Common Wallet Mobile Application Secure Code Review

Technical Report

Cardinal Cryptography

24 March 2025
Version: 2.0

Kudelski Security – Nagravision Sàrl

Corporate Headquarters
Kudelski Security – Nagravision Sàrl
Route de Genève, 22-24
1033 Cheseaux sur Lausanne
Switzerland

DOCUMENT PROPERTIES

Version:	2.0
File Name:	Kudelski_Security_Cardinal_Cryptography_Common_Wallet_Mobile_Application_Secure_Code_Review_v2.0.pdf
Publication Date:	24 March 2025
Confidentiality Level:	For Public Release
Document Recipients:	Jakub Kocikowski
Document Status:	Approved

TABLE OF CONTENTS

EXECUTIVE SUMMARY

Cardinal Cryptography (“the Client”) engaged Kudelski Security (“Kudelski”, “we”) to perform the Secure Code Review.

The assessment was conducted remotely by the Kudelski Security Team.

The review took place between 19 February 2025 and 06 March 2025, and focused on the following objectives:

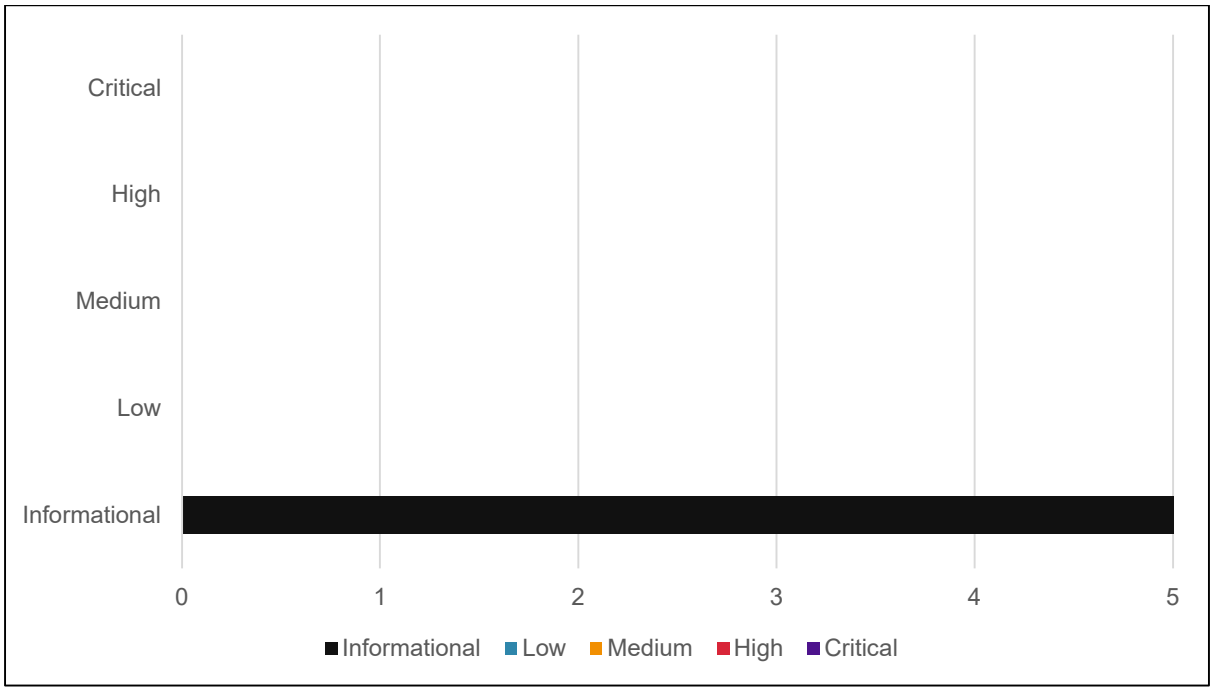
- Provide the customer with an assessment of their overall security posture and any risks that were discovered.
- To provide a professional opinion on the maturity, adequacy, and efficiency of the security measures that are in place.
- To identify potential issues and include improvement recommendations based on the result of our tests.

Key Findings

The following are the major themes and issues identified during the testing period.

These, along with other items within the findings section, should be prioritized for remediation to reduce to the risk they pose.

- Cryptographic keys declared as strings



Findings ranked by severity.

1. PROJECT SUMMARY

This report summarizes the engagement, tests performed, and findings. It also contains detailed descriptions of the discovered vulnerabilities, steps the Kudelski Security Team took to identify and validate each issue, as well as any applicable recommendations for remediation.

