certora

Security Assessment Report

Safe

Safe{Mobile}

July 2025

Prepared for Safe Ecosystem Foundation

Table of Contents

Project Summary

Project Scope

Project Name	Repository (link)	Audited Commits	Platform
Safe{Mobile}	https://github.com/safe-global/safe-wallet-monorepo	b1577a3 (initial commit)	iOS and Android

Project Overview

This document describes the findings of the manual review of **Safe{Mobile}**. The work was undertaken from **June 30, 2025** to **July 14, 2025.**

The following files are included in our scope:

apps/mobile/src/services/key-storage/key-storage.service.ts
apps/mobile/src/services/key-storage/wallet.service.ts
apps/mobile/src/services/key-storage/types.ts
apps/mobile/src/hooks/useSign/useSign.ts
apps/mobile/src/features/ImportPrivateKey/hooks/useImportPrivateKey.ts
apps/mobile/src/hooks/useDelegate.ts
apps/mobile/src/features/ConfirmTx/components/SignTransaction/SignTransaction.tsx
apps/mobile/src/features/DataImport/hooks/useLegacyImport.ts
apps/mobile/app.config.js
apps/mobile/src/hooks/useDelegate.ts
apps/mobile/src/features/DataImport/helpers/transforms.ts
apps/mobile/src/utils/legacyData.ts
apps/mobile/src/features/AccountsSheet/AccountItem/hooks/useEditAccountItem.ts
.yarn/patches/react-native-device-crypto-npm-0.1.7-dbd2698fc4.patch

The team manually audited the respective files using code analysis and inspection tools to search for sensitive patterns and assess code structure, with particular attention to the handling and storage of key material. We also conducted runtime testing on a physical device to validate behavior and identify potential issues during execution. During the manual audit, the Certora team discovered bugs in the code, as listed on the following page.

Security Considerations

The Safe{Mobile} app is developed as a single codebase, supporting both iOS and Android platforms. It leverages React Native to maintain a single code base with unified functionality.

The mobile application handles private keys and data and performs sensitive operations, which require a highly secure design and implementation.

As requested by Safe Labs GmbH, our security assessment focused on the key materials, and our efforts focused on secure storage, handling, and usage of sensitive keys.

Audit Goals

1. Verify secure storage of sensitive key material.
2. Verify secure handling of sensitive keys.
3. Suggest improvements to raise the app's security around key material.
4. Increase the difficulty level for various attackers to carry out attacks that can leak sensitive keys:
    a. Attacker with the ability to execute malicious code on the device (malicious app, etc.)
    b. Attacker with root access to the device (malicious app on a rooted device, attacker that leverages vulnerabilities, etc.).
    c. Physical attacker, with access to the device.
    d. Network-based attacker.

Coverage and Conclusions

1. The Safe{Mobile} application is designed with strong security in mind. It leverages iOS and Android's secure storage mechanisms, carefully handles sensitive keys, employs biometric authentication for sensitive operations, and is generally well developed.
2. The audit's coverage included a comprehensive analysis of Safe{Mobile}'s key material management, focusing on the critical paths of storing, encrypting, decrypting, and fetching private keys. The review encompassed the complete key lifecycle from the initial storage in ***services/key-storage/key-storage.service.ts*** through retrieval and usage in transaction signing flows. Key areas examined included the React Native Keychain integration with DeviceCrypto for platform-specific encryption, the biometric authentication implementation in ***hooks/useSign/useSign.ts***. The audit revealed that while the cryptographic implementation demonstrates strong security practices and secure storage mechanisms, vulnerabilities exist in the access control layer and key cleanup procedures during account removal.
3. Scope limitations - the audit is focused on the secure storage and usage of the sensitive key material. There are security-sensitive areas that were not included in the scope of the audit:
    a. General Client-Side Application Security: This includes the broader security posture of the Safe{Mobile} app itself, such as insecure local data storage (beyond the audited key material), dangerous use of deep-links, use of hardcoded or insecure credentials/access tokens, and other application-level vulnerabilities that could, in extreme cases, lead to code execution within the app, potentially allowing attackers to obtain private keys or control the app in unintended ways.
    b. User Interface (UI/UX) Security: assessment of UI/UX vulnerabilities that could trick users into performing unintended actions (e.g., phishing attacks, transaction manipulation due to misleading displays or hidden elements).
    c. Third-Party Dependencies and Integrations: The security of external libraries, SDKs, APIs, and integrated third-party services or modules used as part of Safe's mobile apps.
    d. Security best practices during the software development lifecycle (SDLC): For example, using automated tools to detect potential security issues early in the SDLC, detecting significant vulnerabilities in third-party libraries, and mitigating risks such as supply chain attacks.
    e. Mobile Operating System (OS) Security Model: The audit assumes the fundamental security and integrity of the underlying mobile operating systems (iOS and

    Android), including their built-in secure storage mechanisms (e.g., iOS Secure Enclave, Android Keystore).

