

ASSURED

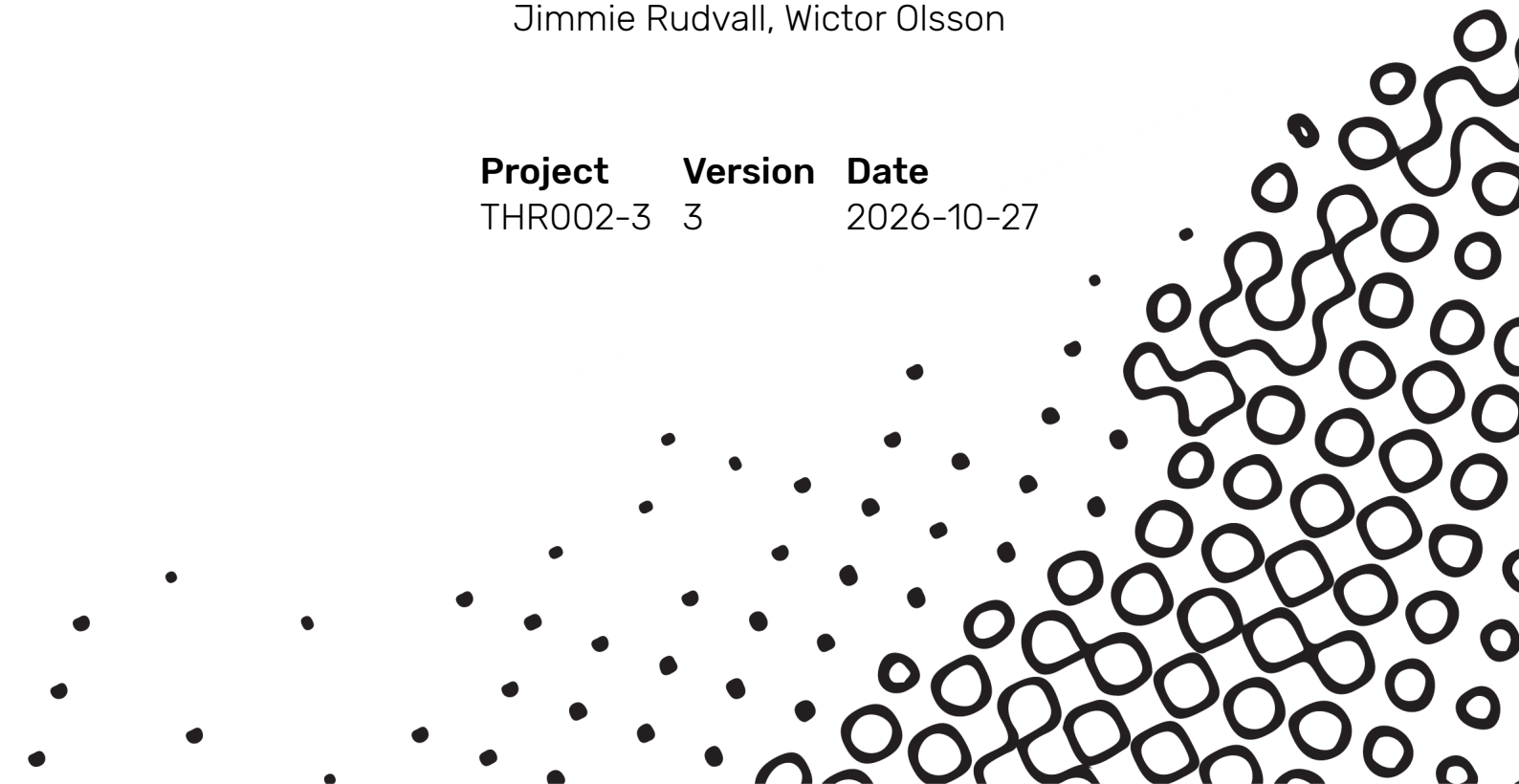
SECURITY CONSULTANTS

Report

Threema Mobile Application Security Assessment

Jimmie Rudvall, Wictor Olsson

Project	Version	Date
THR002-3	3	2026-10-27



ASSURED
SECURITY CONSULTANTS

REPORT

Project	Version	Date
THR002-3	3	2026-10-27

Executive summary

Between 2026-09-14 and 2026-10-02 Assured Security Consultants performed a mobile application security assessment on behalf of Threema AG.

