HALBORN

Make - Casper Mobile Wallet

Mobile App Pentest

Prepared by: **Halborn**

Date of Engagement: **October 4th, 2023 - October 27th, 2023**

Visit: **Halborn.com**

DOCUMENT REVISION HISTORY

VERSION	MODIFICATION	DATE
0.1	Document Creation	10/10/2023
0.2	Draft Review	10/26/2023
0.3	Draft Review	10/27/2023
1.0	Remediation Plan	11/15/2023
1.1	Remediation Plan Review	11/16/2023

CONTACTS

CONTACT	COMPANY	EMAIL
Rob Behnke	Halborn	Rob.Behnke@halborn.com
Steven Walbroehl	Halborn	Steven.Walbroehl@halborn.com
Gabi Urrutia	Halborn	Gabi.Urrutia@halborn.com
Carlos.Polop	Halborn	Carlos.Polop@halborn.com
Alvaro.Macias	Halborn	Alvaro.Macias@halborn.com

EXECUTIVE OVERVIEW

1.1 INTRODUCTION

The team at Halborn was provided a timeline for the engagement and assigned a full-time security engineer to verify the security of the assets in scope. The security engineer is a penetration testing expert with advanced knowledge in web, mobile Android and iOS, recon, discovery & infrastructure penetration testing.

The goals of our security assessments are to improve the quality of the systems we review and to target sufficient remediation to help protect users.

In summary, Halborn identified multiple issues, including weak cryptography and pincode attempts to bypass, which present high risks.

Both apps (iOS/Android) had problems with handling sensitive data, including data logged in cache, certificate pinning bypass and mnemonics dumped from memory. There were concerns related to the wallet sync functionality, which is not protected by any passcode and allows an attacker to transfer funds or delete a wallet account with physical access.

The absence of protective measures, such as jailbreak or root detection mechanisms and anti-hook and anti-debug mechanisms, were also problematic across both platforms.

Addressing these vulnerabilities is recommended to enhance the overall security and integrity of both applications.

1.2 SCOPE

IN-SCOPE:
The security assessment was scoped to:

- Make Casper Mobile Wallet Mobile Android and iOS:
  - Commit/Branch: 8865fc27f4b1afbbb326e81658f00e43e0ebcc98

1.3 TEST APPROACH & METHODOLOGY

Halborn performed a combination of manual and automated security testing to balance efficiency, timeliness, practicality, and accuracy regarding the scope of the pentest. While manual testing is recommended to uncover flaws in logic, process and implementation; automated testing techniques assist enhance coverage of the infrastructure and can quickly identify flaws in it.
The following phases and associated tools were used throughout the term of the assessment:

- Storing private keys and assets securely
- Send/Receive tokens and assets securely to another wallet
- Any attack that impacts funds, such as draining or manipulating of funds
- Application Logic Flaws
- Areas where insufficient validation allows for hostile input
- Application of cryptography to protect secrets
- Brute Force Attempts
- Input Handling
- Source code review
- Fuzzing of all input parameters
- Technology stack-specific vulnerabilities and Code Assessment
- Known vulnerabilities in 3rd party / OSS dependencies.

While the assessment was going on, the following critical components were checked:

- It has been observed that android:allowBackup=“false”. The mobile application is not found to be allowing data backups, as indicated by the android:allowBackup="false" setting in the AndroidManifest.xml file. This cannot lead to potentially sensitive information being exposed if the backup data is mishandled or accessed by malicious parties. Furthermore, this cannot lead to the unintended exposure of sensitive data if the backup is not securely stored or if it is shared across different contexts that have varying security levels.

- It has been checked that during the assessment of the android application, no information being leaked into the Android adb logs. Such logs, when not properly managed, can be accessed by malicious actors with either physical access to the device or, in certain scenarios, network access. This vulnerability can lead to privacy breaches, unauthorized access, and potentially spear-phishing attacks.
- It has been checked that no cleartext network traffic usage in android application.
- It has been checked that the mobile application no lack biometric or password authentication upon startup, despite having these security measures in place for money transactions and viewing the seed phrase. This potentially exposes sensitive user information to unauthorized users who have physical access to the device.
- It has been checked that it was not possible in Android 11 version to bypass fingerprint authentication bypass using Android-Biometric-Bypass-Update-Android-11 and ax/universal-android-biometric-bypass from Frida codeshare projects. Error triggered as “Wrapped error: User not authenticated”. Since it was unable to bypass it, it means the biometric was protected against reverse engineering using some scripts from online. Furtheremore, the client protects in their side with proper mechanism.
- It has been checked that the wallet did automatically lock upon exiting.
- It was checked that the view mechanisms (e.g view seed phrase) were properly protected using passcode / biometrics.
- It was checked that pincode and mnemonic were ciphered in the keystone before authentication.
- It was checked in the iOS application that the copy clipboard seed phrase is only for 1 minute in memory.
- It was checked that in the iOS application implemented a security advisory about view seed phrase screenshot picture. Message implemented: “Unencrypted copies are not recommended”.
- It was checked binary mitigation exploitation to check whether PIE is enabled on the iOS application binary.
- It was checked binary mitigation exploitation to check protection against memory corruption vulnerabilities that attempt to overwrite the stack, such as stack-based buffer overflows.

- It was checked on environment from iOS application that it was not stored userinfo.plist into /var/mobile/Containers/Data/Application/B9F4B5F6-3C96-498A-B93A-5A0E29601004/Documents.
- It was checked no sensitive data logged in cache in the requests made to https://event-store-api-clarity-testnet.make.services.
- It was observed that the iOS application no remains unlocked after the user exits.
- It has been checked that the Touch ID biometrics is not possible to bypass it.

RISK METHODOLOGY:

Vulnerabilities or issues observed by Halborn are ranked based on the risk assessment methodology by measuring the **LIKELIHOOD** of a security incident and the **IMPACT** should an incident occur. This framework works for communicating the characteristics and impacts of technology vulnerabilities. The quantitative model ensures repeatable and accurate measurement while enabling users to see the underlying vulnerability characteristics that were used to generate the Risk scores. For every vulnerability, a risk level will be calculated on a scale of 5 to 1 with 5 being the highest likelihood or impact.

RISK SCALE - LIKELIHOOD

5 - Almost certain an incident will occur.
4 - High probability of an incident occurring.
3 - Potential of a security incident in the long term.
2 - Low probability of an incident occurring.
1 - Very unlikely issue will cause an incident.

RISK SCALE - IMPACT

5 - May cause devastating and unrecoverable impact or loss.
4 - May cause a significant level of impact or loss.
3 - May cause a partial impact or loss to many.
2 - May cause temporary impact or loss.
1 - May cause minimal or un-noticeable impact.

The risk level is then calculated using a sum of these two values, creating a value of 10 to 1 with 10 being the highest level of security risk.