4. RASP (Runtime Application Self Protection) - The audited version of the app did not include a RASP solution, and we highly recommend implementing one. With that, there are a few important considerations:
    a. The quality of RASP solutions can vary widely (e.g., their ability to accurately detect rooted/jailbroken devices, the existence of malware, etc.). The RASP solution is out of the scope of this audit.
    b. As with any third-party code, embedding a RASP solution may introduce its risks.
    c. RASP solutions include many different features. For the goal of improving the overall security of Safe{Mobile}, we recommend at least:
        i. Avoid running on rooted/jailbroken devices.
        ii. Prevent app runtime tampering (hooking/patching, etc.).
        iii. Consider avoiding running on devices where malware is detected.
        iv. Protect sensitive UI screens against screen captures.
5. Although out of scope for this audit, we found (and included in this report) an issue related to dangerous deep link usage, which, under some conditions, could lead a signer to sign an unknown transaction. With that in mind, we recommend:
    a. Application-wide security posture audit
    b. Auditing significant upcoming features (local key generation, sending transactions, etc.)
    c. Performing periodic audits as the codebase changes and new functionality is introduced

Findings Summary

The table below summarizes the findings of the review, including type and severity details.

Severity	Discovered	Confirmed	Fixed
Critical	1	1	1
High	2	2	2
Medium	3	2	2
Low	2	2	1
Informational	1	1	1
Total	9	8	7

Severity Matrix

Impact	High	Medium	High	Critical
	Medium	Low	Medium	High
	Low	Low	Low	Medium
		Low	Medium	High
			Likelihood	

Detailed Findings

ID	Title	Severity	Status
C-01	Deep Link Security Not Implemented	Critical	Fixed
H-01	Private Keys Not Deleted on Account Removal	High	Fixed
H-02	Lack of RASP	High	Fixed
M-01	Private Key remains exposed in React state for too long	Medium	Disregarded
M-02	Sensitive Data Exposure in Logs and Stack Traces	Medium	Fixed
M-03	No Screenshot / Screen Recording Protection	Medium	Fixed
L-01	Lack of Memory Zeroing for Key Material	Low	Acknowledged
L-02	Data Exposure in Logs	Low	Fixed
I-01	Private Key Fingerprint Invalidation Failure (Android)	Informational	Fixed

Critical Severity Issues

C-01 Deep Link Security Not Implemented [This issue is out of the audit's scope]		
Severity: **Critical**	Impact: **High**	Likelihood: **High**
Files: N/A	Status: Fixed	

Description:

The Safe{Mobile} app implements basic deep linking through Expo Router and a custom URL scheme, but lacks security controls for intent handling. The current implementation could be vulnerable to:

- Malicious deep link attacks
- Parameter injection
- Unauthorized app triggering

The unrestricted ability to launch arbitrary app screens can be exploited for a variety of phishing and social engineering-based attacks.

apps/mobile/app.config.js Line 19:

```javascript
// Line 15: Wide open deep link scheme
scheme: 'myapp',

// No restrictions on which apps/domains can trigger deep links
// No validation of referrer sources
// No authentication requirements for deep link access
```

PoC pages and examples:

1. Triggering the transaction sign flow

Phishing page (Signer address and transaction ID were hardcoded in JS) :
https://almog-shachar.github.io/safe-wallet/f75711a26d8ba89c117439840e577d4b/safe-wallet-phishing-poc.html

Intent URI (triggered when clicking the "Verify Identity & Secure Wallet" button):

```HTML
intent://sign-transaction?txId=0xa2eef439a3e1fb47fac4b3377bcca853a8e9e501fcdd171b4f46457a5feabaf0&signerAddress=0x0CE7910b4eF24a5Caf13186fCb108c9fbcE57B08#Intent;scheme=myapp;package=global.safe.mobileapp.dev;end
```

