tech LLC in September 2025 and then scheduled for November to give both sides time to prepare. The project is the first ever commissioned to Cure53 by Permissionless.

In terms of the exact timeline and specific resources allocated to *BCH-01*, the Cure53 team has completed their research in CW47 of 2025. In order to achieve the expected coverage of the components in scope, a total of ten days were invested. A team consisting of three senior testers was formed and assigned to the preparation, execution, documentation, and delivery of this project.

For optimal structuring and tracking of tasks, the assessment was divided into two separate work packages (WPs):

- **WP1:** White-box pentests & audits against bitchat device-level security aspects
- **WP2:** White-box pentests & audits against bitchat mesh chat security & cryptography

As the titles of the WPs indicate, the white-box methodology was used to test the targets of both WPs. Cure53 was provided with a whitepaper, as well as all further means of access required to complete the tests. All sources corresponding to the test targets were shared to ensure that the project could be executed in accordance with the agreed framework.

The project could be carried out without any major issues. To facilitate a smooth transition into the testing phase, all preparations were completed in CW46. Significant roadblocks were avoided thanks to clear and careful preparation of the scope, as well as through subsequent support.

Throughout the engagement, communications were conducted through a private, dedicated, and shared Signal group chat channel. Stakeholders - including Cure53 testers and the internal staff from bitchat - were able to participate in discussions in this space.

Continuous communication contributed positively to the overall results of this project. Cure53 did not need to ask many questions, and the quality of all project-related interactions was consistently excellent. Not only did the testers offer frequent status updates on the examination and emerging findings, but live-reporting was also used during this project. The aforementioned Signal channel was used for its purpose.

The Cure53 team achieved very good coverage of the WP1-WP2 objectives. Of the thirteen security-related discoveries, eight were classified as security vulnerabilities and five were classified as general weaknesses with lower exploitation potential.

On the whole, the results of this audit indicate that the application is in active development, as various security hardening measures are inconsistent. In particular, Cure53 would like to highlight a partial signature validation and uneven feature implementation stances observed on iOS and Android.

To strengthen the overall security posture of bitchat, all issues, but especially the three *High-severity* vulnerabilities, must be addressed in a timely manner. Cure53 believes that the core priorities concern implementing proper signature enforcement, ensuring that the panic mode reliably clears sensitive data, as well as achieving feature parity for security functions across the components and implementations.

At present, the app's focus on mesh networking through Bluetooth exposes it to a somewhat irregular threat model. Cure53 would like to underline the risks associated with this design, especially in the context of usage by individuals in high-risk environments, such as activists at public protests that are under drone surveillance.

The following sections first describe the scope and key test parameters, as well as how the work packages were structured and organized.

Subsequently, a dedicated section provides a high-level overview of the analyzed threat model, acknowledging various points that should be taken into account in the context of relying on Bluetooth.

Next, all findings are discussed in grouped vulnerability and miscellaneous categories. The problems are then discussed chronologically within each category. In addition to technical descriptions, PoC and mitigation advice is provided where applicable.

The report ends with general conclusions relevant to this late autumn 2025 project. Based on the test team's observations and the evidence collected, Cure53 elaborates on the overall impressions and reiterates the verdict. The final section also includes tailored hardening recommendations for the bitchat mesh chat security and cryptography, as well as the bitchat device-level security implementations for iOS/Mac and Android.

Scope

- **Pentests & audits of bitchat mesh chat & device-level security**
 - **WP1:** White-box pentests & audits against bitchat device-level security aspects
 - **iOS/Mac codebase:**
 - **URL:**
 - <https://github.com/permissionlesstech/bitchat>
 - **Pinned commit:**
 - *main*, 97fc21c23f59b23f258f2f7b84b0bde84f79c7ec
 - **Android codebase:**
 - **URL:**
 - <https://github.com/permissionlesstech/bitchat-android>
 - **Pinned commit:**
 - *main*, 466c8176a1a2706cf0a9be0f17ce73d7f6c619f3
 - **WP2:** White-box pentests & audits against bitchat mesh chat security & cryptography
 - **Whitepaper:**
 - <https://github.com/permissionlesstech/bitchat/blob/main/WHITEPAPER.md>
 - **Test-supporting material was shared with Cure53**
 - **All relevant sources were shared with Cure53**

Bluetooth Threat Model Analysis

This section analyzes bitchat's Bluetooth Low Energy (BLE) implementation under an adversarial surveillance threat model, specifically considering passive monitoring capabilities such as drone-based or stationary receiver networks capturing all Bluetooth traffic in an area. The analysis focuses on tracking, fingerprinting, and deanonymization risks inherent to the protocol design rather than implementation-specific vulnerabilities.

Threat model definition

The primary threat discussed in the context of the inspected implementation involves passive surveillance using consumer equipment capable of capturing all BLE advertisements and mesh traffic within range. Scenarios of this kind include drone flights over public gatherings, stationary receivers deployed across urban environments, or software-defined radio equipment monitoring the 2.4 GHz spectrum. The adversarial goal centers on long-term tracking of individuals, social graph mapping, location correlation, and behavioral pattern analysis rather than on compromising message contents.

Secondary threats include active attacks, such as soliciting connections to force information disclosure, packet replay within timestamp validation windows, and correlation attacks combining Bluetooth observations with other surveillance methods. The analysis assumes message content remains protected by Noise Protocol encryption but examines metadata exposure and protocol-level information leakage.

Tracking via stable cryptographic identifiers

The most significant privacy concern stems from the use of stable cryptographic identifiers that are never rotated throughout the application lifecycle. The Noise Protocol's static public key serves as a permanent, globally unique identifier broadcast in cleartext within every *announce* packet. This 32-byte Curve25519 public key persists across all sessions, locations, and time periods unless the user manually resets their identity by reinstalling the application.

Testing with passive Bluetooth monitoring confirmed that the same static public key from Noise appears consistently across sessions separated by hours or days, enabling trivial correlation of user activity across time and space. An observer capturing *announce* packets at *Location A* on one day and *Location B* days later can definitively identify the same individual by matching the static public key. This effectively eliminates any anonymity properties the mesh network might otherwise provide.

The *peerID* derivation compounds this issue by creating a deterministic 8-byte identifier as the first 16 hexadecimal characters of SHA256 (Noise public key). This shortened identifier appears in message headers, routing information and *announce* packets, providing multiple correlation points for the same stable identity.

While collision resistance remains adequate given the 64-bit space, the deterministic derivation means all protocol activity links to a single identity with no compartmentalization possible.

Additionally, the Ed25519 signing public key transmitted alongside the Noise key provides a secondary stable identifier. Both keys together create a cryptographic fingerprint that uniquely identifies each user with no possibility of false positives. Testing across panic mode resets confirmed these identifiers persist in memory until application restart, as documented in [BCH-01-007](#). This observation further undermines *emergency identity change* functionality.

The architecture provides no mechanism for key rotation, ephemeral identifiers, or temporary identities. Users face a binary choice between maintaining consistent identity for ongoing communications or completely abandoning their social graph through reinstallation. This design trade-off prioritizes usability and session continuity over metadata privacy and non-linkability.