10 - CRITICAL
9 - 8 - HIGH
7 - 6 - MEDIUM
5 - 4 - LOW
3 - 1 - VERY LOW AND INFORMATIONAL

2. ASSESSMENT SUMMARY & FINDINGS OVERVIEW

CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL
0	2	2	17	1

SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
(HAL-01) WEAK CRYPTOGRAPHY DUE TO CLEARTEXT PINCODE PASSED AS PASSPHRASE TO AES.PBKDF2 FUNCTION	High	SOLVED - 11/15/2023
(HAL-02) VULNERABLE THIRD PARTY DEPENDENCIES	Low	FUTURE RELEASE
(HAL-03) DEPENDENCIES SHOULD BE PINNED TO EXACT VERSION	Low	FUTURE RELEASE
(HAL-04) ANDROID - FRIDA HOOKING ALLOWS BYPASS PINCODE ATTEMPTS	High	SOLVED - 11/15/2023
(HAL-05) TAPJACKING	Medium	SOLVED - 11/15/2023
(HAL-06) ANDROID - SENSITIVE INFORMATION EXPOSURE VIA ANDROID CLIPBOARD	Low	FUTURE RELEASE
(HAL-07) ANDROID - DUMP CLEAR TEXT MNEMONICS FROM MEMORY	Low	FUTURE RELEASE
(HAL-08) ANDROID- LACK OF ROOT DETECTION MECHANISM	Low	SOLVED - 10/12/2023
(HAL-09) ANDROID - LACK OF ANTI-HOOK ANTI-DEBUG MECHANISMS	Low	SOLVED - 11/15/2023
(HAL-10) ANDROID - BACKGROUND SCREEN CATCHING	Low	SOLVED - 11/15/2023
(HAL-11) ANDROID - TRANSFER FUNCTIONALITY SHOULD REQUIRE PASSCODE	Low	FUTURE RELEASE
(HAL-12) ANDROID - DELETE FUNCTIONALITY SHOULD REQUIRE PASSCODE	Low	SOLVED - 11/15/2023
(HAL-13) ANDROID - CERTIFICATE PINNING BYPASS	Informational	FUTURE RELEASE
(HAL-14) iOS - SENSITIVE DATA IN SNAPSHOT	Medium	SOLVED - 11/15/2023
(HAL-15) iOS - SENSITIVE INFORMATION EXPOSURE VIA IOS CLIPBOARD	Low	FUTURE RELEASE

(HAL-16) iOS - DUMP CLEAR TEXT MNEMONICS FROM MEMORY	Low	FUTURE RELEASE
(HAL-17) iOS - TRANSFER FUNCTIONALITY SHOULD REQUIRE PASSCODE	Low	FUTURE RELEASE
(HAL-18) iOS - DELETE FUNCTIONALITY SHOULD REQUIRE PASSCODE	Low	SOLVED - 11/15/2023
(HAL-19) iOS - LACK OF CERTIFICATE PINNING	Low	FUTURE RELEASE
(HAL-20) iOS - LACK OF JAILBREAK DETECTION MECHANISM ON THE iOS APPLICATION	Low	SOLVED - 11/15/2023
(HAL-21) iOS - LACK OF ANTI-HOOK ANTI-DEBUG MECHANISM ON THE APPLICATION	Low	SOLVED - 11/15/2023
(HAL-22) iOS - BACKGROUND SCREEN CATCHING	Low	SOLVED - 11/15/2023

FINDINGS & TECH DETAILS SOURCE CODE

3.1 (HAL-01) WEAK CRYPTOGRAPHY DUE TO CLEARTEXT PINCODE PASSED AS PASSPHRASE TO AES.PBKDF2 FUNCTION - HIGH

Description:

Storing sensitive data with weak cryptography format in the Android mobile Keystore and iOS mobile Keychain is a serious security vulnerability, as it allows attackers to easily access and steal the mnemonic phrase and private keys. This could lead to theft of funds or unauthorized access to user accounts.

Proof of concept:

During the assessment, it was determined that the encryptedData stored in both Keystore and Keychain employs weak cryptography, as it is susceptible to pincode brute-force attacks. Malicious actors, in possession of the retrieved encryptedData, have the potential to locally conduct pincode brute-force attempts. Upon successfully traversing the pincode iteration loop and invoking the pbkdf2 function, they can obtain a return key value that enables decryption. This decrypted content includes the vault.

Additionally, once the brute-force attack is successfully executed, the encoded mnemonic contained in both Keystore and Keychain becomes readable in plaintext form after decoding with a tool such as casperNetworkRepository.decodeSecretPhrase(mnemonic) and subsequent inspection using a Keystore analysis tool like Frida. This revelation underscores the inadequacy of the secure storage measures for the mnemonic, rendering it susceptible to retrieval by malicious actors with access to the device.

NOTE: The dynamic analysis was performed on a rooted Android device

Step 1 - Planning the attack:

1. Use the following code to execute the application and hook the functions prepared in the frida script.

Listing 1

```
frida -Uf <package.name> -l <frida-script>
```

2. Trace cipher script is used to hook the interactions between the application and the ciphers.

Listing 2

```
tracer-cipher.js
```

GitHub tracer-cipher.js

Note. It has been modified the script for hooking native library functions RCTAes.pbkdf2 and RCTAes.decrypt.

3. In the Keystore, the encryptedData was found.
4. Use the encryptedData in the following steps.

Step 2 - Code Review:

Checking in the source code, the decryption process is described as follows:

Listing 3: src/utils/aes/index.ts

```
import Aes from 'react-native-aes-crypto';
import { Buffer } from '@craftzdog/react-native-buffer';
import { scrypt } from 'react-native-fast-crypto';

const getKeyFromPincode = (pincode: string, salt: string) => Aes.
↳ pbkdf2(pincode, salt, 5000, 256);

const decryptWithKey = (encryptedData: IAesResult, key: string) =>
  Aes.decrypt(encryptedData.cipher, key, encryptedData.iv, 'aes
```

```
↳ -256-cbc');
 9
10 export const decrypt = async <T>(pincode: string, encryptedString:
↳  string) => {
11   const encryptedData: IAesResult = JSON.parse(encryptedString);
12   const key = await getKeyFromPincode(pincode, encryptedData.salt)
↳ ;
13   const data = await decryptWithKey(encryptedData, key);
14
15   return JSON.parse(data) as T;
16 };
```

Constrains:

- pincode = plain text passcode introduced by the user (e.g 111111). This value is not well-known by the attacker.
- encryptedData = It was composed by cipher, iv and salt exposed in both Keystore and Keychain.

With the aforementioned information, the next step for a malicious actor involves acquiring the knowledge of how to interface with the native library class in order to determine the feasibility of exploiting vulnerabilities in the decryption process.

Step 3 - Frida Hooking:

Primarily, it is imperative to intercept the Aes.pbkdf2 function. Employing the jadx reverse engineering tool, it was discerned that this function resides within the RCTAes class, which is contained in the com.tectiv3.aes package. Once the native function was found, it was implemented in the frida code snippet in the trace-cipher.js script.

Listing 4

```
 1  private static String pbkdf2(String str, String str2, Integer num
↳ , Integer num2) throws NoSuchAlgorithmException,
↳ InvalidKeySpecException, UnsupportedEncodingException {
 2         PKCS5S2ParametersGenerator pKCS5S2ParametersGenerator =
↳ new PKCS5S2ParametersGenerator(new SHA512Digest());
 3         pKCS5S2ParametersGenerator.init(str.getBytes("UTF_8"),
↳ str2.getBytes("UTF_8"), num.intValue());
```

```
4         return bytesToHex(((KeyParameter)
↳ pKCS5S2ParametersGenerator.generateDerivedParameters(num2.intValue
↳ ())).getKey());
5     }
```

Listing 5: Hook

```
1 function hookPbkdf2() {
2
3     let RCTAes = Java.use("com.tectiv3.aes.RCTAes");
4     RCTAes["pbkdf2"].overload('java.lang.String', 'java.lang.
↳ String', 'java.lang.Integer', 'java.lang.Integer').implementation
↳ = function (str, str2, num, num2) {
5     console.log(`RCTAes.pbkdf2 is called: str=${str}, str2=${str2
↳ }, num=${num}, num2=${num2}`);
6     let result = this["pbkdf2"](str, str2, num, num2);
7
8     console.log(`RCTAes.pbkdf2 result=${result}`);
9     return result;
10 };
11 }
```

encryptedData.salt, it facilitates the direct execution of a brute-force attack on a rooted device, thereby bypassing the pin code attempts (refer to HAL-04). It is worth noting that this attack can be potentially executed locally, which poses a significant threat to achieving a complete Account Takeover (ATO) due to the exposed nature of the encryptedData stored in the both Keystore and Keychain.