From the command line (adb shell), this can be triggered by a malicious app:

```None
$ am start -W -a android.intent.action.VIEW  -d
"myapp:///sign-transaction?txId=0xa2eef439a3e1fb47fac4b3377bcca853a8e9e501fcdd171b4f46457a5feabaf0&signerAddress=0x0CE7910b4eF24a5Caf13186fCb108c9fbcE57B08" global.safe.mobileapp.dev
```

2. Launch of arbitrary screens:
   Onboarding, Import Legacy Data, Import Private Key, Signers, Contacts, etc.
   https://almog-shachar.github.io/safe-wallet/f75711a26d8ba89c117439840e577d4b/intent-test.html

Exploit Scenario:

- A Multi-Signer account has a multi-million-dollar transaction pending.
- The attacker needs to lure one of the account signers to sign the transaction.
- The attacker crafts a phishing page that triggers the transaction signing flow through a deep link when clicking a button.

- The attacker embeds the page in a link and sends it to the user's WhatsApp account.
- The user clicks the link, navigates to a page that displays a false message about "Account security" or shows fake transaction details.
- The user clicks the "Secure account now" button and is immediately prompted with a biometrics screen.
- The user confirms the biometric prompt, signing the pending transaction unknowingly.

Recommendations:

- Restrict deep link usage for sensitive screens. Specifically sign-transaction and import-signer/private-key screens should be restricted.
- Implement proper intent filter validation, deep link authentication, and parameter sanitization to secure the deep linking functionality.
- Consider enabling deep link triggering only from a trusted list of package names and domains.

Customer's response: Protection implemented. Fixed in PR-6107.

Fix Review: Fix confirmed. Future screens added to the app may be sensitive. Current implementation does not enforce developers to explicitly specify whether a new screen introduced should be protected or not. This is prone to developers forgetting to add a sensitive screen to "protectedScreens". We suggest using a measurement that prevents this.

High Severity Issues

H-01 Signer Private Keys Not cleared on account removal		
Severity: **High**	Impact: **High**	Likelihood: **High**
Files: features/AccountsSheet/AccountItem/hooks/useEditAccountItem.ts	Status: Fixed	

Description:

"Remove Account" feature only removes the Safe account but does not delete the associated signer private keys from the key storage. As a result:

1. Private keys remain stored in the device's secure keychain
2. Singers with their private keys are automatically available when reimporting the account using its public address.
3. No cleanup mechanism exists for removing private keys
4. Misleading behavior - users think they've removed sensitive data

Exploit Scenario:

User decides to remove his Safe Account from the app:

1. Opens the Safe{Mobile} app
2. Navigates to Settings → Remove Account
3. Confirms removal in the dialog
4. App dispatches removeSafe
5. UI updates to show the account is gone

BUT: The private key remains in the device's Keychain storage.

Later, an attacker gains access to a user's device through:

- Physical access while the user is away
- Malware that gives remote access

1. The attacker re-imports the Safe{Mobile} account using its public address.
2. Previously configured account signers are automatically available; re-importing private keys is not required.

Recommendations: While "Remove Signer" functionality is currently not implemented, it is extremely important to clear associated Signer private keys when using the "Remove Account" feature.

useEditAccountItem.ts Line 45::

```JavaScript
      dispatch(removeSafe(address))
// Clear associated signer accounts
```

Customer's response: Signer removal has been implemented, and keys are removed during Safe account deletion as well. Fixed in PR-6135.

Fix Review: Fix confirmed. feat(mobile): add remove private key option introduces wide changes enabling removal of signer's private key. Signer private key removal on Safe account deletion was implemented correctly.

H-02 Lack of RASP (Runtime Application Self-Protection)		
Severity: **High**	Impact: **High**	Likelihood: **High**
Files: N/A	Status: Fixed	

Description:

Safe{Mobile} currently lacks Runtime Application Self-Protection (RASP) capabilities, leaving the mobile app vulnerable to runtime attacks and tampering.