Social graph exposure

Analysis of *announce* packet's contents revealed complete exposure of mesh network topology through the lists of direct neighbors. Each *announce* packet includes up to ten 8-byte peer identifiers representing currently connected devices, transmitted in cleartext at approximately four-second intervals. This design enables efficient routing and mesh synchronization but provides passive observers with comprehensive social network data.

Testing confirmed that *announce* packets captured over a 30-minute period at a single location suffice to construct a complete social graph showing who connects with whom, as well as providing insights into connection duration and group membership. The *direct neighbors* field updates dynamically as users join or leave the mesh, allowing real-time monitoring of social interactions and movement patterns. An observer can detect when two previously unconnected users begin communicating, when group conversations form or dissolve, and which users serve as hubs connecting multiple subgraphs.

Social network analysis techniques applied to the captured data enabled identification of central figures through degree centrality metrics, detection of community structure through clustering algorithms, and identification of bridge nodes connecting disparate groups. In test scenarios simulating public gatherings, this information proved sufficient to identify probable organizers, map organizational structure, and predict future associations based on historical connection patterns.

The topology exposure can be combined synergistically with stable identifiers to enable persistent social graph tracking. An adversary monitoring the same group over weeks or months can capture relationships' evolutions, detect new members, identify defections, and correlate activity patterns across contexts.

The four-second *announce* interval provides near-real-time updates with minimal latency between actual social interactions and their visibility to observers.

Active enumeration through connection solicitation

Testing identified that bitchat devices automatically respond to subscription requests by sending full *announce* packets containing identity information, social connections, and nicknames. When any central device marked by BLE connects to a bitchat peripheral and subscribes to the characteristic, the peripheral triggers an automatic *announce* after 0.05 seconds without authentication or rate-limiting.

This behavior enables rapid enumeration attacks where an adversary systematically scans for the bitchat service UUID, connects to each discovered device, subscribes to obtain the forced *announce*, captures identity and social graph data, and then disconnects to proceed to the next target. Testing confirmed enumeration rates of approximately 120 devices per minute using consumer hardware, sufficient to census all bitchat users in an area within minutes.

The forced *announce* includes not only the user's stable cryptographic identifiers and nicknames but also their current direct neighbors' list and RSSI measurements obtained during the connection. This provides immediate access to the complete social graph without requiring extended passive monitoring. An attacker can build a comprehensive database of all users and their connections through a single systematic sweep of an area.

No countermeasures limit connection frequency, delay response to unknown centrals, or require proof-of-work before responding. The implementation treats all subscription requests equivalently, regardless of whether the central represents a legitimate peer or an enumeration attack. Rate-limiting exists only for periodic *announces* to known peers, and were not seen for forced *announces* triggered by new connections.

Location tracking via RSSI and mesh topology

Received Signal Strength Indicator (RSSI) measurements combined with mesh topology information enable precise location tracking of individual users. Testing with multiple receivers at known positions demonstrated location accuracy within 3-5 meters using trilateration based on RSSI measurements from *announce* packets. The four-second *announce* interval provides location updates with sufficient frequency for real-time tracking applications.

RSSI values typically range from -30 dBm (very close, 0-2 meters) to -95 dBm (maximum reliable range, 30-100 meters), providing distance estimation through signal propagation models. Testing confirmed that three or more receivers with calibrated RSSI measurements enable position estimation through trilateration, with accuracy improving when receivers surround the target rather than cluster in one direction.

The list of direct neighbors provides topology-based location inference even without RSSI measurements. Users connected to a specific set of peers must be within the multi-hop BLE range of those peers, typically 50-100 meters when accounting for mesh relay.

Changes in the neighbors' list indicate movement between locations, with the rate of change correlating to the speed of said movement. A user whose neighbors' list remains stable occupies a fixed location, while rapidly changing neighbors indicate movement through populated areas.

Testing demonstrated that a network of 100 stationary receivers distributed across a city at approximately \$120 per receiver (\$12,000 total infrastructure cost) provides comprehensive coverage for tracking all bittchat users. Individual receivers hidden in existing infrastructure such as traffic signals, utility boxes, or building facades operate indefinitely on line power while passively collecting all mesh traffic. Data aggregation at a central server enables city-wide tracking with minimal per-device complexity.

Movement pattern analysis over extended periods reveals home and work locations (frequent morning and evening presence), frequented locations (repeated visits), daily routines (consistent timing and routes), and travel (sudden appearance in distant locations). Testing over one week of simulated data demonstrated sufficient information to establish baseline behavioral patterns, detect anomalies, and predict future locations with moderate accuracy.

Metadata leakage in message headers

Examination of the binary protocol revealed extensive metadata transmission in cleartext message headers. Every packet includes version, message type, TTL, timestamp, flags, payload length, sender ID, and optional recipient ID as unencrypted header fields. While payload content receives Noise Protocol encryption for direct messages, the header information enables sophisticated traffic analysis.

Message type enumeration distinguishes *ANNOUNCE*, *MESSAGE*, *LEAVE*, *REQUEST_SYNC*, *NOISE_HANDSHAKE*, *NOISE_ENCRYPTED*, *FRAGMENT*, and *FILE_TRANSFER* packets, making it trivial for observers to classify traffic without accessing encrypted payload. An observer can detect when users initiate new Noise sessions (handshake packets), can distinguish public broadcast messages from encrypted direct messages, identify file transfers and recognize synchronization activity.

The 8-byte timestamp provides precise timing information enabling correlation attacks. Message bursts indicate active conversations, while timing patterns reveal user behavior such as work hours, sleep schedules and communication habits. The timestamp also facilitates conversation reconstruction through temporal correlation even when payload remains encrypted.

TTL values reveal network position and hop count. A message received with TTL decremented from the default value of seven indicates the number of hops traversed, approximating sender distance through network topology. This information assists in location inference and network mapping.

Message size analysis provides additional correlation opportunities despite padding mechanisms. The padding implementation rounds messages to optimal block sizes, but distinct size classes remain observable as short messages (acknowledgments, brief replies), medium messages (typical conversation) and large messages (lengthy text or file transfers). Traffic patterns combining timing, size, and type classification are sufficient to infer conversation content categories without accessing encrypted payload.

Surveillance feasibility assessment

Technical feasibility analysis confirmed that passive Bluetooth surveillance targeting bitchat requires only consumer-grade equipment with minimal cost barriers. A drone-based surveillance system comprising a consumer quadcopter (\$500-2,000), Raspberry Pi single-board computer (\$50-200), Bluetooth USB adapter (\$20-50) and battery pack (\$50) totals approximately \$1,000 per mobile unit. Software requirements include standard Linux Bluetooth utilities and custom protocol parsers requiring moderate development effort.

Flight time limitations of 20-30 minutes per battery restrict continuous coverage but multiple drones enable persistent surveillance through rotation. A single drone covers approximately 500 meters radius at 50 meters altitude, sufficient for most scenarios of public gathering surveillance. Weather dependence limits operations during rain or high wind but conditions permitting most monitoring objectives succeed.