1.1 Context

The application being audited is Common Wallet, a mobile application self-custody wallet for Aleph Zero EVM.

1.2 Scope

The scope consisted in specific TypeScript files and folders located at:

- Git Repository: https://github.com/Cardinal-Cryptography/common-wallet-mobile
- Commit hash: `807e194c1235f2720f3841183912f17ba9c8ae71`

The files and folders in scope are:

- `common-wallet-mobile/src/setup/storage/*`
- `common-wallet-mobile/src/stores/wallets/*`
- `common-wallet-mobile/src/hooks/useBiometrics/*`
- `common-wallet-mobile/src/stores/auth/*`
- `common-wallet-mobile/src/stores/autoLock.ts`
- `common-wallet-mobile/src/utils/secureStore/*`
- `common-wallet-mobile/src/screens/SignInScreen/*`
- `common-wallet-mobile/src/components/organisms/AuthorizeModal/*`
- `common-wallet-mobile/src/screens/CreatedWalletScreen/*`
- `common-wallet-mobile/src/screens/SecurityAndPrivacyScreen/*`
- `common-wallet-mobile/src/screens/AppSplashScreen/*`

1.3 Remarks

During the code review, the following positive observations were noted regarding the scope of the engagement:

- The developers have made a careful and in-depth analysis of their project.
- The repository is well structured, and the quality of the code is good.
- Finally, we had regular and very enriching technical exchanges on various topics.

1.4 Additional Note

It is important to notice that, although we did our best in our analysis, no code audit assessment is per se guarantee of absence of vulnerabilities. Our effort was constrained by resource and time limits, along with the scope of the agreement.

In assessing the severity of some of the findings we identified, we kept in mind both the ease of exploitability and the potential damage caused by an exploit.

While assessing the severity of the findings, we considered the impact, ease of exploitability, and the probability of attack. This is a solid baseline for severity determination. Information about the severity ratings can be found in Chapter Vulnerability Scoring System of this document.

2. TECHNICAL DETAILS OF SECURITY FINDINGS

This chapter provides detailed information on each of the findings, including methods of discovery, explanation of severity determination, recommendations, and applicable references. The following table provides an overview of the findings.

- No security issue was identified during the testing period.

3. OBSERVATIONS

This chapter contains additional observations that are not directly related to the security of the code, and as such have no severity rating or remediation status summary. These observations are either minor remarks regarding good practice or design choices or related to implementation and performance. These items do not need to be remediated for what concerns security, but where applicable we include recommendations.

#	SEVERITY	TITLE	STATUS
KS–CWM–O–1	Informational	Cryptographic Keys Stored as String	Informational
KS–CWM–O–2	Informational	Default Lockout Case Should be Different	Informational
KS–CWM–O–3	Informational	Best Secure Code Practices	Informational
KS–CWM–O–4	Informational	Finding Lack of Zeroization of Sensitive Data	Informational
KS–CWM–O–5	Informational	Auto-lock uses Device Global Time	Informational

Observations overview.

3.1 KS–CWM–O–1 Cryptographic Keys Stored as String

Description

The provided codes for the mobile application stores cryptographic keys in a string. Declaring such data as a string in TypeScript could pose security risks due to the immutability of strings and their handling in memory. Strings persist until garbage collection occurs, and there is no possibility to manually clear them from memory. This could leave them vulnerable to unauthorized access. For example, accidental logging or exposure through debugging could lead to the cryptographic keys being compromised.

Evidence

```
export const encryptData = async (data: string, key: string) => {
  const cryptoKey = await getCryptoKeyFromStringKey(key);
  const iv = window.crypto.getRandomValues(new Uint8Array(12));
  const encodedData = encodeUTF8(data);

  const encryptedBuffer = await window.crypto.subtle.encrypt(
    {
      name: "AES-GCM",
      iv,
    },
    cryptoKey,
    encodedData
  );

  const encryptedData = base64Encode(encryptedBuffer);
  const ivString = base64Encode(iv);

  return `${ivString}:${encryptedData}`;
};
```

common-wallet-mobile/src/components/custom/PolkadotWebviewBridge/webview/methods/accounts/utils/wallets.ts lines 3-20

```
export const mmkvWalletsStorage: StateStorage = {
  setItem: async (key: string, value: string) => {
    try {
      const encryptionKey = await getEncryptionKey();
      const encryptedData = await encryptData(value, encryptionKey);

      walletsStorage.set(key, encryptedData);
    } catch (error) {
      // @todo: handle error
      // eslint-disable-next-line no-console
      console.error("Error storing data", error);
    }
  },
```

common-wallet-mobile/src/setup/storage/index.ts line 65-77

Affected Resources

This is valid for the whole project and requires verification through all files.