RASP systems commonly:

- Detect root/jailbreak environments
- Prevent tampering - code injection / hooking / dynamic code execution
- Prevent reverse engineering and debugging
- Block unauthorized network connections
- Prevent screenshot/video recording of sensitive screens

Specific Risks Without RASP:

1. Memory Attacks

- Memory dumps can extract private keys
- Process injection can modify transaction data
- Hook attacks can intercept signing functions

2. Runtime Manipulation

- Code injection can bypass security checks
- API hooking can modify transaction parameters
- Dynamic patching can disable security features

3. Environment Attacks

- Rooted/jailbroken devices can access secure storage
- Emulators can be used for automated attacks
- Debugging tools can inspect app internals

Recommendations:

Implement RASP and enable the listed checks:

1. Code integrity verification
    a. Verify code signature
    b. Check file integrity
    c. Detect code modifications
2. Runtime environment protection
    a. Detect root access
    b. Detect jailbreak
    c. Prevent debugging
    d. Detect emulator usage
    e. Prevent screen capturing
3. Memory Protections
    a. Apply memory encryption

Customer's response: Fixed in PR-6086. RASP library was added to

- prevent tampering
- detect root/jailbreak devices
- prevent reverse engineering
- detect installations from unautorized stores

Fix Review: Fix confirmed. The integration of the rasp package significantly increased the application's security posture.

Medium Severity Issues

M-01 Private Key remains exposed in React state for too long		
Severity: **Medium**	Impact: **High**	Likelihood: **Medium**
Files: ImportPrivateKey/useImportPrivateKey.ts	Status: Disregarded	

Description: During the signer key import process, the private key remains in memory for longer than required. This could lead to private key leak in certain conditions, as explained below.

The key is kept in memory in these locations:

- React state (privateKey)
- Wallet object (wallet)
- Memory from function calls

Component Lifecycle Issues

The ImportPrivateKey component can be:

- Navigated away from (but not unmounted)
- Backgrounded (app goes to background)
- Revisited (user navigates back)
- Private key remains in JavaScript memory until component unmount
- Accessible through memory analysis tools
- Survives app backgrounding/foregrounding

Timeline of Private Key Exposure

1. User enters private key → Stored in useState('')
2. User clicks "Import signer" → Private key still in state
3. Navigates to loading screen → Private key still in state (component mounted)
4. Loading completes → Private key still in state (component mounted)

5. Shows success screen → Private key still in state (component mounted)
6. User clicks "Continue" → Private key still in state (component mounted)
7. Navigation completes → FINALLY component unmounts and state is cleared

ImportPrivateKey/useImportPrivateKey.ts Line 11:

```
JavaScript
const [privateKey, setPrivateKey] = useState('')  // React state holds private key
in memory
const [wallet, setWallet] = useState<ethers.Wallet>()

const handlePrivateKeyChange = (text: string) => {
  setPrivateKey(text)  // Updates state, private key remains in memory
  const wallet = new ethers.Wallet(text)  // Creates wallet with private key in
memory
  setWallet(wallet)
}
```

Exploit Scenario:

There are 3 scenarios which can lead to private key leakage due to this issue:

1. Attacker with root privileges scans the memory
2. Physical attacker with access to the device uses advanced tools to scan the memory
3. The application crashes due to any reason, and a memory dump is created which contains the memory (and might be kept plaintext on the device, and/or sent to the vendor's servers)

Recommendations: Limit private key React state exposure window

- Clear key from state as soon as it is no longer required.
- Clear key from state if the app window is not in front.
- Make sure to zero-out key memory before freeing the key (separate issue)

Lines 38-50:

JavaScript
```
// Create a delegate for this owner
      try {
        // We don't want to fail the private key import if delegate creation fails
        // by passing null as the safe address, we are creating a delegate for the
chain and not for the safe
        const delegateResult = await createDelegate(privateKey, null)
        if (!delegateResult.success) {
          Logger.error('Failed to create delegate during private key import',
delegateResult.error)
        }
      } catch (delegateError) {
        // Log the error but continue with the import
        Logger.error('Error creating delegate during private key import',
delegateError)
      }
      // Clear private key immediately after import
      setPrivateKey('')
      setWallet(undefined)
```

Customer's response: Disregarded. The private key import should be viewed as a multistep form. It stays in memory for the time the form is rendered on the screen and needs to stay in memory for as long the user is navigating through the form screens. Once the user closes the form, it is being removed from memory.