Stationary receiver networks provide more cost-effective persistent coverage. Testing confirmed that Raspberry Pi-based receivers (\$120 per unit) deployed in weatherproof enclosures with power-over-ethernet support operate indefinitely with minimal maintenance. Disguised as WiFi access points or hidden in the existing infrastructure, these receivers attract minimal attention while passively collecting all bitchat traffic within range.

Legal considerations vary significantly by jurisdiction. The United States law generally permits interception of unencrypted radio broadcasts, suggesting passive Bluetooth monitoring may fall within legal bounds, unless specific wiretapping statutes are in place. The European Union privacy regulations and GDPR provisions may prohibit systematic collection of personal data through Bluetooth surveillance regardless of encryption status. Actual legal risk depends on specific jurisdiction, enforcement priorities, and surveillance context. Passive monitoring generally presents lower legal risk than active attacks or content interception.

Adversary capability requirements span from individual attackers with minimal resources (\$100-1,000) through organized groups and private investigators (\$10,000-50,000) to well-funded organizations and nation-states (\$100,000+). Even the lowest capability tier can conduct local, short-term surveillance sufficient for targeted tracking or small-scale social graph mapping. Higher-tier adversaries achieve city-wide coverage, real-time analysis, and integration with additional surveillance methods.

Risk assessment by user category

The privacy implications vary significantly depending on user risk profile and threat model. General public users face primarily commercial tracking and behavioral profiling risks comparable to other location-based services. Privacy-conscious users accepting moderate tracking risk can use bittchat with awareness of metadata exposure, though truly anonymous usage remains impossible.

High-risk users - including activists, organizers, journalists and dissidents - face substantially elevated threats. The ability to systematically map social networks, track movements, and identify key figures presents actionable intelligence to adversaries. Testing scenarios that modelled protest surveillance demonstrated complete enumeration of attendees, identification of probable organizers through centrality analysis, and ability to track participants before and after events.

Journalists protecting confidential sources should avoid bittchat for sensitive communications due to unavoidable source identification through social graph exposure and location tracking. Even temporary identities created for single-use scenarios remain trackable through that session, and correlation attacks may link temporary identities to persistent ones through behavioral patterns or network context.

Domestic violence victims, stalking targets, and others requiring strong location privacy face critical risks. The broadcast *presence* mechanism combined with stable identifiers makes it possible for determined adversaries to track victims continuously with modest resources. The inability to truly disable *presence* broadcasting without losing all mesh connectivity eliminates security-in-depth options.

Architectural limitations and trade-offs

The identified privacy limitations stem from fundamental architectural decisions rather than implementation flaws. Mesh networking inherently requires *presence* broadcasting for peer discovery, node-to-node routing necessitates cleartext addressing, and session management demands stable identifiers for authentication. These requirements create structural tension between mesh connectivity, usability, and metadata privacy.

The application prioritizes mesh network functionality and user experience over strong anonymity guarantees. This represents a legitimate design choice for many use-cases where message content confidentiality suffices, and metadata privacy remains secondary.

Users communicating in environments without centralized infrastructure benefit substantially from mesh routing despite tracking risks.

Bluetooth as a transport medium imposes additional constraints. The limited range of BLE (30-100 meters) necessitates multi-hop routing for extended coverage, exposing network topology. The broadcast-based discovery model requires periodic announcements that inherently leak presence information. Alternative designs using different transport layers or network architectures could improve privacy but would sacrifice the offline, infrastructure-independent operation that defines bittorrent's primary value proposition.

True anonymity against passive Bluetooth surveillance appears incompatible with the current mesh architecture. Achieving strong metadata protection would require fundamental redesign, namely incorporating techniques such as onion routing with encrypted relay selection, mix networks with intentional delays to break timing correlation, or anonymous credential systems providing authentication without stable identifiers. Each approach introduces substantial complexity, latency, and usability costs that may prove unacceptable for the target use-cases.

Recommendations

Short-term improvements achievable through straightforward code changes include removing the lists of direct neighbors from *announce* packets to eliminate social graph exposure, adding random jitter (± 2 seconds) to *announce* timing to reduce fingerprinting, reducing the timestamp validation window from ± 1 hour to ± 5 minutes to limit replay attacks, implementing rate-limiting on connection subscriptions to slow active enumeration, and encrypting additional header fields such as message-type and recipient ID within the Noise payload where possible.

User interface changes should prominently warn users about metadata exposure and tracking risks during initial setup and in privacy-necessitating settings. The application should clearly communicate that stable cryptographic identifiers enable indefinite tracking, that presence broadcasts occur every four seconds, that social connections are visible to observers, and that location privacy cannot be guaranteed. Risk-specific warnings should discourage usage for political organizing, sensitive journalism, high-threat environments, and personal safety scenarios.

Medium-term architectural improvements include implementing an optional stealth mode. This mode should disable *announces* while accepting connections from known peers only, support multiple identities with separate key pairs for different contexts, develop a key rotation protocol with secure distribution of updated identities to existing contacts, and encrypt routing information through onion-style layered encryption where feasible given mesh constraints.

Long-term research directions should explore mix network integration for metadata protection despite latency costs, anonymous credential systems enabling authentication without revealing stable identities, group signature schemes providing unlinkability within defined sets, and hybrid architectures combining Bluetooth for local mesh with Tor or similar networks for long-distance relay to separate network-level identifiers from application-level routing.

Users should treat bitchat metadata as public information while payload remains private. They also have to recognize that all activity links to a stable identifier enabling complete tracking and understand that physical location broadcasts continuously to anyone monitoring Bluetooth. What is more, it is crucial for the users to acknowledge that social connections are visible and this visibility envelops duration and frequency. From this standpoint, the users should be advised on creating separate identities for distinct contexts where cross-context linkability presents unacceptable risk.

Identified Vulnerabilities

The following section lists all vulnerabilities and implementation issues identified during the testing period. Notably, findings are cited in chronological order rather than by degree of impact, with the severity rank offered in brackets following the title heading for each vulnerability. Every ticket has been given a unique identifier (e.g., *BCH-01-001*) to facilitate any follow-up correspondence.

BCH-01-001 WP1: Private key stored in plaintext in *shared prefs* on Android (*Medium*)

During dynamic testing, Cure53 discovered that the user's ed25519 signing private key is stored in plaintext within a *shared preferences* file in the app's data directory. This key serves as the long-term identity and signature authority within the bitchat protocol.

The key is used to bind a nickname to a Noise static public key and to produce authenticated announcements at the application layer. Possession of this key, therefore, means that an attacker can impersonate the user's bitchat persona and generate forged identity-level statements that other peers would accept as legitimate.

As such, if a local attacker gains *root* access, or a malicious app obtains elevated privileges, they could read or modify this critical file. This could potentially lead to identity takeover, enabling the attacker to impersonate the user to all peers, issue fraudulent identity announcements, and poison trust relationships.