The Assured team performed a security assessment of the Threema mobile applications (Private, Work) for both iOS and Android. The engagement spanned over three weeks, including testing and documentation, and was aligned with the Mobile Application Security Verification Standard (MASVS).

Assessment activities included automated analysis, dynamic testing, and manual code review, supplemented by LLM-assisted code auditing to support broader application analysis.

The Threema applications demonstrated a robust security implementation and alignment with mobile security best practices. A small number of low or informational-level observations were identified during the assessment. No high or critical-severity issues were reported. Overall, the results indicate that the Threema mobile applications maintain a strong security posture for their respective platforms.

This report lists the security issues found, along with recommendations for remediation or mitigation. In our conclusions we discuss the issues and address apparent patterns in areas where security is lacking.

Observations were made with the following risk severity assessments (number of issues):



Assured would like to thank Danilo Bargen, Patrik Oprandi, and Thomas Moser for their support during this penetration test. We are happy to answer any questions and provide further advice.

Contents

1 Introduction **1**
1.1 Background . . . 1
1.2 Constraints and disclaimer . . . 1
1.3 Project period and staffing . . . 1
1.4 Assured Security Consultants . . . 1
1.5 Revision history . . . 1

2 Scope and methodology **2**
2.1 Scope . . . 2
2.2 Methodology . . . 2
2.2.1 Tools used . . . 3
2.3 Observation tagging and risk rating . . . 3
2.3.1 OWASP Risk Rating Methodology . . . 3
2.3.2 Observations without risk rating . . . 4
2.3.3 Verification state tagging . . . 4

3 Observations **5**
3.1 LOW Cryptographic Binding for Biometric Authentication . . . 5
3.2 NOTE FIXED Cargo Patch Management . . . 7
3.3 NOTE ACCEPTED Static Nonce, Blob Encryption . . . 7
3.4 NOTE FIXED Latent Cross-Site Scripting Sink . . . 9

4 Conclusions and recommendations **11**
4.1 Recommendations for future testing . . . 12

1 Introduction

1.1 Background

Assured AB (Assured) was contracted to perform a mobile application security assessment by Threema AG.

1.2 Constraints and disclaimer

This report contains a summary of the observations made during the project period. This report should not be considered as a complete list of all vulnerabilities, security flaws and/or misconfigurations.

1.3 Project period and staffing

Assured started the project on 2026-09-14 and finished on 2026-10-02, dedicating a total of 18 person-days to the effort.

This report was last reviewed on 2026-10-27.

Involved in the penetration testing were Assured consultants Jimmie Rudvall and Wictor Olsson.

1.4 Assured Security Consultants

Assured Security Consultants (Assured AB[1]) was founded in 2015 with the mission to provide premier technical cybersecurity services as an independent consultancy, not affiliated with any vendor. Our team of experienced and dedicated cybersecurity specialists perform penetration testing, red team activities, secure design, embedded development, advisory, training and similar services.

We are committed to the security community as OWASP chapter leaders, event organizers and podcasters. We also take active part in security research projects into areas such as cryptography and automotive security.

1.5 Revision history

v	Date	Description
1	2026-10-06	Initial report
2	2026-10-23	Revised frontmatter, added verification status to Observations
3	2026-10-27	Observation 3.4 verified fixed

[1]Assured AB, Org.nr. 556985-8276, registered in Sweden. www.assured.se

2 Scope and methodology

2.1 Scope

In scope were the Threema mobile applications including Threema Work and Private for both Android and iOS.

Target repositories:

- `https://github.com/threema-ch/threema-android`
- `https://github.com/threema-ch/threema-ios`

Android commit: `1e8ce857f3f8a656667da247260db39598279105`

iOS commit: `356ae5ca9ae3944d622aa5ae714d02d16cd03bd7`

Debug application packages were also supplied by Threema.

2.2 Methodology

The assessment of the Threema iOS and Android applications was based on the OWASP Mobile Application Security Verification Standard (MASVS) as the primary baseline. Testing covered MASVS requirements relevant to local storage, cryptography, authentication and session handling, network communication, platform interaction, input validation and code quality.

Automated analysis was performed to identify common mobile application issues, packaging characteristics, permission configurations, hardcoded values, and other static indicators.