- `common-wallet-mobile/src/setup/storage/index.ts`
- `common-wallet-mobile/src/components/custom/PolkadotWebviewBridge/webview/methods/accounts/utils/wallets.ts`

Recommendation

Use Buffer for sensitive data instead of Strings. Buffers are mutable, which allows for manual clearing from memory. This finding is particularly important for cryptographic keys stored in Strings.

References

- [1] Buffer in JavaScript
- [2] String vs Buffer

3.2 KS–CWM–O–2 Default Lockout Case Should be Different

Description

In the `getLockoutTime` function, there is a switch returning the lockout time, based on the number of attempts. The default case, which will trigger for the values 0-2 and 7+, returns 0. While this is the expected return value for the case 0-2, the application is meant to lock after 6 attempts.

Evidence

```
// Returns time in milliseconds to lockout user based on number of login attempts
and labelId for translation
export const getLockoutTime = (
  loginAttempts: number
): { time: number; labelId: string } => {
  if (loginAttempts < 3) {
    return { time: 0, labelId: "global.time.xMinutes" };
  }

  switch (loginAttempts) {
    case 3:
      return { time: 30000, labelId: "global.time.xSeconds" }; // 30 seconds
    case 4:
      return { time: 300000, labelId: "global.time.xMinutes" }; // 5 minutes
    case 5:
      return { time: 900000, labelId: "global.time.xMinutes" }; // 15 minutes
    case 6:
      return { time: 3600000, labelId: "global.time.xHours" }; // 1 hour
```

```
    default:
      return { time: 0, labelId: "global.time.xMinutes" };
  }
};
```

common-wallet-mobile/src/utils/getLockoutTime.ts lines 1-21.

Affected Resources

- common-wallet-mobile/src/utils/getLockoutTime.ts lines 1-21.

Recommendation

The application should lock permanently in case of more than 6 incorrect logins, so the case 7+ should never trigger in the current version of the code. To prevent problems to occurs as the implementation evolves, it would be best to split the default case as follows. In the case of 0,1, or 2 unsuccessful logins, return 0 (no waiting time). In the case of 7+ attempts, return some “large” (for example, one hour/day/month/year).

References

- [1] CWE-447: Unimplemented or Unsupported Feature in UI

3.3 KS–CWM–O–3 Best Secure Code Practices

Deprecated Constant

In common-wallet-mobile/src/utils/secureStore/index.ts, the constant ALWAYS_THIS_DEVICE_ONLY is used. According to Expo SecureStore Documentation, this is deprecated and not secure:

> *The data in the keychain item can always be accessed regardless of whether the device is locked. This is the least secure option.*

Recommendation

Use WHEN_UNLOCKED_THIS_DEVICE_ONLY,AFTER_FIRST_UNLOCK_THIS_DEVICE_ONLY, or another alternative.

TODOs in code

In the mobile application, several comments are marked as //@todo, identifying incomplete or unimplemented features.

Recommendation

Implement the missing features and then remove the commment.

3.4 KS–CWM–O–4 Lack of Zeroization of Sensitive Data

Description

The provided code handles sensitive data, such as mnemonics and cryptographic keys, but does not include mechanisms to zeroize (clear) this data from memory after it is no longer needed. Zeroization is a security practice that ensures sensitive data is overwritten in memory to prevent it from being recovered by unauthorized parties.

The lack of memory zeroization for sensitive data can lead to potential security vulnerabilities. If the memory containing the private key is not zeroized, it may be possible for an attacker to recover the private key through memory dumps or other techniques. It is important to notice that TypeScript relies on a garbage collector to manage memory, this reduce the above mentioned risk, but does not nulify it.

Evidence

```
export const mmkvWalletsStorage: StateStorage = {
  setItem: async (key: string, value: string) => {
    try {
      const encryptionKey = await getEncryptionKey();
      const encryptedData = await encryptData(value, encryptionKey);

      walletsStorage.set(key, encryptedData);
    } catch (error) {
      // @todo: handle error
      // eslint-disable-next-line no-console
      console.error("Error storing data", error);
    }
  },
```

common-wallet-mobile/src/setup/storage/index.ts line 65-77

Affected Resources

This finding is valid for the entire projects as zeroization was never done.