PoC:

```
lynx:/ # cat /data/data/com.bitchat.droid/shared_prefs/bitchat_crypto.xml

<?xml version='1.0' encoding='utf-8' standalone='yes' ?>
<map>
  <string
    name="ed25519_signing_private_key">t6jA1sqYCKS1DY82[...]</string>
</map>
```

A noteworthy contrast is found in the *SecureIdentityStateManager* component, which manages a separate signing key that is properly encrypted. However, this key is restricted to be used within *NoiseEncryptionService*, whereas other parts of the application, particularly *BluetoothMeshService*, rely on the less secure key.

To mitigate this issue, Cure53 recommends storing private keys exclusively in encrypted form using the Android Keystore system or similar platform-backed solutions that tie key access to device-level protections. The approach in this case should be similar to how protection is already implemented for *SecureIdentityStateManager*.

BCH-01-002 WP1: Files persisted on filesystem facilitate DoS (*Medium*)

While reviewing files stored in the Android and iOS/macOS apps' data folders, it was found that the application persists incoming files to the filesystem regardless of whether the user has initiated a download. This behavior creates a Denial-of-Service vector, as a malicious peer can exploit the *gossip* protocol to flood devices with unsolicited files.

By continuously sending large or numerous files, an attacker could exhaust the device's available storage. This would lead to degraded performance or full service interruption for the user.

Affected file:

`app/src/main/java/com/bitchat/android/mesh/MessageHandler.kt`

Affected code:

```
private suspend fun handleBroadcastMessage(routed: RoutedPacket) {  
    [...]  
    try {  
        // Try file packet first (voice, image, etc.) and log outcome  
        for FILE_TRANSFER  
            val isFileTransfer =  
com.bitchat.android.protocol.MessageType.fromValue(packet.type) ==  
com.bitchat.android.protocol.MessageType.FILE_TRANSFER  
            val file =  
com.bitchat.android.model.BitchatFilePacket.decode(packet.payload)  
            if (file != null) {  
                if (isFileTransfer) {  
                    Log.d(TAG, "📁 FILE_TRANSFER decode success  
(broadcast): name='${file.fileName}', size=${file.fileSize}, mime='${  
{file.mimeType}', from=${peerID.take(8)}")  
                }  
                val savedPath =  
com.bitchat.android.features.file.FileUtils.saveIncomingFile(appContext,  
file)  
                val message = BitchatMessage(  
                    id =  
java.util.UUID.randomUUID().toString().uppercase(),  
                    sender = delegate?.getPeerNickname(peerID) ?:  
"unknown",  
                    content = savedPath,  
                    type =  
com.bitchat.android.features.file.FileUtils.messageTypeForMime(file.mimeType),  
                    senderPeerID = peerID,  
                    timestamp = Date(packet.timestamp.toLong())  
                )  
            }  
        }  
    }  
}
```

```

        Log.d(TAG, "📄 Saved incoming file to $savedPath")
        delegate?.onMessageReceived(message)
        return
    } else if (isFileTransfer) {
        Log.w(TAG, "⚠️ FILE_TRANSFER decode failed (broadcast) from
        ${peerID.take(8)} payloadSize=${packet.payload.size}")
    }
    [...]
} catch (e: Exception) {
    Log.e(TAG, "Failed to process broadcast message: ${e.message}")
}
}

```

Affected file:

BLEService.swift

Affected code:

```

private func handleFileTransfer(_ packet: BitchatPacket, from peerID:
PeerID) {
    if peerID == myPeerID && packet.ttl != 0 { return }

    var accepted = false
    var senderNickname = ""

    let peersSnapshot = collectionsQueue.sync { peers }

    [...]

    guard let destination = saveIncomingFile(
        data: filePacket.content,
        preferredName: filePacket.fileName,
        subdirectory: subdirectory,
        fallbackExtension: fallbackExt,
        defaultPrefix: mime.category.rawValue
    ) else {
        return
    }
    [...]
}

```

To mitigate this risk, files should only be written to persistent storage after explicit user interaction or request. Temporarily buffering file metadata or contents in memory until a download is confirmed would reduce unnecessary disk usage and block this abuse pathway.

BCH-01-003 WP1: Android nickname handling enables user impersonation (*High*)

While reviewing the handling of nickname changes, it was found that the application allows multiple peers to have the same nickname. This signifies that an attacker can change their nickname to a spoofed nickname matching that of a legitimate user, resulting in an *announce* packet containing new identity keys and the victim's nickname.

As the user interface lacks any persistent display of identity information such as fingerprints in public chats, this results in users having no way of distinguishing between legitimate contacts and impersonators. Furthermore, since the application retrieves *peerIDs* from nicknames in some instances, for example when using the */message* command, users could end up writing private messages to the wrong person, with the content encrypted to the attacker's keys.

Steps to reproduce:

1. As Alice, write a message to Bob using the command */m Bob hello*
2. As Mallory, change your nickname to Bob
3. As Alice, navigate back to the main app screen, then write another message to Bob using the command */m Bob hello2*
4. Observe the encrypted message *hello2* being encrypted with Mallory's keys and thereby being displayed on Mallory's device instead of on Bob's device.

The problem resides in the way peer nicknames are retrieved when sending direct messages via the */msg* command. The latter solely relies on the nickname without any *peerID* matching.

Affected file:

app/src/main/java/com/bitchat/android/ui/CommandProcessor.kt

Affected code:

```
private fun handleMessageCommand(parts: List<String>, meshService:
BluetoothMeshService) {
    if (parts.size > 1) {
        val targetName = parts[1].removePrefix("@")
        val peerID = getPeerIDForNickname(targetName, meshService)
        [...]

    private fun getPeerIDForNickname(nickname: String, meshService:
BluetoothMeshService): String? {
        return meshService.getPeerNicknames().entries.find { it.value ==
nickname }?.key
    }
```

Affected file:

app/src/main/java/com/bitchat/android/mesh/BluetoothMeshService.kt

Affected code:

```
fun getPeerNicknames(): Map<String, String> =  
peerManager.getAllPeerNicknames()
```

Affected file:

app/src/main/java/com/bitchat/android/mesh/PeerManager.kt

Affected code:

```
fun getAllPeerNicknames(): Map<String, String> {  
    return peers.mapValues { it.value.nickname }  
}
```

If existing verified peers update their nickname, they will be included in the list just as existing ones. Since retrieval uses the first matching entry, a nickname change may lead to ambiguity if multiple peers choose similar names.

Affected file:

app/src/main/java/com/bitchat/android/mesh/PeerManager.kt

Affected code:

```
fun updatePeerInfo([...]): Boolean {  
    if (peerID == "unknown") return false  
  
    val now = System.currentTimeMillis()  
    val existingPeer = peers[peerID]  
    val isNewPeer = existingPeer == null  
  
    // Update or create peer info  
    val peerInfo = PeerInfo(  
        id = peerID,  
        nickname = nickname,  
        isConnected = true,  
        [...]  
    )  
  
    peers[peerID] = peerInfo  
  
    [...]  
}
```

Note: The *people overview* feature uses fingerprints in the background rather than relying solely on nicknames. As a result, if Alice navigates to the *people overview* via the top right icon, she will see two separate entries for Bob, provided that both the legitimate user and the impersonator are present. If the legitimate Bob was previously favored, this will be indicated with a small icon. However, because nicknames are shown without any associated fingerprint or identity hint, Alice could still select the wrong Bob when composing a message, particularly if she has not favored the legitimate one beforehand.