M-02 Sensitive Data Exposure in Logs and Stack Traces		
Severity: **Medium**	Impact: **High**	Likelihood: **Medium**
Files: features/DataImport/hooks/useLegacyImport.ts utils/legacyData.ts	Status: Fixed	

Description: The decodeLegacyData function exposes sensitive data in error logs and stack traces when it fails mid-parsing, particularly:

1. Private keys in decrypted JSON data
2. Mnemonics in decrypted JSON data
3. Encrypted data in base64 format
4. Partial decrypted content in error contexts

- JSON.parse() Failure- If the decrypted data is corrupted or contains invalid JSON, the error will include the decrypted content in the stack trace.

utils/legacyData Line 29:

```JavaScript
return JSON.parse(decrypted.toString('utf8'))
```

- Unsanitized error logging - the error thrown by *decodeLegacyData* is being logged without sanitization leading to sensitive data exposure in logs.

features/DataImport/hooks/useLegacyImport.ts Line 68:

```JavaScript
Logger.error('Failed to import legacy data', e)
```

Exploit Scenario:

1. The user exports his account data from the Safe{Wallet} mobile application.
2. The user uses the data import process to migrate account data to the Safe{Mobile} application.
3. The import process fails due processing to invalid characters in the JSON object containing the account data.
4. Sensitive material such as private keys remain exposed in error logs.
5. Attackers may obtain access to the logs in various ways, such as - executing malicious code on the device, obtaining physical access to the device, or intercepting these logs as they are sent to the vendor

Recommendations:

- Implement Safe Error Handling.

features/DataImport/hooks/useLegacyImport.ts Line 68:

```javascript
Logger.error('Failed to import legacy data', e)
```

- Sanitize error logging. Do not include error objects directly.

```javascript
// In useLegacyImport.ts, replace:
Logger.error('Failed to import legacy data', e)

// With:
Logger.error('Failed to import legacy data', {
  errorType: e.constructor.name,
  message: e.message,
  // DO NOT include the error object directly
})
```

Customer's response: Fixed in PR-6090.
Fix Review: Fix confirmed.

M-03 No Screenshot / Screen Recording Protection		
Severity: **Medium**	Impact: **Medium**	Likelihood: **High**
Files: ImportPrivateKey/ImportPrivateKey.container.tsx	Status: Fixed	

Description: Private keys displayed on screen could be captured through screenshots or screen recording, and are exposed to a physical attacker

Lines 36-45: Private Key Input Field

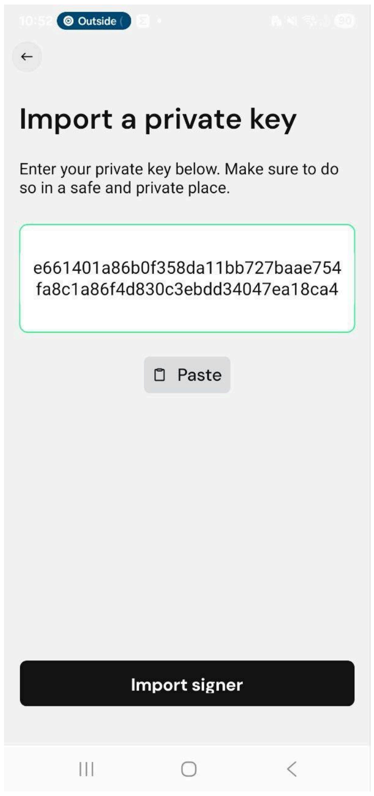
```javascript
        <View>
          <SafeInput
            height={114}
            value={privateKey}  // Raw private key exposed on screen
            onChangeText={handlePrivateKeyChange}
            placeholder="Paste here or type..."
            multiline
            success={!!wallet}
            textAlign="center"
            error={error}
          />
         </View>
```

Exploit Scenario:

1. The user enters a coffee shop, gets a table, installs the Safe{Mobile} app, configures his Safe account and uses the “Import Private Key” screen to type in his signer's private key.
2. The import process completes successfully and the user is able to use his account to sign transactions..

3. An attacker that sits behind the user can physically spy on his screen, view the raw key exposed and later import the key to his own instance of the Safe{Mobile} app.

Alternatively, malware with screenshot capabilities may leak the private key by taking a screenshot or a screen recording when the user imports a private key.


Import a private key

Enter your private key below. Make sure to do so in a safe and private place.

e661401a86b0f358da11bb727baae754
fa8c1a86f4d830c3ebdd34047ea18ca4

Recommendations:

- Disable taking screenshots while using the app.
- Use FLAG_SECURE on Android to prevent screenshots.
- Disable screen recording / mirroring while using the app.
- Mask input to reduce time window where the raw key is exposed on screen

Customer's response: Fixed in PR-6116. Import PK keys are protected.

Fix Review: Fix confirmed. Screenshot protection and Private Key input masking implemented.