The evidence presented below demonstrates the successful hooking of the function through the use of Frida, ultimately capturing the key within the result variable.

Figure 1: RCTAes.pbkdf2

Having gained insight into the method of key retrieval, the subsequent phase entails comprehending the workings of the Aes.decrypt function. It is noteworthy that this function resides within the same RCTAes class:

Listing 6

```
private static String decrypt(String str, String str2, String str3
↳ ) throws Exception {
        if (str == null || str.length() == 0) {
            return null;
        }
        SecretKeySpec secretKeySpec = new SecretKeySpec(Hex.decode
↳ (str2), "AES");
        Cipher cipher = Cipher.getInstance("AES/CBC/PKCS7Padding")
↳ ;
        cipher.init(2, secretKeySpec, str3 == null ? emptyIvSpec :
↳  new IvParameterSpec(Hex.decode(str3)));
        return new String(cipher.doFinal(Base64.decode(str, 2)),
↳ Key.STRING_CHARSET_NAME);
    }
```

Once the native function was found, it has been implemented the frida code snippet in the trace-cipher.js script.

Listing 7

```
function decrypt() {
    let RCTAes = Java.use("com.tectiv3.aes.RCTAes");
```

```
RCTAes["decrypt"].overload('java.lang.String', 'java.lang.String',
  'java.lang.String').implementation = function (str, str2, str3) {
    console.log(`RCTAes.decrypt is called: str=${str}, str2=${str2
}, str3=${str3}`);
    let result = this["decrypt"](str, str2, str3);
    console.log(`RCTAes.decrypt result=${result}`);
    return result;
};
}
```

Figure 2: RCTAes.decrypt

Step 4 - Confirming the evidence with correct pincode:

In this phase, the attack becomes comprehensible to the malicious actors. In the context of this particular test case, the attacker employed a brute-force attack to successfully supply the correct pin code to the application. To substantiate this claim, the attacker employed identical Frida hooks on both native functions, thus validating the evidence.

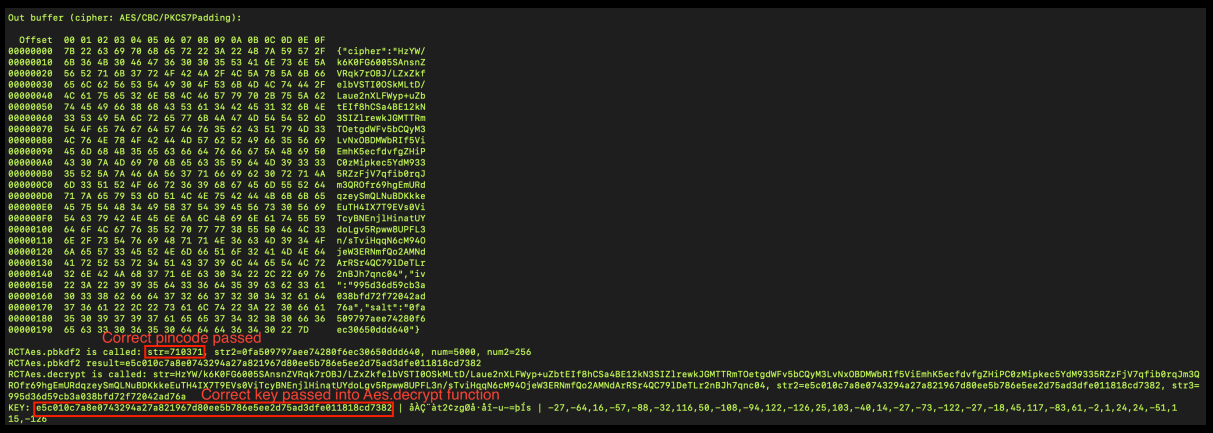


Figure 3: Correct key passed into RCTAes.decrypt.

Since the result in the RCTAes.decrypt was correct, the return value was the JSON data decrypted contained the Vault data.

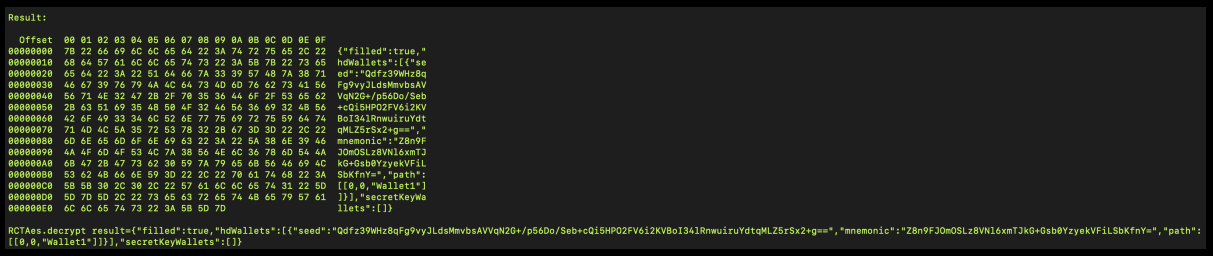


Figure 4: Successfully decryption.

The seed and mnemonic data is decrypted using the correct pincode bruteforced.

Step 5 - Getting the mnemonic to fully ATO:

After successfully decrypting the vault, the malicious user's subsequent objective was to determine the means by which they could access the mnemonic in plaintext form. The following code provides an illustrative example of how this mnemonic is utilized, with reference to data recovery carried out in the preceding step:

Listing 8

```
result={"filled":true,"hdWallets":[{"seed":"
↳ Qdfz39WHz8qFg9vyJLdsMmvbsAVVqN2G+/p56Do/Seb+
↳ cQi5HPO2FV6i2KVBoI34lRnwuiruYdtqMLZ5rSx2+g==","mnemonic":"
↳ Z8n9FJOmOSLz8VNl6xmTJkG+Gsb0YzyekVFiLSbKfnY=","path":[[0,0,"
```

```
↳ Wallet1"]]}],"secretKeyWallets":[]}
```

The `vault?.hdWallets?.[0]?.mnemonic = Z8n9FJOmOSLz8VNl6xmTJkG+Gsb0YzyekVFiLSbKfnY=` was the mnemonic encoded.

Listing 9: src/application/screens/unlockApp(use-screen.ts)

```
viewSecretPhrase: async pincode => {
        const vault = await vaultRepository.getVaultFromBackup(
↳ pincode);
        const mnemonic = vault?.hdWallets?.[0]?.mnemonic;

        navigation.goBack();

        if (mnemonic) {
          const secretPhrase = casperNetworkRepository.
↳ decodeSecretPhrase(mnemonic);
          navigation.navigate(screenNames.VIEW_SECRET_PHRASE, {
            secretPhrase,
            title: t('settings:your-secret-recovery-phrase'),
          });
        }
      }
```

Since the mnemonic is used as parameter in `casperNetworkRepository.decodeSecretPhrase(mnemonic)` the `secretPhrase` is recovered in plaintext and fully account takeover.