Dynamic testing was performed to observe application behavior at runtime, inspect network communication, analyze API interactions, and verify security controls in operation.

Manual code review was performed to evaluate data handling, authentication logic, storage mechanisms, cryptographic implementation, input validation, and platform-specific behavior.

LLM-assisted audit support was used to assist with code review and identify candidate weaknesses. All LLM-supported observations were manually reviewed and validated before being included in the results.

2.2.1 Tools used

- Jailbroken iOS device: runtime execution and inspection of the iOS application.
- Android emulator: controlled runtime environment for dynamic testing of the Android application.
- Burp Suite Professional: network traffic interception, API analysis, and request-level inspection.
- Frida.re: dynamic instrumentation to observe runtime behavior and application logic.
- Objection: runtime inspection of mobile application state and platform interactions.
- MobSF: automated mobile application analysis.
- Semgrep: static code analysis and pattern-based review.
- Cargo-audit: rust crate vulnerability analysis.

2.3 Observation tagging and risk rating

2.3.1 OWASP Risk Rating Methodology

In this report we have assessed the severity of issues and identified vulnerabilities according to the OWASP Risk Rating Methodology [1].

Table 1: OWASP Risk Rating overall severity model

Overall risk severity				
Impact	HIGH	Medium	High	Critical
	MEDIUM	Low	Medium	High
	LOW	Note	Low	Medium
		LOW	MEDIUM	HIGH
	Likelihood			

As Table 1 visualizes, the overall risk assessment is determined from a combined likelihood and impact of an identified vulnerability or security issue. A value from 0 to 9 is assessed for each variable, where 0-2 is designated LOW, 3-5 is MEDIUM and 6-9 is HIGH.

Likelihood is dependent on attributes related to threat actors and the identified vulnerability, with factors such as: the skill level and motivations of the threat agents; how easily the vulnerability can be found and exploited, and; how likely an exploit may be detected.

Impact depends on technical and business factors, such as: level of loss of confidentiality, integrity, availability and accountability; potential financial damage; potential brand damage, and; potential violations of privacy.

Please note that the severity assessment is made by Assured consultants and ratings may differ from the resource owners' ratings.

2.3.2 Observations without risk rating

Observations that do not pose a direct vulnerability or security issue, but should be addressed, we mark with NOTE.

Any observations that we deem follow good security practice or aligning with provided requirements, we mark with GOOD.

2.3.3 Verification state tagging

When we conduct a verification test, we tag the re-tested observation with the state depending on whether they are fixed, partially fixed, remaining or accepted, with the following corresponding tags:

- FIXED for verified fixed findings.
- PARTIALLY FIXED for findings where we found partial mitigation in effect.
- REMAINING where the finding was verified still valid.
- ACCEPTED where the finding was valid but will not be fixed by the customer for some reason.

3 Observations

This section contains the observations made during the assessment. If an observation was deemed to only affect a specific version of Threema this is specified in the description of the observation.

3.1 LOW Cryptographic Binding for Biometric Authentication

Likelihood: LOW (1), Impact: MEDIUM (3)

We observed that biometric authentication for app locking does not appear to use the CryptoObject functionality of Android's BiometricPrompt.

In a modified or instrumented runtime environment, an attacker could potentially hook the authentication callback and cause it to report success without valid biometric verification. This may allow an application user or runtime instrumentation tool to trigger the unlocked state without completing the intended biometric check.

The implementation appears to rely on the `onAuthenticationSucceeded` callback to indicate that biometric authentication has completed successfully. No cryptographic binding between the biometric result and a subsequent sensitive key operation was identified, as is indicated in Listing 1.

According to Android guidance, when protecting sensitive operations with biometric authentication, `BiometricPrompt.authenticate()` should be used together with a `CryptoObject` backed by an Android Keystore key configured to require user authentication. This cryptographically binds the biometric result to the protected operation, rather than relying solely on a success callback.

app/src/main/java/ch/threema/app/applock/AppLockUtil.kt:63+

Listing 1: Biometric validation relies on AuthenticationCallback return value without CryptoObject validation

```
63        return suspendCoroutine { continuation ->
64            val prompt = BiometricPrompt(
65                activity,
66                executor,
67                object : BiometricPrompt.AuthenticationCallback() {
68                    override fun onAuthenticationSucceeded(result: BiometricPrompt.
    AuthenticationResult) {
69                        logger.info("Successfully authenticated (auth type={})", result.
    authenticationType)
70                        continuation.resume(AuthenticationResult.Success)
71                    }
```

In this context, it represents a potential weakness in the robustness of biometric-based access control rather than an issue affecting users with standard, unmodified devices.