Low Severity Issues

L-01 Lack of Memory Zeroing for Key Material		
Severity: **Low**	Impact: **High**	Likelihood: **Low**
Files: key-storage.service.ts useSign.ts web3/index.ts ConfirmTx/SignTransaction.tsx ImportPrivateKey/useImportPrivateKey.ts) DataImport/helpers/transforms.ts tx-sender/sign.ts	Status: Acknowledged	

Description:

Throughout the codebase, the application does not implement secure memory clearing of private keys. Keys remain in memory after operations, potentially accessible to other apps or system services.

key-storage.service.ts Line 27:

```javascript
async storePrivateKey(
 userId: string,
 privateKey: string,  // NOT CLEARED
 options: PrivateKeyStorageOptions = { requireAuthentication: true },
): Promise<void> {
 // privatKey parameter remains in memory
}
```

key-storage.service.ts Line 92:

```
JavaScript
private async storeKey(userId: string, privateKey: string, requireAuth: boolean, isEmulator:
boolean): Promise<void> {
  // privateKey parameter remains in memory
  const encryptedPrivateKey = await DeviceCrypto.encrypt(keyName, privateKey,
this.BIOMETRIC_PROMPTS.SAVE)
  // privateKey still in memory after encryption
}
```

key-storage.service.ts Line 113:

```
JavaScript
private async getKey(userId: string, requireAuth: boolean): Promise<string> {
  // ...
  const { encryptedPassword, iv } = JSON.parse(result.password)

  return DeviceCrypto.decrypt(keyName, encryptedPassword, iv,
this.BIOMETRIC_PROMPTS.STANDARD)
  // Returned private key string is NOT cleared from memory
}
```

useSign.ts Line 10:

```
JavaScript
const storePrivateKey = useCallback(
  async (
    userId: string,
    privateKey: string,  // NOT CLEARED
    options: PrivateKeyStorageOptions = { requireAuthentication: true },
  ): Promise<boolean> => {
    await keyStorageService.storePrivateKey(userId, privateKey, options)
    // privateKey parameter remains in memory
  },
  [],
)
```

useSign.ts Line 33:

```
JavaScript
const getPrivateKey = useCallback(
  async (
    userId: string,
    options: PrivateKeyStorageOptions = { requireAuthentication: true },
  ): Promise<string | undefined> => {
    return await keyStorageService.getPrivateKey(userId, options)
    // Returned private key string is NOT cleared from memory
  },
››  [],
)
```

useSign.ts Line 54:

```
JavaScript
const createMnemonicAccount = useCallback(async (mnemonic: string): Promise<HDNodeWallet |
undefined> => {
  setIsLoading(true)
  setError(null)
  try {
    return await walletService.createMnemonicAccount(mnemonic)  // MNEMONIC NOT CLEARED
  } catch (err) {
    const errorMessage = err instanceof Error ? err.message : 'Failed to create mnemonic
account'
    setError(errorMessage)
    Logger.error('createMnemonicAccount', { error: errorMessage })
    return undefined
  } finally {
    setIsLoading(false)
  }
}, [])
```

web3/index.ts Line 35:

```
JavaScript
export const createConnectedWallet = async (
  privateKey: string,  // NOT CLEARED
  activeSafe: SafeInfo,
  chain: ChainInfo,
): Promise<{ wallet: ethers.Wallet; protocolKit: Safe }> => {
  const wallet = new ethers.Wallet(privateKey)  // privateKey copied to ethers.Wallet
  // privateKey parameter and wallet.privateKey remain in memory
}
```

web3/index.ts [Line 53](#):

```javascript
let protocolKit = await Safe.init({
  provider: RPC_URL,
  signer: privateKey,  // privateKey passed to Safe, remains in memory
  safeAddress: activeSafe.address,
})

protocolKit = await protocolKit.connect({
  provider: RPC_URL,
  signer: privateKey,  // privateKey passed again, remains in memory
})
```

tx-sender/sign.ts [Line 14](#):

```javascript
export const signTx = async ({
  chain,
  activeSafe,
  txId,
  privateKey,  // NOT CLEARED
}: signTxParams): Promise<{ signature: string; safeTransactionHash: string }> => {
  const { protocolKit, wallet } = await createConnectedWallet(privateKey, activeSafe, chain)
  // privateKey remains in memory throughout function execution
}
```

ConfirmTx/SignTransaction.tsx [Line 24](#):

```javascript
const sign = useCallback(async () => {
  const privateKey = await getPrivateKey(signerAddress)  // NOT CLEARED
```

```
    const signedTx = await signTx({
      chain: activeChain as ChainInfo,
      activeSafe,
      txId,
      privateKey,  // privateKey passed to signTx, remains in memory
    })
  }, [activeChain, activeSafe, txId, signerAddress])
```