Listing 10: Evidence

```
-------- casperNetworkRepository.decodeSecretPhrase('
↳ Z8n9FJOmOSLz8VNl6xmTJkG+Gsb0YzyekVFiLSbKfnY=') ["guitar", "exit",
↳ "early", "chef", "glory", "multiply", "sound", "clerk", "grass", "
↳ flee", "gospel", "erode", "brief", "aspect", "bridge", "midnight",
↳  "develop", "virus", "february", "session", "sponsor", "gown", "
↳ what", "skate"]
```

In summary, it has been ascertained that a malicious actor, upon gaining access to the victim's device, possesses the capability to retrieve the `encryptedData` stored in both the Keystore / Keychain and subsequently

devise an exploit for the purpose of conducting a pin code brute-force attack. This exploit allows the malicious actor to obtain the mnemonic in its plaintext form, thereby achieving a comprehensive Account Takeover (ATO).

iOS:

An observation has revealed the capability to access encryptedData stored within the Keychain. This implies the potential for unauthorized account takeovers in the context of wallet accounts utilized within the iOS application, as per the proof of concept delineated previously.

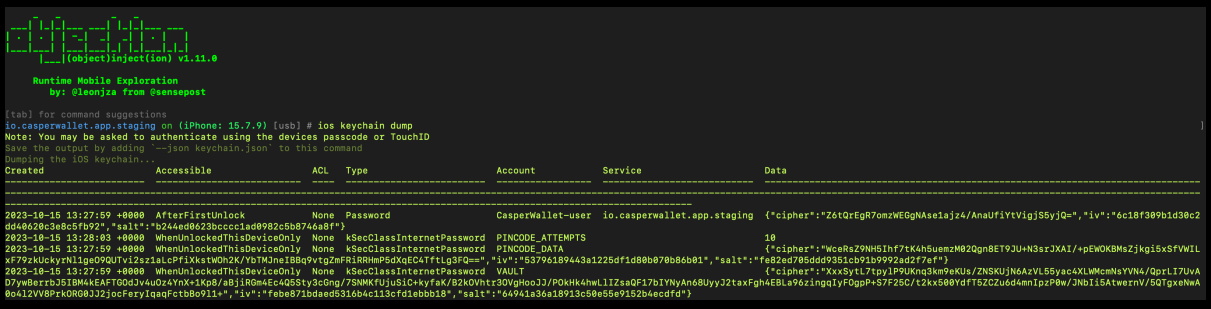


Figure 5: encryptedData.

CVSS Vector:

- CVSS:3.1/AV:P/AC:L/PR:N/UI:N/S:C/C:H/I:H/A:H

Risk Level:

Likelihood - 3
Impact - 5

Recommendation:

The time it takes to crack a 6-digit PIN depends on several factors, such as the strength of the encryption algorithm, the processing power of the computer or device used to crack the PIN, and the method used to crack the PIN (brute force, dictionary attack, etc.). However, it is generally estimated that a 6-digit PIN can be cracked in a matter of hours to days,

depending on the specific circumstances.
It is important to use a strong and complex passphrase to protect the encryption flow, and not use the cleartext pincode as a parameter on the `Aes.pbkdf2(pincode, salt, 5000, 256);` function to avoid the bruteforce attack.

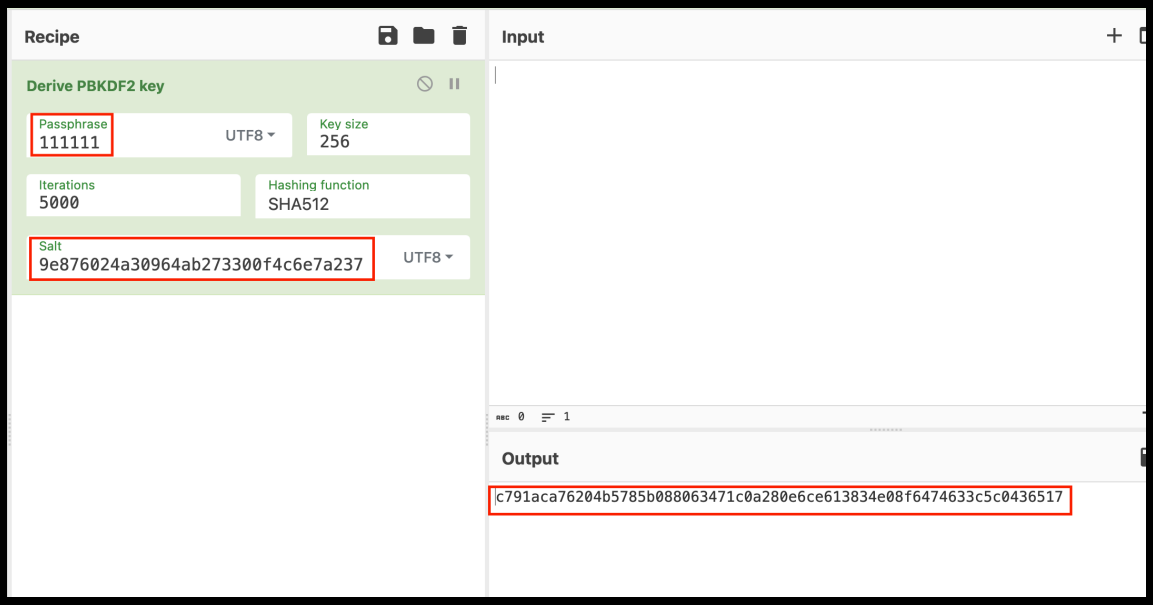
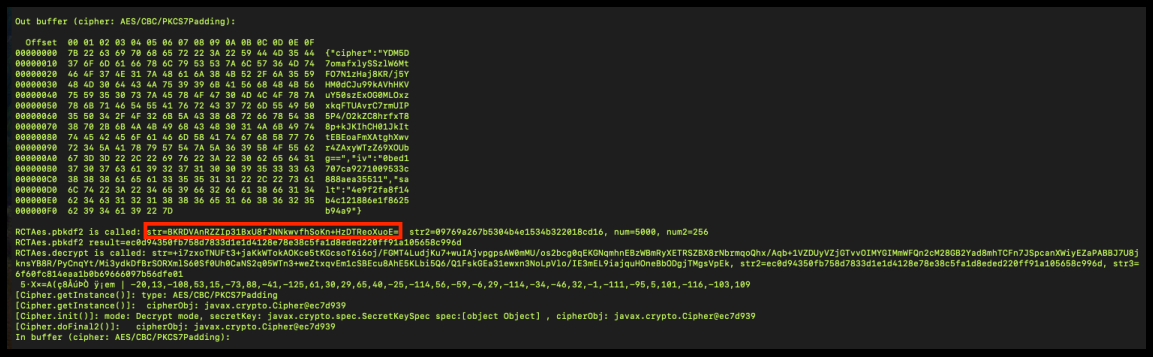


Figure 6: Insecure passphrase used.

Remediation Plan:

SOLVED: The Make solved this finding. Instead of cleartext pincode, the blake3 hash from `pincode + salt` is used.