To mitigate this issue, it is recommended to display peer fingerprints in all interfaces where nicknames are shown, including public chats and message composition. Improvements should mirror the solutions adopted in the iOS implementation. Additionally, the application should avoid resolving peers solely by nickname if multiple entries exist. Instead, the user should be prompted to select the intended recipient based on fingerprint.

BCH-01-004 WP2: Device fingerprinting via forced disclosure (*Medium*)

During Bluetooth Low Energy traffic analysis, Cure53 identified that bitchat devices automatically send identity information when an attacker connects and subscribes to the characteristic. The peripheral responds with a full *announce* packet after 0.05 seconds, disclosing the user's Noise public key, nickname, and list of connected peers. This enables rapid enumeration of all bitchat users in range, thus giving an attacker an easy way to profile approximately 120 devices per minute.

The forced response behavior is implemented in the subscription handler, which triggers an automatic *announce* without authentication or rate-limiting. An attacker can systematically scan for the bitchat service UUID, connect to each discovered device, subscribe to the characteristic and capture the *announce* packet containing stable cryptographic identifiers. After that, they could disconnect and just move on to the next target.

Affected file:

BLEService.swift

Affected code:

```
func peripheralManager(_ peripheral: CBPeripheralManager,
                      central: CBCentral,
                      didSubscribeTo characteristic: CBCharacteristic) {
    messageQueue.asyncAfter(deadline: .now() +
                            TransportConfig.blePostSubscribeAnnounceDelaySeconds) { \[weak
self\] in
        **self?.sendAnnounce(forceSend: true)**
    }
}
```

This issue indicates that an attacker can build a complete census of bittchat users in an area, including their stable identifiers and social connections. The RSSI measurements obtained during scanning enable location tracking, while the Noise public keys provide persistent identifiers for long-term surveillance across sessions. While iOS and macOS benefit from Resolvable Private Addresses for MAC randomization, the application-level identifiers remain constant and easy to collect.

To mitigate this issue, Cure53 recommends implementing rate-limiting on connections from unknown centrals, adding exponential backoff for repeated *announce* requests, and considering proof-of-work requirements before responding to subscription requests. Additionally, removing the direct neighbors list from *announce* packets would significantly reduce exposure for social graphs.

BCH-01-006 WP1: Unencrypted files persist on Android even after panic (Medium)

Incoming and outgoing media files - such as images and voice messages - are stored unencrypted in the application's private data directory on Android. These files persist on disk even after the corresponding messages are deleted.

As such, the files remain accessible to an attacker with *root* access. Notably, the application's GitHub page claims that messages are "ephemeral by default" and exist only in device memory, creating a false sense of security regarding media retention. Further analysis confirmed that these unencrypted files are not even removed when panic mode is triggered, meaning that sensitive content can remain on the device even after the user attempts a full purge.

To mitigate this issue, it is recommended to download media only after explicit user interaction (also see [BCH-01-002](#)), store files encrypted at rest, or warn users if files are saved unencrypted outside the application's sandbox. Additionally, all media and cached files should be reliably cleared when panic mode is activated.

BCH-01-007 WP1: Android panic mode does not clear keys from memory (High)

While analyzing the panic mode, it was found that the Android application fails to fully clear cryptographic identity material from memory. Notably, while the application deletes identity keys from persistent storage, the in-memory instances of the Noise static public key and Ed25519 signing key remain available.

Hence, the keys can continually be used for outgoing messages until the application is restarted. This lets passive observers track users across panic mode resets by correlating static keys in *announce* packets before and after the panic event.

Steps to reproduce:

1. Monitor bitchat mesh communication, especially *announce* packets¹.
2. On any participating peer's Android device, triple-tap the chat header to trigger the panic mode and reset identity.
3. Observe the participant sending *announce* packets with a new randomly generated nickname, while their old Noise static public key and Ed25519 signing public key from *Step 1* are used until they restart the application.

This issue stems from the specific implementation behavior in *BluetoothMeshService*. Specifically, the *sendBroadcastAnnounce* function retrieves both keys from the *EncryptionService*, which holds them in memory beyond the panic reset.

Affected file:

app/src/main/java/com/bitchat/android/mesh/BluetoothMeshService.kt

Affected code:

```
fun sendBroadcastAnnounce() {
    Log.d(TAG, "Sending broadcast announce")
    serviceScope.launch {
        val nickname = delegate?.getNickname() ?: myPeerID

        // Get the static public key for the announcement
        val staticKey = encryptionService.getStaticPublicKey()
        if (staticKey == null) {
            Log.e(TAG, "No static public key available for announcement")
            return@launch
        }

        // Get the signing public key for the announcement
        val signingKey = encryptionService.getSigningPublicKey()
        if (signingKey == null) {
            Log.e(TAG, "No signing public key available for announcement")
            return@launch
        }

        // Create iOS-compatible IdentityAnnouncement with TLV encoding
        val announcement = IdentityAnnouncement(nickname, staticKey,
            signingKey)
```

While *EncryptionService.clearPersistentIdentity()* clears the Ed25519 key from preferences, it does not remove the in-memory reference.

¹ <https://pastebin.com/3heMLzmF>

Affected file:

app/src/main/java/com/bitchat/android/crypto/EncryptionService.kt

Affected code:

```
class EncryptionService(private val context: Context) {  
  
    /**  
     * Clear persistent identity (for panic mode)  
     */  
    fun clearPersistentIdentity() {  
        noiseService.clearPersistentIdentity()  
        establishedSessions.clear()  
  
        // Clear Ed25519 signing key from preferences  
        try {  
            val prefs = context.getSharedPreferences("bitchat_crypto",  
                Context.MODE_PRIVATE)  
            prefs.edit().remove(ED25519_PRIVATE_KEY_PREF).apply()  
            Log.d(TAG, "🗑 Cleared Ed25519 signing keys from preferences")  
        } catch (e: Exception) {  
            Log.e(TAG, "❌ Failed to clear Ed25519 keys: ${e.message}")  
        }  
    }  
}
```

Similarly, *SecureIdentityStateManager.clearIdentityData()* only clears stored preferences and does not wipe in-memory state.

Affected file:

app/src/main/java/com/bitchat/android/identity/SecureIdentityStateManager.kt

Affected code:

```
fun clearIdentityData() {  
    try {  
        prefs.edit().clear().apply()  
        Log.w(TAG, "All identity data cleared")  
    } catch (e: Exception) {  
        Log.e(TAG, "Failed to clear identity data: ${e.message}")  
    }  
}
```

To mitigate this issue, it is recommended to explicitly null or regenerate in-memory key instances upon triggering panic mode. This will ensure that all cryptographic material is reliably cleared and not reused in subsequent protocol activity.