Furthermore, user confirmation is not required as can be seen in Listing 2. The implementation allows fallback to device credentials, a compatibility choice, seen in Listing 3.

app/src/main/java/ch/threema/app/applock/AppLockUtil.kt:98+

Listing 2: Confirmation is not required

```
98            val promptInfo = PromptInfo.Builder().apply {
99                setTitle(title)
100               setSubtitle(subtitle)
101               setConfirmationRequired(false)
102               setAllowedAuthenticators(authenticators)
103               if ((authenticators and DEVICE_CREDENTIAL) == 0) {
104                   setNegativeButtonText(activity.getString(R.string.cancel))
105               }
106           }
```

app/src/main/java/ch/threema/app/applock/AppLockUtil.kt:113+

Listing 3: Fallback flow

```
113   private fun getEffectiveAuthenticators(desiredAuthType: AuthType): Int {
114       val biometricsAuthenticatorState: BiometricsState = getBiometricsAuthenticatorState()
115       return when (desiredAuthType) {
116           AuthType.ANY ->
117               if (biometricsAuthenticatorState == BiometricsState.AVAILABLE) {
118                   DEVICE_CREDENTIAL or BIOMETRIC_WEAK
119               } else {
120                   getDeviceCredentialAuthenticatorCompat()
121               }
122           AuthType.BIOMETRIC ->
123               if (biometricsAuthenticatorState == BiometricsState.AVAILABLE) {
124                   BIOMETRIC_WEAK
125               } else {
126                   logger.warn("Biometrics not available, fallback to credentials")
127                   getDeviceCredentialAuthenticatorCompat()
128               }
129       }
130   }
```

We recommend to consider strengthening the implementation. Because biometric authentication can be cryptographically bound to a protected operation, it will make it more difficult to bypass biometric authentication through runtime callback manipulation or instrumentation. Also consider requiring user confirmation of biometrics usage.

Find good security practices at: `https://mas.owasp.org/MASTG/best-practices/MASTG-BEST-0036/`, and `https://mas.owasp.org/MASTG/best-practices/MASTG-BEST-0038/`.

3.2 NOTE FIXED Cargo Patch Management

Threema confirmed that a patch management process is already in place. The identified vulnerabilities were publicly disclosed shortly after the build date, just before testing began. Additionally, Threema verified that the affected packages are excluded from production builds and are used solely within a CLI-based testing environment.

Automated dependency analysis of the Threema Rust library component, performed using cargo-audit against the RustSec advisory database, identified two known vulnerabilities and one unsoundness warning among 337 crate dependencies.

- Crate: **h2**, Issue: `https://rustsec.org/advisories/RUSTSEC-2026-0258`
- Crate: **rustls**, Issue: `https://rustsec.org/advisories/RUSTSEC-2026-0285`

The rustls advisory relates to the handling of TLS 1.3 handshake messages and may allow certain messages to be accepted across encryption level boundaries. In practice, exploitation requires an active network attacker positioned in a position to manipulate TLS handshake traffic. The h2 advisory relates to the HTTP/2 framing layer and concerns unbounded processing of empty DATA frames, which could affect resource consumption under specific adversarial conditions.

Overall, the identified dependencies are at versions that have known fixes available upstream. The practical risk to end users is low.

We recommend to upgrade the indicated crates and continuously monitor for vulnerabilities in dependencies.

3.3 NOTE ACCEPTED Static Nonce, Blob Encryption

This finding is a hardening recommendation. Risk is mitigated by the architecture's cryptographic design. A static nonce is combined with a unique, single-use key to ensure data confidentiality and eliminates the need to transmit the nonce alongside encrypted data. Addressing this would introduce backward incompatibility and is therefore an accepted risk.

Static nonce values were identified for multiple file and photo blob encryption operations.