3.2 (HAL-02) VULNERABLE THIRD PARTY DEPENDENCIES - LOW

Description:

The scoped repository uses multiple third-party dependencies. Using vulnerable third-party libraries can result in security vulnerabilities in the project that can be exploited by attackers. This can result in data breaches, theft of sensitive information, and other security issues. However, some of them were affected by public-known vulnerabilities that may pose a risk to the global application security level.

Evidences:

Run: yarn audit --groups “dependencies” on root directory:

- Critical: vm2 Sandbox Escape vulnerability
- Critical: vm2 Sandbox Escape vulnerability
- Critical: vm2 Sandbox Escape vulnerability
- Critical: Babel vulnerable to arbitrary code execution when compiling specifically crafted malicious code
- High: fast-xml-parser vulnerable to Regex Injection via Doctype Entities - fast-xml-parser

CVSS Vector:

- CVSS:3.1/AV:N/AC:H/PR:N/UI:N/S:C/C:L/I:N/A:L

Risk Level:

Likelihood - 2
Impact - 3

Recommendation:

Update all affected packages to its latest version.
It is strongly recommended to perform an automated analysis of the dependencies from the birth of the project and if they contain any security issues. Developers should be aware of this and apply any necessary mitigation measures to protect the affected application.

References:

OWASP. Vulnerable and Outdated Components
OWASP. Vulnerable Dependency Management Cheat Sheet

Remediation Plan:

PENDING: The Make will solve this finding in a future release.

3.3 (HAL-03) DEPENDENCIES SHOULD BE PINNED TO EXACT VERSION - LOW

Description:

The application contains some dependencies that are not pinned to an exact version, but are set to a supported version (x.x.x). This can potentially allow dependency attacks, as seen with the flow of events package with Copay Bitcoin Wallet.

Code Location:

Listing 11: package.json

```
"dependencies": {
    "@craftzdog/react-native-buffer": "^6.0.5",
    "@gorhom/bottom-sheet": "^4",
    "@gorhom/portal": "^1.0.14",
    "@hookform/resolvers": "^3.1.0",
    "@make-software/ces-js-parser": "^1.3.3",
    "@react-native-async-storage/async-storage": "^1.18.1",
    "@react-native-clipboard/clipboard": "^1.11.2",
    "@react-native-community/blur": "^4.3.2",
    "@react-native-masked-view/masked-view": "^0.2.9",
    "@react-navigation/bottom-tabs": "^6.5.7",
    "@react-navigation/elements": "^1.3.17",
    "@react-navigation/native": "^6.1.6",
    "@react-navigation/native-stack": "^6.9.12",
    "@reduxjs/toolkit": "^1.9.5",
    "@scure/bip32": "^1.3.0",
    "@scure/bip39": "^1.2.0",
    "@tanstack/react-query": "4.29.5",
    "apisauce": "^3.0.0",
    "casper-cep18-js-client": "^1.0.2",
    "casper-js-sdk": "^2.15.2",
    "date-fns": "^2.30.0",
    "decimal.js": "^10.4.3",
    "fast-text-encoding": "^1.0.6",
    "i18next": "^22.4.15",
    "inversify": "^6.0.1",
```

```
    "key-encoder": "^2.0.3",
    "lodash.clonedeep": "^4.5.0",
    "lodash.debounce": "^4.0.8",
    "md5": "^2.3.0",
    "react": "18.2.0",
    "react-hook-form": "^7.43.9",
    "react-i18next": "^12.2.2",
    "react-native": "0.72.5",
    "react-native-aes-crypto": "^2.1.1",
    "react-native-background-timer": "^2.4.1",
    "react-native-bootsplash": "^4.6.0",
    "react-native-code-push": "^8.0.1",
    "react-native-config": "^1.5.0",
    "react-native-device-info": "^10.6.0",
    "react-native-document-picker": "^9.0.1",
    "react-native-fast-crypto": "^2.2.0",
    "react-native-fast-image": "^8.6.3",
    "react-native-fs": "^2.20.0",
    "react-native-gesture-handler": "^2.12.0",
    "react-native-get-random-values": "^1.9.0",
    "react-native-haptic-feedback": "^2.0.3",
    "react-native-keyboard-aware-scroll-view": "^0.9.5",
    "react-native-keychain": "^8.1.1",
    "react-native-linear-gradient": "^2.7.3",
    "react-native-localize": "^3.0.0",
    "react-native-pager-view": "^6.2.0",
    "react-native-permissions": "^3.8.0",
    "react-native-popup-menu": "^0.16.1",
    "react-native-qrcode-svg": "^6.2.0",
    "react-native-quick-base64": "^2.0.6",
    "react-native-reanimated": "^3.5.4",
    "react-native-safe-area-context": "^4.5.2",
    "react-native-screens": "^3.20.0",
    "react-native-screenshot-prevent": "^1.1.1",
    "react-native-skeleton-placeholder": "^5.2.4",
    "react-native-svg": "^13.10.0",
    "react-native-tab-view": "^3.5.2",
    "react-native-vision-camera": "^2.15.4",
    "react-native-web": "^0.19.4",
    "react-native-webview": "^12.1.0",
    "react-redux": "^8.0.5",
    "redux-persist": "^6.0.0",
    "redux-saga": "^1.2.3",
    "reflect-metadata": "^0.1.13",
```

```
71      "styled-components": "^5.3.10",
72      "uuid": "^9.0.0",
73      "vision-camera-code-scanner": "^0.2.0",
74      "yup": "^1.1.1"
75    }
```

CVSS Vector:

- CVSS:3.1/AV:N/AC:H/PR:N/UI:N/S:U/C:L/I:N/A:N

Risk Level:

Likelihood - 2
Impact - 2

Recommendation:

Pinning dependencies to an exact version (=x.x.x) can reduce the chance of inadvertently introducing a malicious version of a dependency in the future.

Remediation Plan:

PENDING: The Make will solve this finding in a future release.

FINDINGS & TECH DETAILS ANDROID

4.1 (HAL-04) ANDROID - FRIDA HOOKING ALLOWS BYPASS PINCODE ATTEMPTS - HIGH

Description:

It has come to our attention that it was feasible to bypass PIN code attempts through the utilization of Frida hooks. In light of the application's absence of countermeasures such as anti-root and anti-hook mechanisms designed to mitigate reverse engineering endeavors, the potential exists for the manipulation of memory buffer values, thereby making it possible to bypass PIN code authentication procedures.

Proof of concept:

Throughout the assessment, it was discovered that the PIN code attempts, which were stored both in the Keystore and Keychain, were susceptible to being read in plain text when scrutinized with a Keystore inspection tool, such as Frida. This revelation underscores the potential for a malicious actor to tamper with the buffer memory by manipulating the clear text value, effectively allowing them to bypass the PIN code authentication process.

Step 1 - Planning the attack:

1. Use the following code to execute the application and hook the functions prepared in the frida script.

Listing 12

```
1 frida -Uf <package.name> -l <frida-script>
```

2. Trace cipher script is used to hook the interactions between the application and the ciphers.

Listing 13

```
1  tracer-cipher.js
```

GitHub tracer-chiper.js

Note. It has been modified the script for hooking native library function Cipher.update.

4. In the Keystore, the pincode attempts were found in clear text.

Step 2 – Frida Enumeration:

Initially, the attacker with physical device access employs Frida hooks to extract cipher tracing information.