- `common-wallet-mobile/src/setup/storage/index.ts`

Recommendation

To mitigate this risk, it is recommended to zeroize the memory used to store sensitive data, such as cryptographic keys after they are no longer needed.

References

- [1] CWE-226: Sensitive Information in Resource Not Removed Before Reuse
- [2] Stack Overflow, clearing memory in JavaScript

3.5 KS–CWM–O–5 Auto-lock uses Device Global Time

Description

The wallet features auto-lock, which logs out the user over a predetermined amount of time (configurable by the user). It uses `Date.now()` to compute the current time and the time until/that has elapsed since the autolock time, which is something like `Date.now() + time_interval`.The global time can be modified either by the user, or by external events (daylight savings time, travelling across timezones). For example, let's assume the user sets the lockout time to 1 hour, then sets his phone back one month. By using the global device time, he will stay logged in for 1 month and 1 hour, much longer than intended or expected.

Evidence

```
export const useAutoLockStore = create<AutoLockTimerStore>()(
  persist(
    (set, get) => ({
      ...INITIAL_VALUES,
      set: (key: AutoLockTime) => {
        const seconds = timeMap[key];

        set({
          selected: key,
          autoLockDate: dayjs().add(seconds, "seconds").toDate(),
        });
      },
      reset: () => {
        const seconds = timeMap[get().selected];

        set({
          autoLockDate: dayjs().add(seconds, "seconds").toDate(),
          isLocked: false,
        });
      },
      setIsLocked: (value: boolean) => {
        set({ isLocked: value });
      },
    }),
    {
      name: "autolock-storage",
      partialize: (state) =>
        Object.fromEntries(
          Object.entries(state).filter(([key]) => !["isLocked"].includes(key))
        ),
      storage: createJSONStorage(() => mmkvAutolockStorage),
    }
  )
);
```

common-wallet-mobile/src/src/stores/autoLock.ts lines 37-71

Affected Resources

- `common-wallet-mobile/src/src/stores/autoLock.ts` lines 37-71

Recommendation

A possible mitigation is to use a second source to verify the time, compare both values, and lock the wallet if the difference exceeds an acceptable margin. While this would not fully resolve the issue, it would add an extra hurdle for an attacker.

4. METHODOLOGY

For this engagement, Kudelski Security used a methodology that is described at a high level in this chapter. This is broken up into the following phases.

4.1 Kickoff

The Kudelski Security Team set up a kickoff meeting where project stakeholders were gathered to discuss the project as well as the responsibilities of participants. During this meeting, we verified the scope of the engagement and discussed the project activities.

4.2 Ramp-up

Ramp-up consisted of the activities necessary to gain proficiency on the particular project. This included the steps required for gaining familiarity with the codebase and technological innovations utilized.

4.3 Review

The review phase is where most of the work on the engagement was performed. In this phase we have analyzed the project for flaws and issues that could impact the security posture. The review for this project was performed using manual methods and utilizing the experience of the reviewer. No dynamic testing was performed, only the use of custom-built scripts and tools was used to assist the reviewer during the testing. We discuss our methodology in more detail in the following subsections.

Code Review

Kudelski Security Team reviewed the code within the project utilizing an appropriate IDE. During every review, the team spends considerable time working with the client to determine correct and expected functionality, business logic, and content, to ensure that findings incorporate this business logic into each description and impact. Following this discovery phase, the team works through the following categories:

- authentication (*e.g.* A07:2021, CWE-306)
- authorization and access control (*e.g.* A01:2021, CWE-862)
- auditing and logging (*e.g.* A09:2021)
- injection and tampering (*e.g.* A03:2021, CWE-20)
- configuration issues (*e.g.* A05:2021, CWE-798)
- logic flaws (*e.g.* A04:2021, CWE-190)
- cryptography (*e.g.* A02:2021)

These categories incorporate common weaknesses and vulnerabilities such as the OWASP Top 10 and MITRE Top 25.

4.4 Reporting

Kudelski Security delivered to the Client a preliminary report in PDF format that contained an executive summary, technical details, and observations about the project.

In the report we not only point out security issues identified but also observations for improvement. The findings are categorized into several buckets, according to their overall severity: **Critical**, **High**, **Medium**, **Low**.

Observations are considered to be **Informational**. Observations can also consist of code review, issues identified during the code review that are not security related, but are general best practices and steps, that can be taken to lower the attack surface of the project.

The technical details are aimed more at developers, describing the issues, the severity ranking and recommendations for mitigation.