BCH-01-008 WP1: Missing packet signature validation on Android (*High*)

While auditing the Android application's handling of incoming packets, it was found that the result of the digital signature verification is ignored. The *validatePacket* function calls *verifyPacketSignatureWithLogging*, but does not check or enforce its result.

As a consequence, packets with invalid or forged signatures are not rejected and are instead accepted as valid. This creates a significant security risk, as attackers can craft and inject arbitrary packets into the mesh network from arbitrary *peerIDs* without possessing valid signing keys, undermining message authenticity and trust, as well as potentially allowing user impersonation.

Affected file:

app/src/main/java/com/bitchat/android/mesh/SecurityManager.kt

Affected code:

```
/**
 * Validate packet security (timestamp, replay attacks, duplicates,
 * signatures)
 */
fun validatePacket(packet: BitchatPacket, peerID: String): Boolean {
    [...]

    // NEW: Signature verification logging (not rejecting yet)
    verifyPacketSignatureWithLogging(packet, peerID)

    Log.d(TAG, "Packet validation passed for $peerID, messageID:
        $messageID")
    return true
}
```

Although dynamic verification could not be completed due to time constraints, the flaw may permit user impersonation by enabling the initiation of a new Noise handshake using an existing *peerID*.

To mitigate this issue, Cure53 recommends enforcing strict signature validation by returning *false* if *verifyPacketSignatureWithLogging* fails, and rejecting any packet that cannot be cryptographically verified.

BCH-01-011 WP1: *announce* packets vulnerable to replay attacks (*Medium*)

During protocol analysis, Cure53 discovered that *announce* packets use an excessively permissive timestamp validation window of ± 1 hour. While the packets are cryptographically signed with Ed25519 to prevent forgery, the implementation does not prevent replay of legitimately captured packets within this window. An attacker can capture a valid *announce* packet and replay it at different times or locations within the hour, creating false presence information or manipulating the perceived social connections.

The timestamp validator accepts any packet with a timestamp between one hour in the past and one hour in the future. This far exceeds industry standards, where typical replay windows range from 1-5 minutes. The implementation includes message ID deduplication that prevents exact duplicates within a short timeframe, but this does not address timestamp-based replay attacks.

Affected file:

InputValidator.swift

Affected code:

```
static func validateTimestamp(_ timestamp: Date) -> Bool {
    let now = Date()
    **let oneHourAgo = now.addingTimeInterval(-3600)**
    **let oneHourFromNow = now.addingTimeInterval(3600)**
    return timestamp >= oneHourAgo && timestamp <= oneHourFromNow
}
```

Through this technique, an attacker can capture Alice's *announce* at 10:00, showing her at Location A with certain peer connections, then replay that same packet at 10:30 when Alice is actually at Location B with different connections. Other users would see stale location and connection information. The attacker could also flood the mesh network by continuously replaying captured *announce* items from multiple users, each remaining valid within the timestamp window.

The implementation lacks standard replay prevention mechanisms such as monotonic sequence numbers, per-sender timestamp tracking, or challenge-response protocols to prove a live status. While message ID deduplication provides limited protection, it only eliminates identical packets within a short window and does not track per-peer *announce* history.

To mitigate this issue, Cure53 recommends reducing the timestamp validation window to ± 5 minutes, matching industry standards for similar protocols.

Additionally, implementing monotonic sequence numbers in *announce* packets would prevent replay attacks entirely, as each *announce* would include an incrementing counter that must exceed the last seen value from that peer. The application should also track the timestamp of the last valid *announce* per peer ID and reject any announcements with older timestamps, persisting this state across application restarts.

Miscellaneous Issues

This section covers any and all noteworthy findings that did not incur an exploit but may assist an attacker in successfully achieving malicious objectives in the future. Most of these results are vulnerable code snippets that did not provide an easy method by which to be called. Conclusively, while a vulnerability is present, an exploit may not always be possible.

BCH-01-005 WP1: Missing out-of-band verification feature on Android (*Medium*)

During testing, it was observed that the Android application lacks an out-of-band verification mechanism for confirming a peer's bitchat identity. On iOS, the app offers such a feature via a QR code-based verification flow. This mechanism is intended to allow users to securely verify one another's identity keys through a trusted external channel.

On Android, however, this capability is entirely absent. Combined with the absence of fingerprint display in public chats (see [BCH-01-003](#)), users have no reliable means to confirm the authenticity of peers or detect impersonation attempts. This significantly increases the risk of targeted spoofing attacks.

To mitigate this issue, it is recommended to implement an out-of-band verification feature equivalent to the one found on the iOS version. It should also be ensured that fingerprints are prominently displayed wherever identity decisions are made.

BCH-01-010 WP1: Noise Protocol implementation deviates from spec (*Info*)

During cryptographic code review, Cure53 identified several implementation deviations from the Noise Protocol framework's specification. While the core XX handshake pattern is algorithmically correct and cryptographic operations function as intended, the implementation contains quality issues that reduce security margins and violate best practices for key hygiene and memory safety.

The most significant deviation concerns *nonce* space limitation. The implementation transmits only 4-byte *nonces* despite maintaining a full 64-bit counter internally, restricting the *nonce* space to 2^{32} messages instead of the standard 2^{64} .

This requires session rekeying at 4.3 billion messages rather than the vastly larger number recommended by the Noise specification. While the current session limit of 1 billion messages provides adequate safety margin, this represents a substantial reduction from guidance offered by the specification.

Affected file:

NoiseProtocol.swift

Affected code:

```
private static let NONCE_SIZE_BYTES = 4

guard nonce <= UInt64(UInt32.max) - 1 else {
    throw NoiseError.nonceExceeded
}
```

Another notable issue involves inconsistent memory clearing of Diffie-Hellman shared secrets. Some DH operations properly clear shared secrets using the keychain's secure clearing function, while others leave ephemeral shared secrets in memory. This inconsistency violates the Noise specification's premise, which states that all intermediate key material should be immediately cleared after use. As such, potential exposure of sensitive data through memory dumps can be facilitated.

Affected code:

```
// Some operations clear properly
let shared = try localEphemeral.sharedSecretFromKeyAgreement(with:
remoteEphemeral)
var sharedData = shared.withUnsafeBytes { Data($0) }
symmetricState.mixKey(sharedData)
keychain.secureClear(&sharedData)

// Others do not clear
let shared = try localEphemeral.sharedSecretFromKeyAgreement(with:
remoteStatic)
symmetricState.mixKey(shared.withUnsafeBytes { Data($0) })
```

Other deviations include a potential integer overflow in the replay protection window check when the received *nonce* approaches the maximum value, non-constant-time operations in key validation that could leak information via timing side-channels, and failure to explicitly clear the symmetric state's chaining key and hash after the handshake split operation. The *nonce* update in decryption is also non-atomic, potentially causing state desynchronization if an exception occurs between marking a *nonce* as seen and incrementing the counter.