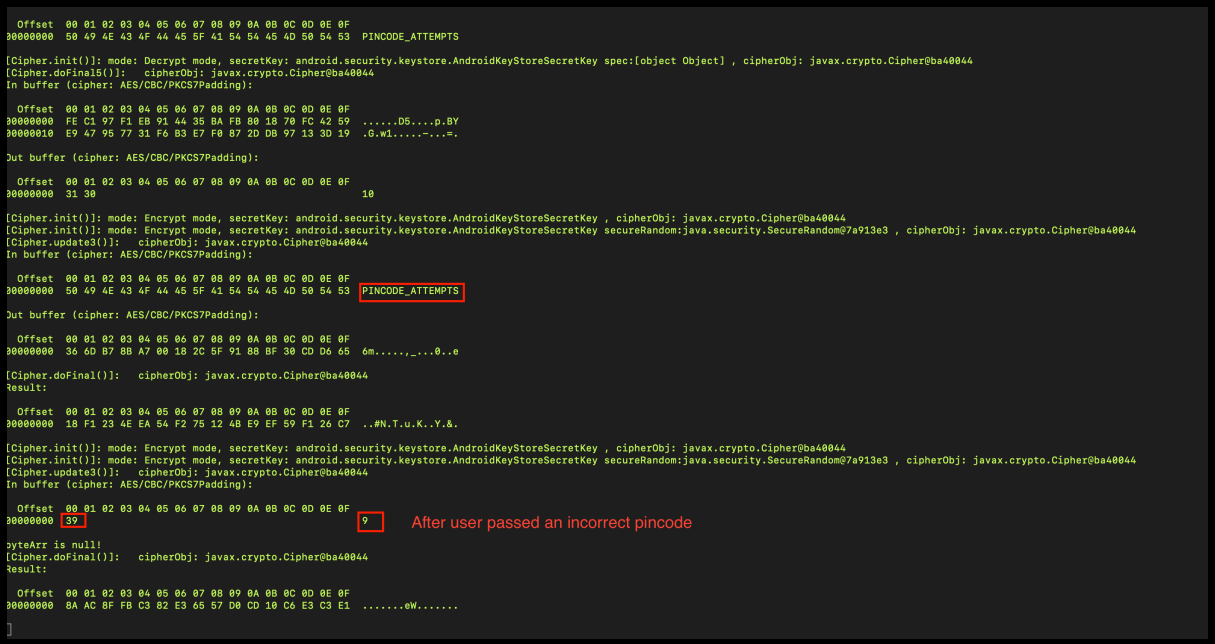
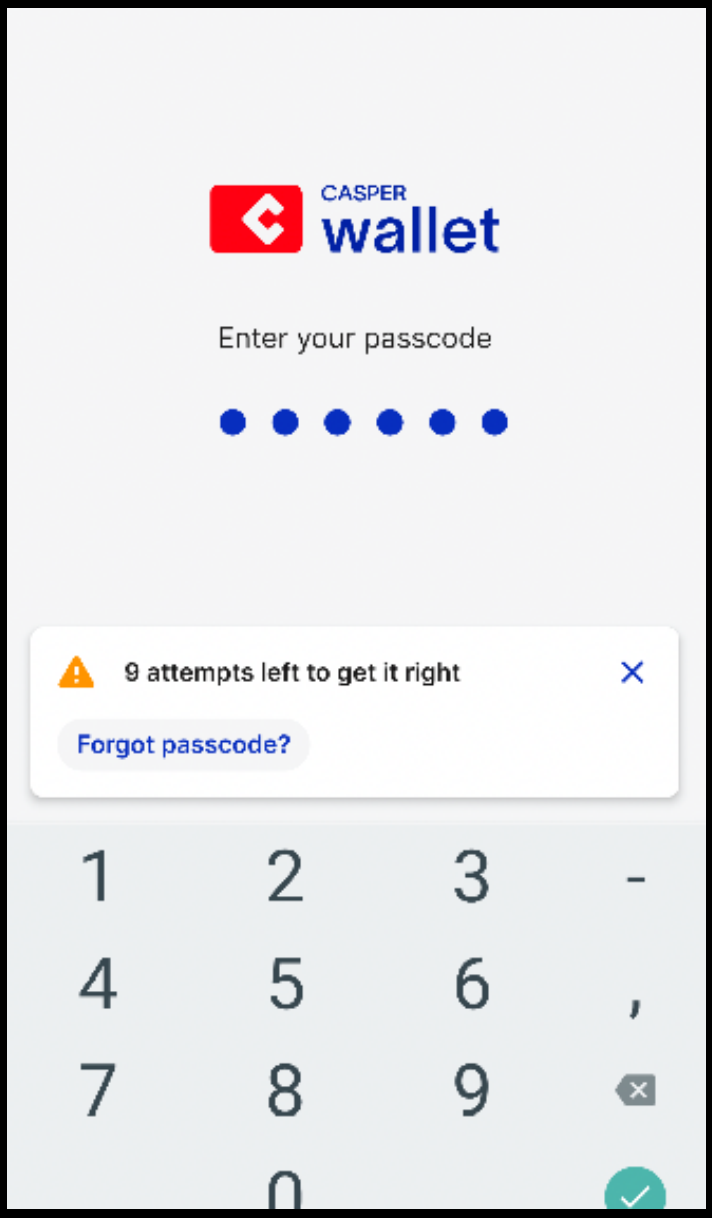


Figure 7: After user passed an incorrect pincode, 9 attemprs left to get it right.



Step 3 – Frida Hooking:
The attacker modified the "trace-cipher.js" script to bypass pincode attempts, leveraging the "Cipher.update3()" function in the Frida script. When the first byte in the "byteArr" array is equal to 55 (corresponding to the decimal value 7 in ASCII), it is altered to 57 (ASCII representation of 9). Consequently, each time the malicious user exhausts the seven allowable attempts, the counter is adjusted accordingly to the eighth.

Furthermore, it should be noted that the "Cipher.update3()" function maintains application stability even when the attacker manipulates the buffer memory to introduce a different numerical pincode attempt value.

This action does not result in the application crashing.

Listing 14

```
function hookUpdate3() {
    var cipherInit = Java.use('javax.crypto.Cipher')['update'].
↳ overload('[B', 'int', 'int');
    cipherInit.implementation = function (byteArr, a1, a2) {
        console.log("[Cipher.update3()]: " + "  cipherObj: " +
↳ this);
        dumpByteArray('In buffer (cipher: ' + this.getAlgorithm()
↳ +')', byteArr);
        if (byteArr[0] === 55) {
            byteArr[0] = 57;
        }
        var output = '';
            for (var property in byteArr) {
            output += property + ': ' + byteArr[property]+'; ';
            }
        var tmp = this.update(byteArr, a1, a2);
        dumpByteArray('Out buffer (cipher: ' + this.getAlgorithm()
↳  +')', tmp);
        return tmp;
    }
}

```

Loom Video: Proof of concept

The issue's significance becomes apparent as it affords an attacker the ability to orchestrate a chaining attack (referred to as HAL-01), thereby facilitating an unlimited series of brute force PIN code attempts aimed at acquiring the correct key that is subsequently passed to the Aes.decrypt function, ultimately leading to the decryption of the vault.

CVSS Vector:

- CVSS:3.1/AV:P/AC:L/PR:N/UI:N/S:C/C:H/I:H/A:H

Risk Level:

Likelihood - 3
Impact - 5

Recommendation:

To enhance security, it is advisable to employ a hashed value for PINCODE_ATTEMPTS. This approach ensures unpredictability and prevents users from manipulating the application to gain an unlimited number of login attempts.