4.5 Verify

After the preliminary findings have been delivered, we verify the fixes applied by the Client. After these fixes were verified, we updated the status of the finding in the report.

The output of this phase is the final report with any mitigated findings noted.

5. VULNERABILITY SCORING SYSTEM

Kudelski Security utilizes a custom approach when computing the vulnerability score, based primarily on the **Impact** of the vulnerability and **Likelihood** of an attack.

Each metric is assigned a ranking of either low, medium or high, based on the criteria defined below. The overall severity score is then computed as described in the next section.

Severity

Severity is the overall score of the finding, weakness or vulnerability as computed from Impact and Likelihood. Other factors, such as availability of tools and exploits, number of instances of the vulnerability and ease of exploitation might also be taken into account when computing the final severity score.

IMPACT / LIKELIHOOD	LOW	MEDIUM	HIGH
HIGH	Medium	High	High
MEDIUM	Low	Medium	High
LOW	Low	Low	Medium

Compute overall severity from Impact and Likelihood. The final severity factor might vary depending on a project's specific context and risk factors.

- **Critical** The identified issue may be immediately exploitable, causing a strong and major negative impact system-wide. They should be urgently remediated or mitigated.
- **High** The identified issue may be directly exploitable causing an immediate negative impact on the users, data, and availability of the system for multiple users.
- **Medium** The identified issue is not directly exploitable but combined with other vulnerabilities may allow for exploitation of the system or exploitation may affect singular users. These findings may also increase in severity in the future as techniques evolve.
- **Low** The identified issue is not directly exploitable but raises the attack surface of the system. This may be through leaking information that an attacker can use to increase the accuracy of their attacks.
- **Informational** findings are best practice steps that can be used to harden the application and improve processes. Informational findings are not assigned a severity score and are classified as Informational instead.

Impact

The overall effect of the vulnerability against the system or organization based on the areas of concern or affected components discussed with the client during the scoping of the engagement.

- **High** The vulnerability has a severe effect on the company and systems or has an effect within one of the primary areas of concern noted by the client.
- **Medium** It is reasonable to assume that the vulnerability would have a measurable effect on the company and systems that may cause minor financial or reputational damage.
- **Low** There is little to no effect from the vulnerability being compromised. These vulnerabilities could lead to complex attacks or create footholds used in more severe attacks.

Likelihood

The likelihood of an attacker discovering a vulnerability, exploiting it, and obtaining a foothold varies based on a variety of factors including compensating controls, location of the application, availability of commonly used exploits, difficulty of exploitation and institutional knowledge.

- **High** It is extremely likely that this vulnerability will be discovered and abused.
- **Medium** It is likely that this vulnerability will be discovered and abused by a skilled attacker.
- **Low** It is unlikely that this vulnerability will be discovered or abused when discovered.

6. CONCLUSION

The objective of this Secure Code Review was to evaluate whether there were any vulnerabilities that would put Cardinal Cryptography or its customers at risk.

The Kudelski Security Team identified 0 security issue: On average, the effort needed to mitigate these risks is estimated as low.

In order to mitigate the risks posed by this engagement's findings, the Kudelski Security Team recommends applying the following best practices:

- Avoid the use of string for cryptographic keys

An additional recommendation is to perform an offensive security assessment (Pentest) once the mobile application is completed.

Kudelski Security remains at your disposal should you have any questions or need further assistance.

Kudelski Security would like to thank Cardinal Cryptography for their trust, help and support over the course of this engagement and is looking forward to cooperating in the future.

DOCUMENT RECIPIENTS

NAME	POSITION	CONTACT INFORMATION
Adrian Dudko	Project Manager	adrian.dudko@pagepro.co
Jakub Kocikowski	Head of Product	jakub.kocikowski@cardinals.cc
Joanna Chmiel	Front-end Engineer	joanna.chmiel@pagepro.co

KUDELSKI SECURITY CONTACTS

NAME	POSITION	CONTACT INFORMATION
Jean-Sebastien Nahon	Application and Blockchain Security Practice Manager	jean-sebastien.nahon@kudelskisecurity.com
Ana Acero	Project Manager/ Operations Coordinator	ana.acero@kudelskisecurity.com

DOCUMENT HISTORY

VERSION	DATE	STATUS/ COMMENTS
1.0	6 March 2025	Draft version
1.1	12 March 2025	New version
1.2	17 March 2025	Updated version
1.3	24 March 2025	Approved version
2.0	24 March 2025	Public version