ImportPrivateKey/useImportPrivateKey.ts Line 11:

```JavaScript
const [privateKey, setPrivateKey] = useState('')  // React state holds private key in memory
const [wallet, setWallet] = useState<ethers.Wallet>()

const handlePrivateKeyChange = (text: string) => {
  setPrivateKey(text)  // Updates state, private key remains in memory
  const wallet = new ethers.Wallet(text)  // Creates wallet with private key in memory
  setWallet(wallet)
}
```

DataImport/helpers/transforms.ts Line 56:

```JavaScript
export const transformKeyData = (
  key: NonNullable<LegacyDataStructure['keys']>[0],
): { address: string; privateKey: string; signerInfo: AddressInfo } => {
  const hexPrivateKey = `0x${Buffer.from(key.key, 'base64').toString('hex')}`  // NOT CLEARED

  return {
    address: key.address,
    privateKey: hexPrivateKey,  // Returned private key remains in memory
    signerInfo,
  }
}
```

DataImport/helpers/transforms.ts [Line 100]:

```JavaScript
export const storeKeys = async (data: LegacyDataStructure, dispatch: AppDispatch):
Promise<void> => {
 if (!data.keys) {
   return
 }

 for (const key of data.keys) {   // data.keys contains base64 private keys
   try {
     const { address, privateKey, signerInfo } = transformKeyData(key)

     await storePrivateKey(address, privateKey)  // NOT
     dispatch(addSignerWithEffects(signerInfo))

     Logger.info(`Imported signer: ${address}`)
   } catch (error) {
     Logger.error(`Failed to import signer ${key.address}:`, error)
   }
 }
// data.keys not cleared from memory
}
```

useDelegate.ts Line 37:

```JavaScript
 const createDelegate = useCallback(
    async (ownerPrivateKey: string, safe: string | null = null) => {
      try {
        setIsLoading(true)
        setError(null)

        // Create the owner wallet from the provided private key
        const ownerWallet = new Wallet(ownerPrivateKey) // NOT CLEARED
        const ownerAddress = ownerWallet.address
ownerWallet.signTypedData(typedData.domain, typedData.types, typedData.message)
  // Clear Private Key From ownerWallet Object
  //...CODE...
}
```

Exploit Scenario:

1. The user signs a transaction using the Safe{Mobile} app.
2. Due to a bug in native key material handler function, the application crashes.
3. Crash log containing Register values, Memory map, Stack trace and some bytes from nearby memory addresses is created.
4. Attackers can later fetch the bug report from the device and extract key material parts from the crash dumps.

Recommendations: Implement secure memory clearing. Use native modules or ArrayBuffers to store the private key and fill with zero before releasing the memory.

Customer's response: Will not be addressed before MVP launch.

certora Safe

L-02 Data Exposure in Logs		
Severity: **Low**	Impact: **High**	Likelihood: **Low**
Files: key-storage.service.ts	Status: Fixed	

Description:

Logging of the full error object on key handling code paths can contain sensitive material, particularly:

1. Native Module Errors

- Key names/aliases that could reveal user identifiers
- Hardware security module (HSM) error codes that might leak implementation details
- Stack traces from native Android Keystore operations
- Device-specific security information (TEE/StrongBox status)

2. Android Keystore Specific Risks

- Key generation failure details that could indicate security level
- Biometric authentication error codes revealing device capabilities
- Access control violation details that might leak key existence
- Hardware security module error messages with sensitive context

key-storage-service.ts Line 87:

```javascript
 private async getOrCreateKeyAndroid(keyName: string, requireAuth: boolean, isEmulator: boolean): Promise<void> {
    try {
      await DeviceCrypto.getOrCreateSymmetricKey(keyName, {
       // ... CODE ...
    } catch (error) {
      Logger.error('Error creating symmetric encryption key:', error)    // Sanitize error
object
      throw new Error('Failed to create symmetric key')
    }
  }
```

key-storage-service .ts Line 70:

```javascript
 private async getOrCreateKeyIOS(keyName: string, requireAuth: boolean, isEmulator: boolean):
Promise<string> {
    try {
      await DeviceCrypto.getOrCreateAsymmetricKey(keyName, {
       // ... CODE ...
    } catch (error) {
      Logger.error('Error creating key:', error)    // Sanitize error object
      throw new Error('Failed to create encryption key')
    }
  }
```