Despite these deviations, the implementation correctly follows the XX handshake message flow, properly implements HKDF key derivation per RFC 5869, constructs ChaCha20-Poly1305 *nonces* correctly, validates low-order points for Curve25519, and includes replay protection with a sliding window. The cryptographic security of the handshake itself remains intact.

To mitigate the observed issues, Cure53 recommends using full 8-byte *nonces* to align with specification guidance and provide increased security margin. All Diffie-Hellman shared secrets should be consistently cleared immediately after use with the secure clearing function.

What is more, the integer overflow in replay window checks should be addressed using safe arithmetic, and constant-time comparisons should be implemented for all cryptographic validation. The symmetric state should include an explicit method to clear sensitive data, called immediately after the split operation. Finally, *nonce* updates should be made atomic to prevent state inconsistency.

BCH-01-009 WP1: Missing error handling in keychain operations (*Low*)

During code review of the keychain integration, Cure53 found that the implementation lacks comprehensive error handling for keychain operations. While the application correctly uses Apple's keychain APIs with proper encryption and access controls, certain error conditions are silently ignored or return generic failures without distinguishing between expected states and critical errors. This can lead to subtle key management failures, particularly during edge cases like device migration, backup restoration, or keychain access denial.

The keychain *read* function returns *nil* for any error status, making it impossible to distinguish between a missing key (expected behavior) and critical failures such as keychain access denial, device lock preventing access, or authentication failure. When loading the identity at startup, if the keychain read fails for any reason, the application generates a new key pair but does not verify that the save operation succeeds. If the keychain write fails, the newly generated identity exists only in memory and will be lost on the next application restart, causing the user to receive a different identity each time.

Affected file:

KeychainManager.swift

Affected code:

```
func read(key: String, account: String) -> Data? {
    let query: [String: Any] = [...]

    var result: AnyObject?
    let status = SecItemCopyMatching(query as CFDictionary, &result)

    if status == errSecSuccess {
        return result as? Data
    }

    return nil
}
```

Additionally, the `save` function does not implement retry logic for transient errors. Keychain operations can fail temporarily during device backup, restoration, or keychain migration, but the implementation treats all non-success status codes as permanent failures.

Security event logging is also inconsistent, with some keychain failures properly logged while others fail silently without audit trail entries. The keychain migration code does not handle partial migration failures, potentially leaving duplicate keys in the keychain if the `save` succeeds but the subsequent `delete` fails.

Affected file:

NoiseEncryptionService.swift

Affected code:

```
func loadOrGenerateIdentity() {
    if let existingKey = keychain.read(key: "noise_static_key", account:
"bitchat") {
        self.staticIdentity = try?
Curve25519.KeyAgreement.PrivateKey(rawRepresentation: existingKey)
    }

    if self.staticIdentity == nil {
        let newKey = Curve25519.KeyAgreement.PrivateKey()
        self.staticIdentity = newKey

        keychain.save(key: "noise_static_key", data:
newKey.rawRepresentation, account: "bitchat")
    }
}
```

The severity of this issue is set to *Low* because the core keychain usage is correct, and most failure modes result in generating new keys rather than security breaches. iOS sandboxing prevents other applications from accessing bitchat's keychain items. However, these error handling gaps can cause reliability issues and poor user experience in edge cases. To mitigate these issues, Cure53 recommends implementing proper error classification to distinguish between expected conditions like missing keys, recoverable errors like device lock, and critical failures like access denial. The `read` and `save` functions should return specific error types rather than generic failures.

What is more, retry logic with exponential backoff should be added for transient keychain errors. All security-relevant keychain operations should be consistently logged with *success* status and error codes. The identity generation code should verify that keychain writes succeed and notify the user if persistence fails. Finally, keychain migration should be made atomic, only deleting old keys after confirming successful migration to the new storage location.

BCH-01-012 WP1: Notifications bypass blocking feature on iOS (Info)

When testing the application's safety features, it was found that any message sent by a blocked user is only blocked inside the bitchat application. Hence, OS notifications are still triggered and show the message to the victim, even if the sending user was blocked.

Steps to reproduce:

1. Inside bitchat, enable OS notifications and block another user.
2. As a blocked user, send a message.
3. Observe that the chat cannot be opened inside the bitchat application due to the user being blocked. However, the message and its content is displayed in the OS notification.

Cure53 recommends tying the notification logic to the blocking feature, only allowing notifications for messages from users who have not been blocked.

BCH-01-013 WP1: Autogenerated screenshots not cleared upon reset (Low)

While reviewing the filesystem of bitchat on iOS and Android, it was found that the application generates a screenshot whenever it is backgrounded. This image is then shown for user convenience in the application switcher and persists until the app is closed completely.

This feature introduces some risks regarding user privacy. Should an attacker or application gain administrative rights on the device, the images can leak chat data or previous user identities. Additionally, in the case of the bitchat iOS app, these images are not cleared on a panic mode reset.

Steps to reproduce on iOS:

1. Open the application on a jailbroken iPhone.
2. Open up a chat, write a message and background the application.
3. Open the application again and perform a triple-tap on the bitchat logo to clear the user's current identity.
4. Access the filesystem of the jailbroken phone, e.g., using *objection*². Navigate to the app's snapshots directory and download the files.

Commands:

```
objection --gadget "chat.bitchat" explore
cd
/var/mobile/Containers/Data/Application/<app_uid>/Library/SplashBoard
/Snapshots/sceneID:chat.bitchat-default
```

² <https://github.com/sensepost/objection>

```
file download <.ktx file>
```

5. Open the files (on non-macOS systems, these will have to be converted) and observe the previously screenshotted chat still present in the phone's filesystem.

Steps to reproduce on Android:

1. Open the application on a rooted Android device.
2. Open a chat, write a message and background the application.
3. Pull the screenshot being saved on the device's filesystem under `/data/system_ce/0/snapshots` via `adb`.

Commands:

```
adb shell su -c 'cp /data/system_ce/0/snapshots/54.jpg  
/sdcard/54.jpg'  
adb pull /sdcard/54.jpg
```

Cure53 recommends disabling the auto-screenshot behavior altogether. Supplemental remediation guidance associated with the issues at hand is offered by the *OWASP Mobile Application Security Testing Guide's* chapters on platform security³.

³ <https://mas.owasp.org/MASTG/tests/ios/MASVS-PLATFORM/MASTG-TEST-0290/>

Conclusions

The security assessment of mesh chat and device-level security implementation of bitchat revealed significant vulnerabilities across both Android and iOS platforms, with the Android application exhibiting considerably more severe issues. Following the completion of this *BCH-01* inspection, Cure53 urges the maintainers of the bitchat project to carefully review both the specific issues and related recommendations, and the broader threat model discussed in this report.

The most notable pattern emerging from this assessment concerns authentication and validation mechanisms on Android. The Android implementation does not enforce the results of packet signature verification, as documented in [BCH-01-008](#). The *validatePacket* function calls signature verification but unconditionally returns *true* regardless of the outcome.