These nonces, as listed in Listing 4, are passed to `NaCl.symmetricEncryptData()` using the XSalsa20-Poly1305 AEAD construction when encrypting uploaded blobs (see Listing 5). A fresh random 256-bit key is generated for each encryption operation. The nonce, however, remains fixed per blob type across all invocations and all users.

domain/src/main/java/ch/threema/domain/protocol/csp/ProtocolDefines.java:160+

Listing 4: Static nonce values

```
160     /* nonces */
161     public static final byte[] GROUP_PHOTO_NONCE = new byte[]{0x00, 0x00, 0x00, 0x00, 0x00, 0x00
        , 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
        0x00, 0x00, 0x01};
162     public static final byte[] CONTACT_PHOTO_NONCE = new byte[]{0x00, 0x00, 0x00, 0x00, 0x00, 0
        x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0
        x00, 0x00, 0x00, 0x01};
163     public static final byte[] FILE_NONCE = new byte[]{0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
         0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0
        x00, 0x01};
164     public static final byte[] FILE_THUMBNAIL_NONCE = new byte[]{0x00, 0x00, 0x00, 0x00, 0x00, 0
        x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0
        x00, 0x00, 0x00, 0x02};
```

app/src/main/java/ch/threema/app/profilepicture/GroupProfilePictureUploader.kt:38+

Listing 5: Use of nonce value

```
38      private fun uploadGroupProfilePicture(profilePicture: ProfilePicture):
        GroupProfilePictureUploadResult =
39          try {
40              val encryptionKey = secureRandom.generateRandomBytes(NaCl.SYMM_KEY_BYTES)
41              val encryptedData = NaCl.symmetricEncryptData(
42                  data = profilePicture.bytes,
43                  key = encryptionKey,
44                  nonce = ProtocolDefines.GROUP_PHOTO_NONCE,
45              )
46              val blobUploader = apiService.createUploader(
47                  /* data = */
48                  encryptedData,
49                  /* shouldPersist = */
50                  true,
51                  /* scope = */
52                  BlobScope.Public,
53              )
54              val blobId: ByteArray? = blobUploader.upload()
```

The primary cryptographic risk of nonce reuse in XSalsa20-Poly1305 is keystream and Poly1305 tag-key collision when the same key is reused. Because a new random key is generated on every call, the static nonce does not produce the classic nonce-reuse vulnerability in normal operation.

That said, using a fixed nonce is a deviation from standard AEAD best practice, which calls for a unique nonce per key. The practical impact is considered informational given that fresh keys are in use and the values involved are user-generated media blobs rather than long-lived session material.

To align with standard AEAD usage and increase the security margin **we recommend** to generate a fresh random 24-byte nonce for each encryption operation and transmit or store it alongside the ciphertext, rather than using a fixed constant. If protocol constraints require a known nonce on the decryption side, derive it from a per-blob random value (e.g., a 24-byte IV included in the blob header) rather than a compile-time constant.

3.4 NOTE FIXED Latent Cross-Site Scripting Sink

> Threema addressed this finding by refactoring the affected code to not rely on the cross-site scripting prone `evaluateJavaScript` function.

During the static code analysis of the iOS codebase, a potential cross-site scripting vulnerability was identified in the `SettingsWebViewViewController` class. The application extracts the fragment portion (#) of a navigated URL, dynamically embeds it directly into a JavaScript string using `String(format:)`, and subsequently executes that string within a WebView context via `evaluateJavaScript(_:completionHandler:)`.

Threema/Settings/SettingsWebViewViewController.swift:220

Listing 6: Latent XSS sink

```
220 webView.evaluateJavaScript(String(format: "window.location.hash='%@';", anchorName))
```

This code was found in the overloaded `webView(_:decidePolicyFor:decisionHandler:)` function within the WKNavigationDelegate extension of `SettingsWebViewViewController`:

Threema/Settings/SettingsWebViewViewController.swift:197+

Listing 7: Second webView function definition

```
197 func webView(
198         _ webView: WKWebView,
199         decidePolicyFor navigationAction: WKNavigationAction,
200         decisionHandler: @escaping (WKNavigationActionPolicy) -> Void
201     ) {
202         if navigationAction.navigationType == .linkActivated, let url = navigationAction.request
    .url {
203             if let scheme = url.scheme, scheme.hasPrefix("http") || scheme.hasPrefix("https") {
204                 let config = SFSafariViewController.Configuration()
205                 let safariView = SFSafariViewController(url: url, configuration: config)
206                 present(safariView, animated: true)
207             }
208             else if let host = url.host,
209                     host == "compose",
210                     let query = url.query,
211                     query.hasPrefix("id=*SUPPORT") {
212                 addSupportContactAndOpenChat()
213             }
214             else if let anchorName = url.fragment {
```

```
215                 webView.evaluateJavaScript(String(format: "window.location.hash='%@';",
    anchorName))}
216             }
217             else {
218                 UIApplication.shared.open(url, options: [:], completionHandler: nil)
219             }
220
221             decisionHandler(.cancel)
222             return
223         }
224         decisionHandler(.allow)
225     }
```