Remediation Plan:

SOLVED: The Make solved this finding. Instead of cleartext attempts storage, AES encrypted storage was added.

4.2 (HAL-05) TAPJACKING - MEDIUM

Description:

Tapjacking is a type of user interface-based attack where a malicious application tricks a user into interacting with a different app's user interface (UI) than they intended. This malicious action is achieved by overlaying the legitimate app's UI with deceptive elements or a transparent layer, without the user noticing. When the user interacts with this overlay, they might perform unintended actions in the underlying genuine app.

If successfully exploited, tapjacking can lead to several harmful outcomes:

- Unauthorized Actions: An attacker can make the victim perform unintended actions within an app, like sending messages or altering settings.
- Permission Harvesting: The attacker might trick users into granting malicious apps permissions they would otherwise deny.
- Data Theft: Users could be deceived into entering credentials or sensitive information into disguised input fields.

Proof of concept:

The application exports the following components:

The activity io.casperwallet.app.MainActivity is being exported, and it is not protected by any permissions, nor it has tapjacking protection:

Listing 15

```
1         <activity android:label="@string/app_name" android:name="
↳ io.casperwallet.app.MainActivity" android:exported="true"
↳ android:launchMode="singleTask" android:screenOrientation="
↳ portrait" android:configChanges="smallestScreenSize|screenSize|
↳ uiMode|screenLayout|orientation|keyboardHidden|keyboard"
↳ android:windowSoftInputMode="adjustPan">
```

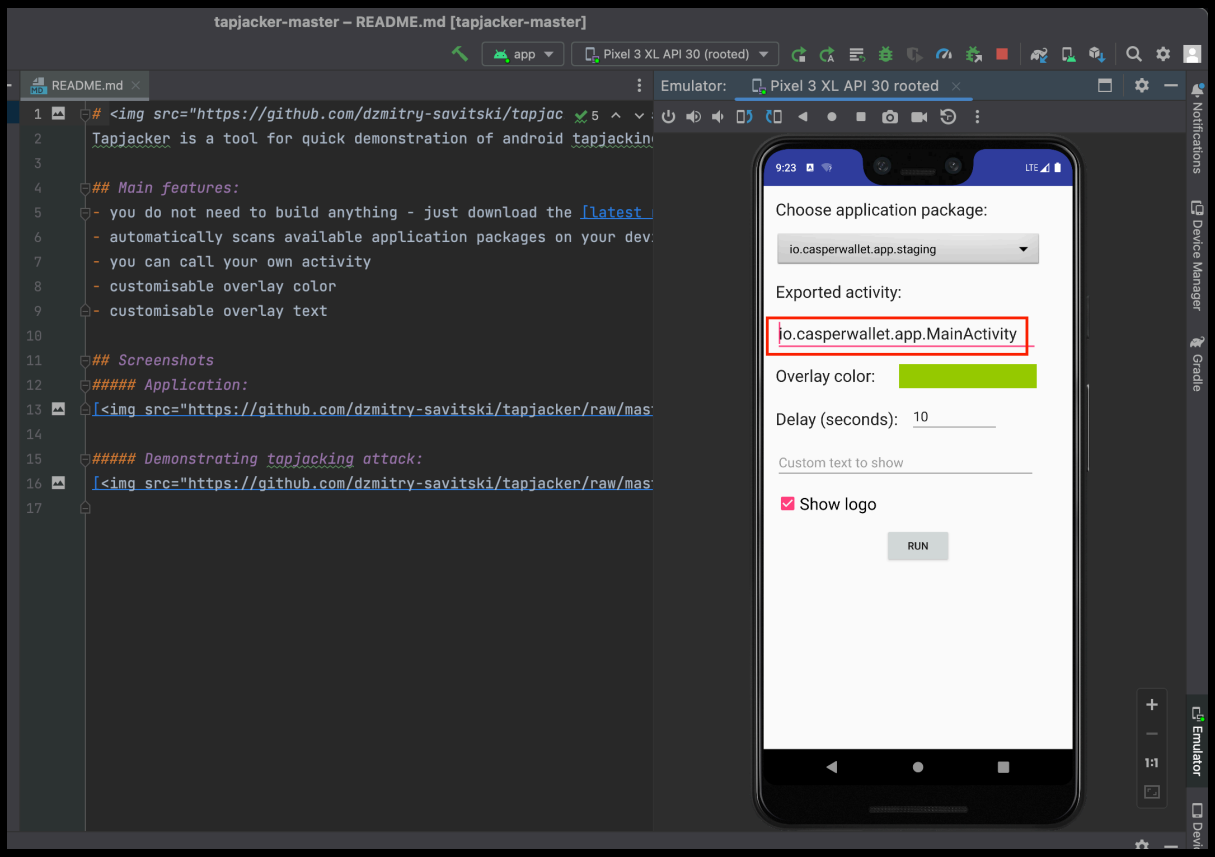


Figure 8: Tapjacking application used.

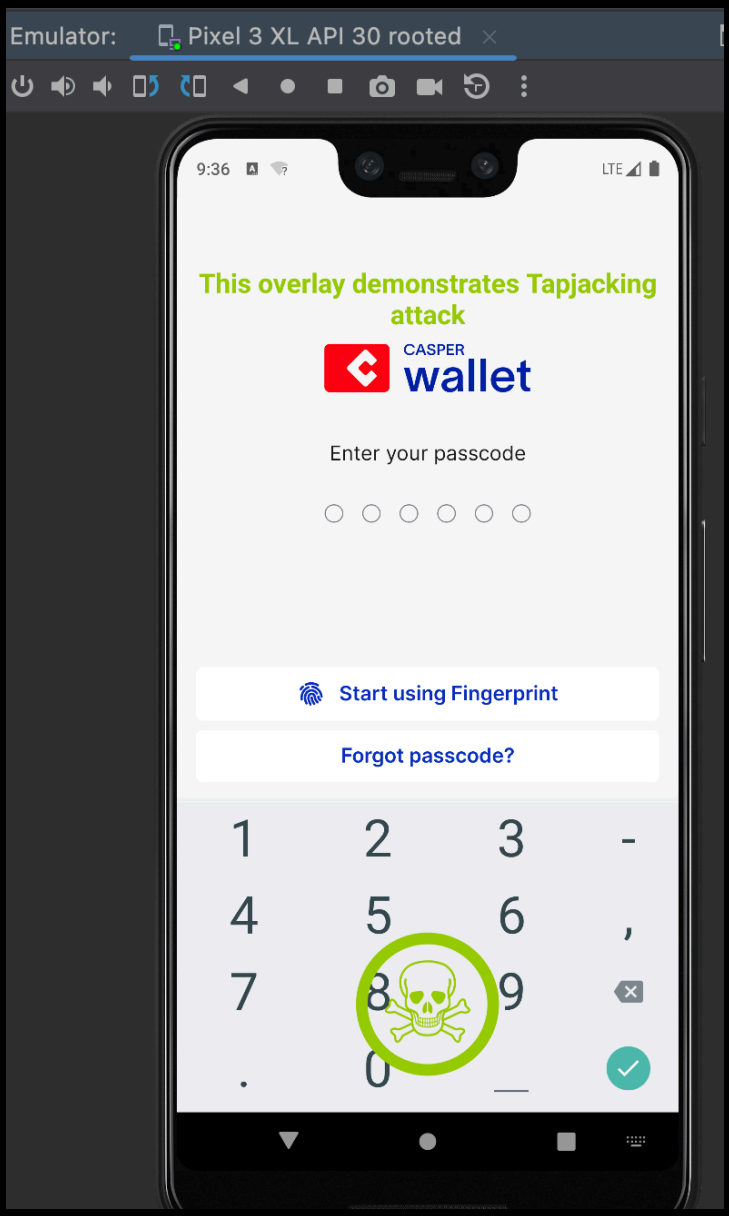


Figure 9: Succesfully injection into application.