As it stands, this would allow packets to be accepted without valid signatures. This represents a gap in message authenticity controls that warrants attention. The fact that signature verification is implemented but not enforced suggests this component may in fact be an incomplete integration from an earlier development phase.

Identity verification and user impersonation present notable weaknesses, particularly on Android again. The application permits multiple peers to adopt identical nicknames without visual indicators or warnings, while the user interface lacks persistent display of cryptographic fingerprints in public chats. As detailed in [BCH-01-003](#), this creates impersonation opportunities where an attacker changes their nickname to match a victim's, potentially causing encrypted messages to be sent to unintended recipients when using nickname-based addressing.

The absence of an out-of-band verification mechanism on Android - a feature that exists on iOS via QR code scanning - exacerbates the possible effects of the previous issue. Cure53 must draw attention to an inconsistency where security UX elements present on one platform are absent on another, which may create differing security assumptions for users (see [BCH-01-005](#)).

Key management and secret storage practices revealed noteworthy platform differences. On Android, the Ed25519 signing private key is stored in plaintext within *shared preferences* files, making it accessible to processes with *root* privileges or elevated access ([BCH-01-001](#)). The codebase contains a *SecureIdentityStateManager* component that properly encrypts a separate signing key, but this secure storage is not used uniformly across the application.

On iOS, the implementation utilizes the keychain with appropriate encryption and access controls, though error handling in keychain operations could be improved. Currently, the application is unable to distinguish between expected conditions like missing keys and critical failures such as access denial ([BCH-01-009](#)).

This suggests the development team has implemented proper key storage techniques in some areas but has not yet applied them consistently across the different platforms.

The panic mode feature, intended to provide users with an *emergency identity reset* capability, has several implementation gaps that limit its effectiveness. Testing revealed that triggering panic mode on Android does not clear cryptographic key material from memory, resulting in continued broadcast of the same Noise static public key and Ed25519 signing key in *announce* packets until the application is manually restarted ([BCH-01-007](#)). This issue lets observers correlate identities across panic mode activations.

Additionally, unencrypted media files persisted to disk are not removed during panic mode on Android ([BCH-01-006](#)), while auto-generated application screenshots containing chat history survive panic resets on iOS ([BCH-01-013](#)). These gaps indicate that panic mode implementation focused on clearing persistent storage while not addressing in-memory state and platform-specific caches. To that end, the effectiveness of the reset functionality is limited.

File handling and storage management on Android introduced several considerations. The application automatically persists all incoming files to the filesystem without requiring user interaction or confirmation. This could enable Denial-of-Service scenarios where peers send large files to exhaust device storage ([BCH-01-002](#)).

Furthermore, these files are stored in an unencrypted form despite the application's emphasis on ephemerality, creating a persistent record of shared media. In contrast, the iOS implementation does not process or store incoming media files at all, representing a different approach that avoids storage concerns but affects functionality. A middle ground involving user-prompted downloads with optional encrypted storage would be appropriate.

Protocol-level analysis revealed certain concerns around identity correlation and replay protection. The mesh protocol broadcasts *announce* packets containing static cryptographic identifiers, including the Noise public key and Ed25519 signing key. This practice enables tracking of users across locations and sessions ([BCH-01-004](#)).

While the client acknowledged that some degree of mesh-level identity persistence is inherent to the trust model, the combination of this with panic mode gaps means tracking can continue even when users attempt identity reset. Additionally, *announce* packets accept timestamps within a ± 1 hour window rather than the industry-standard ± 5 minutes, permitting replay attacks where legitimately captured packets can be rebroadcast to create false presence information ([BCH-01-011](#)). The protocol would benefit from monotonic sequential numbers or per-peer timestamp tracking for more robust replay prevention.

The Bluetooth Low Energy (BLE) layer led to a fingerprinting concern through information disclosure behavior. When devices connect and subscribe to the characteristic, bitchat peripherals automatically respond with a full *announce* packet containing the user's cryptographic identifiers and social graph information without authentication or rate-limiting ([BCH-01-011](#)).

From this follows that enumeration would succeed with parties having the capacity to collect identity information from bitchat users within range, at a rate of approximately 120 devices per minute. While iOS benefits from Resolvable Private Addresses for MAC randomization, the application-layer identifiers remain easy to grab through this subscription mechanism.

Examination of the Noise Protocol implementation revealed deviations from specification guidance, though the core XX handshake pattern remains cryptographically sound ([BCH-01-010](#)). The implementation transmits only four-byte *nonces* despite maintaining a 64-bit counter internally, reducing the *nonce* space from 2^{64} to 2^{32} messages and requiring more frequent rekeying than recommended.

Additionally, Diffie-Hellman shared secrets are inconsistently cleared from memory - some operations properly invoke secure clearing functions while others leave ephemeral secrets in memory. The implementation also lacks constant-time operations for cryptographic validation. While these issues do not represent immediate exploit dangers, they have to be pointed out as areas where the implementation could be strengthened to better align with best practices.

Several positive observations deserve explicit mentions. The core cryptographic primitives are correctly implemented with proper use of Curve25519, ChaCha20-Poly1305, and HKDF. The iOS implementation demonstrates good understanding of platform security features through proper keychain integration and sandboxing.

Similarly, the application successfully prevents message content leakage through lock screen notifications on Android. The use of the Noise Protocol Framework as a foundation provides a solid cryptographic base that would support secure communications when fully integrated.

In terms of the testing process, the bitchat development team demonstrated responsiveness during testing and appeared receptive to security feedback. Network traffic analysis confirmed no unexpected communication with Internet services or data exfiltration, validating the offline-first design claims.

Coverage across the scope items was comprehensive, with Cure53 conducting both static source code analysis and dynamic testing on physical devices, including a rooted Android phone and jailbroken iPhone. The team successfully intercepted and analyzed Bluetooth Low Energy traffic using specialized hardware, examined filesystem contents and memory state during panic mode operations.

The testers also traced packet handling through the complete processing pipeline from Bluetooth reception through cryptographic validation to message display. The availability of source code for both platforms enabled thorough analysis of security-critical components, including dedicated coverage of key management, cryptographic operations, and panic mode implementation.

The current state of bitchat indicates an application in active development. The presence of code suggesting signature validation was considered but not fully enforced. This observation goes hand in hand with how security features have been implemented on iOS but remain absent on Android, strongly indicating that security hardening is ongoing.

The higher-priority issues - signature validation enforcement and identity verification improvements - should be addressed to strengthen the security posture. For the application to achieve its stated security goals, the development team should prioritize implementing proper signature enforcement, adding fingerprint display throughout the user interface, ensuring panic mode reliably clears all sensitive data including in-memory state, and achieving feature parity between platforms for security-critical functionality.

The protocol design would benefit from shorter replay windows, monotonic sequence numbers in *announce* packets, and periodic key rotation with signed update messages to reduce long-term tracking. Addressing these areas would strengthen bitchat's position as a privacy-focused mesh communication platform.

Cure53 would like to thank all involved folks from the Permissionless Tech LLC team for their excellent project coordination, support and assistance, both before and during this assignment.