key-storage-service .ts Line 49:

```javascript
  async getPrivateKey(
    userId: string,
    options: PrivateKeyStorageOptions = { requireAuthentication: true },
  ): Promise<string | undefined> {
    try {
      return await this.getKey(userId, options.requireAuthentication ?? true)
    } catch (err) {
      Logger.error('Error getting private key:', err)
      return undefined
    }
  }
```

key-storage-service .ts Line 37

```javascript
 async storePrivateKey(
    userId: string,
    privateKey: string,
    options: PrivateKeyStorageOptions = { requireAuthentication: true },
  ): Promise<void> {
    try {
             // ...CODE...
    } catch (err) {
      Logger.error('Error storing private key:', err)   // Sanitize error object
      throw new Error('Failed to store private key')
    }
  }\
```

Exploit Scenario:

- Attacker gains access to device logs
- Extracts Ethereum addresses from key names
- Targets specific users for phishing, social engineering, or physical attacks

Recommendations: Replace the current logging with sanitized error handling:

Customer's response: Fixed in PR-6178.

Fix Review: Fixed.

Informational Issues

I-01. Private Key Fingerprint Invalidation Failure (Android)

Description:
When enrolling or deleting fingerprints on Android, Signer Private Keys don't get invalidated. Instead, transaction signing with existing private keys can continue using newly configured fingerprints. This allows attackers with physical access and control over other unlocking methods configured on the device to create additional fingerprints and use them to sign transactions.

The root cause of this issue is a mistake in a third-party library - *react-native-device-crypto*. Safe app attempts to create the privateKey with the “invalidateOnNewBiometry” option set which should revoke encryption keys used to encrypt the signer private keys.
The library calls “builder.setInvalidatedByBiometricEnrollment(invalidateOnNewBiometry)” to enforce the invalidation, but it also unnecessarily calls “builder.setUserAuthenticationParameters(0, KeyProperties.AUTH_DEVICE_CREDENTIAL | KeyProperties.AUTH_BIOMETRIC_STRONG)” which cancels the invalidation.
This is documented in the Android KeyStore API.

key-storage-service.ts Line 62:

JavaScript
```javascript
  private async getOrCreateKeyIOS(keyName: string, requireAuth: boolean, isEmulator: boolean): Promise<string> {
    try {
      await DeviceCrypto.getOrCreateAsymmetricKey(keyName, {
        accessLevel: requireAuth ? (isEmulator ? 1 : 2) : 1,
        invalidateOnNewBiometry: requireAuth,
      })

      return keyName
    } catch (error) {
      Logger.error('Error creating key:', error)
      throw new Error('Failed to create encryption key')
    }
  }
```

package/android/src/main/java/com/reactnativedevicecrypto/Helpers.java Line 200:

```
JavaScript
          // Invalidate the keys if the user has registered a new biometric
            // credential. The variable "invalidatedByBiometricEnrollment" is true by
default.
            if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.N) {
              builder.setInvalidatedByBiometricEnrollment(invalidateOnNewBiometry);
            }
```

Recommendation: There are 3 options for remediation:

1. Have the authors of react-native-device-crypto library fix their implementation
2. Directly and correctly implement the logic performed by the react-native-device-crypto library
3. A quick resolution could be patching the library (Helpers.java), removing the call to: builder.setUserAuthenticationParameters(0, KeyProperties.AUTH_DEVICE_CREDENTIAL | KeyProperties.AUTH_BIOMETRIC_STRONG)

Customer's response: Fixed in PR-6178.
Fix Review: Fixed.

Disclaimer

Even though we hope this information is helpful, we provide no warranty of any kind, explicit or implied. The contents of this report should not be construed as a complete guarantee that the contract is secure in all dimensions. In no event shall Certora or any of its employees be liable for any claim, damages, or other liability, whether in an action of contract, tort, or otherwise, arising from, out of, or in connection with the results reported here.

About Certora

Certora is a Web3 security company that provides industry-leading formal verification tools and smart contract audits. Certora's flagship security product, Certora Prover, is a unique SaaS product that automatically locates even the most rare & hard-to-find bugs on your smart contracts or mathematically proves their absence. The Certora Prover plugs into your standard deployment pipeline. It is helpful for smart contract developers and security researchers during auditing and bug bounties.

Certora also provides services such as auditing, formal verification projects, and incident response.