Variable `anchorName` holds the fragment of the destination URL and is inserted with no escaping. This result in special characters like a single quote or `';alert(1)//` in the fragment will break out of the string literal, landing in the JavaScript execution context.

When assessing the exploitability, we traced the reachability of this sink back through the content-loading chain:

- The delegate and the loader (`loadURL()`) share the same controller and the same WKWebView instance (`navigationDelegate = self`).
- `self.url` resolves to the class initializer.
- All five construction call sites pass hardcoded `ThreemaURLProvider` constants, resulting in no external user controlled input reaching the sink.
- The `appendQueryItems` function only appends local app state (parameters `lang`, `version`, `platform`, and `theme`).

We conclude that, as shipped, no attacker-controlled path reaches the fragment, so the defect is latent rather than exploitable.

If the fragment ever became attacker-influenced, a new call site accepting an external URL, for example via deep-link handling, a redirect that preserves the fragment, or a refactor that loads remote content, the injected JavaScript would execute inside the WebView's origin. In a `WKWebView` that can mean:

- Reading and manipulating DOM and page state.
- Abusing any exposed `WKScriptMessageHandler` or JS bridges to reach native functionality.
- Exfiltrating content rendered in the WebView.

However, current impact is limited to code quality and is a latent risk. The unsafe pattern is one refactor away from becoming a real client-side code execution primitive.

We recommend passing the fragment value as a structured argument so it is never parsed as code. If this is not possible, the value can be JSON encoded as JSON string literals is valid JavaScript and will be correctly escaped.

4 Conclusions and recommendations

Against the MASVS baseline, the application shows strong controls in the STORAGE and CRYPTO domains. For Android, local data uses SQLCipher and encrypted preferences with randomly generated master keys, Argon2id-based passphrase protection, authenticated encryption, and Android Keystore wrapping on supported devices.

While Android relies on application-level defenses, Threema embraces Apple's tightly integrated hardware and software ecosystem for the iOS application, leaning entirely into vertical platform integration. It utilizes native Core Data and hardware-accelerated filesystem features (such as the Secure Enclave) for main data storage, while leveraging the iOS Keychain to protect critical cryptographic keys and identity credentials

NETWORK controls are also well in both operating systems, with both applications ensuring that cleartext traffic is disabled via their respective controls, certificate pinning is configured, and hostname validation is retained, and OnPrem configuration profiles are cryptographically signed before use.

The application provides good AUTH, PLATFORM, and PRIVACY protections through applock integration, sensitive-screen capture blocking, backup exclusions, notification and widget redaction, restricted file sharing, and non-exported internal components. Messaging, group, call, and device-linking flows include replay protections, authenticated sender and membership checks, and user verification during device linking.

Our recommendations can be summarized as follows:

- Review and upgrade the cryptographic binding process for biometric authentication to ensure a highly secure fallback algorithm is used.
- Ensure that the blob encryption implementation use cryptographically secure nonces for every individual encryption process.
- Refactor the latent XSS sink to ensure it cannot execute scripts, even when handling internal or trusted data.

4.1 Recommendations for future testing

The following recommendations are intended as guidance for planning future security testing activities. We assess that these areas will benefit from further assessment to strengthen the overall security posture.

To strengthen the application's security standing and eliminate technical debt, we recommend addressing the following low-severity vulnerability and informational notations made during this project to proactively harden the application against future compliance and supply chain risks.

The application reflects a high standard of security and privacy, highlighting a strong commitment to secure development. We recommend maintaining this excellent momentum by continuing a regular cadence of penetration tests and security audits.

References

[1] OWASP, "OWASP Risk Rating Methodology." `https://owasp.org/www-community/OWASP_Risk_Rating_Methodology`, 2023.