The code used to create the exploit application was from: Tapjacker. It can be used from Carlos Polop the following application as well: Tapjacking-ExportedActivity

CVSS Vector:

- CVSS:3.1/AV:P/AC:L/PR:L/UI:R/S:U/C:L/I:H/A:N

Risk Level:

Likelihood - 3
Impact - 3

Recommendation:

It is recommended to stop exporting the activity if possible. If the activity needs to be exported, configure the activity with android:filterTouchesWhenObscured="true" in the AndroidManifest to prevent exploitation of the vulnerability or require a minimum version 31.

Remediation Plan:

SOLVED: The Make solved this finding. Added tapjacking prevention to MainActivity.java.

Listing 16

```
1 // get the root view and activate touch filtering to prevent tap
↳ jacking
2     View v = findViewById(android.R.id.content);
3     v.setFilterTouchesWhenObscured(true);
```

4.3 (HAL-06) ANDROID - SENSITIVE INFORMATION EXPOSURE VIA ANDROID CLIPBOARD - LOW

Description:

The Android application allows users to copy the seed phrase to the Android clipboard. This can introduce potential security risks, as other malicious apps could potentially access and compromise sensitive information stored in the clipboard.

If a malicious actor gains access to the user's seed phrase, they could potentially gain access to the user's cryptocurrency wallets or other accounts that rely on that seed phrase for authentication. This could result in the theft of funds or other sensitive data, as well as damage to the user's reputation and trust in your application.

Proof of concept:

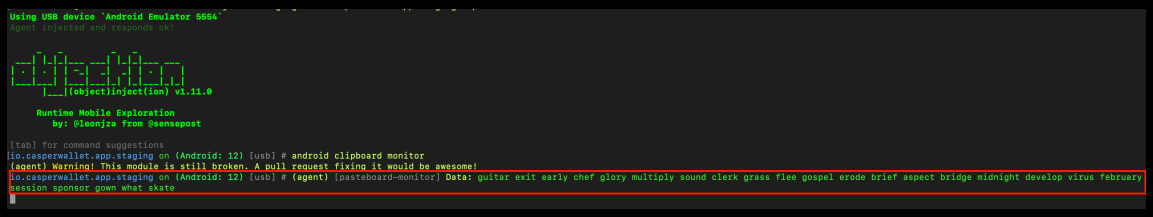


Figure 10: Mnemonic phrase captured from Android Clipboard

Risk Level:

Likelihood - 2
Impact - 3

CVSS Vector:

- CVSS:3.1/AV:N/AC:H/PR:H/UI:R/S:U/C:H/I:H/A:N

Recommendation:

To mitigate this vulnerability, it is recommended to consider disabling the ability to copy the seed phrase to the clipboard. Instead, alternative methods can be implemented for users to securely store their seed phrase, such as:

- Allowing users to export the seed phrase as an encrypted file, which can then be stored on an external storage device or a secure cloud storage service.
- Integrating your app with hardware wallets or other secure storage solutions to store the seed phrase.
- Encouraging users to write down the seed phrase on a piece of paper and store it in a secure location.

Reference:

- OWASP Sensitive Data Exposure

Remediation Plan:

PENDING: The Make will solve this finding in a future release.

4.4 (HAL-07) ANDROID - DUMP CLEAR TEXT MNEMONICS FROM MEMORY - LOW

Description:

It was possible to dump the mnemonic phrase from the memory of the application and find the mnemonic pattern with regex. As there were no checks against the rooted devices, it makes possible to dump the running memory from the app and extract the mnemonics from it.

Note: In the application, Fridump was used to dump memory. Our goal was to dump the memory of the application and find the mnemonic pattern with regex.

Proof of concept:

Using fridump, dump the memory of the application running. Find the mnemonic pattern with regex over the dumped files.

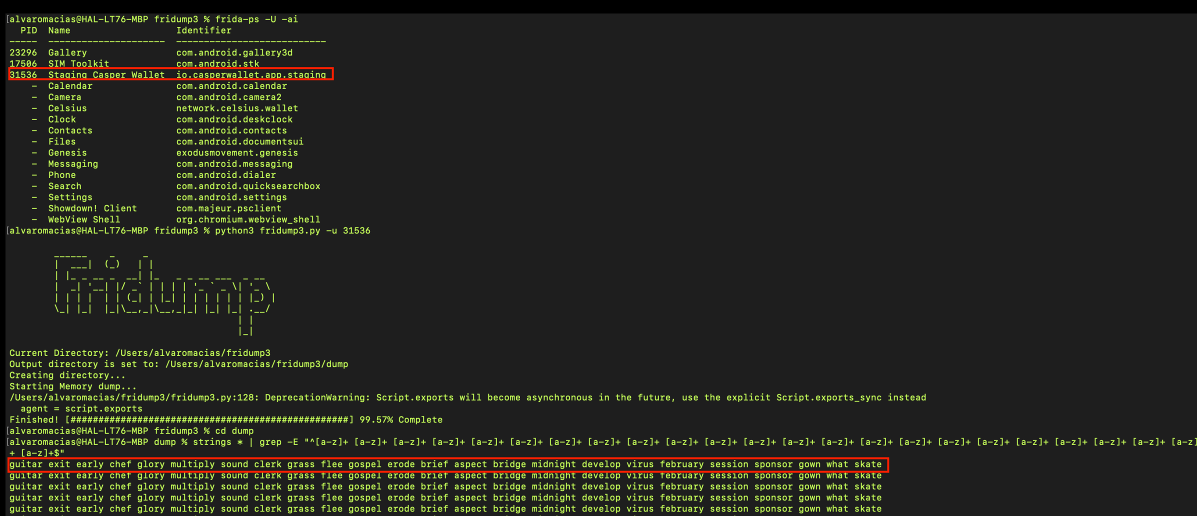


Figure 11: Memory dump of wallet application

Risk Level:

Likelihood - 1

Impact - 3

CVSS Vector:

- CVSS:3.1/AV:P/AC:H/PR:H/UI:R/S:U/C:H/I:N/A:N

Recommendation:

It is recommended to have protected measures against jailbreak detection and some open-source tools like Frida in the application to prevent loading/running the application if these tools are detected on the device. Also, the crypto wallet application should erase the mnemonics from the memory after a wallet delete. This ensures that users can maintain the protection of the wallet during regular use.

Reference:

- OWASP Tampering and Reverse Engineering
- OWASP Root Detection Methods
- OWASP Android Lack of binary protections

Remediation Plan:

PENDING: The Make will solve this finding in a future release.

4.5 (HAL-08) ANDROID- LACK OF ROOT DETECTION MECHANISM - LOW

Description:

Anti-root mechanisms are not used in the Android applications. These mechanisms can help mitigate reverse engineering, application modification, and unauthorized versions of mobile applications to some extent, but few if any will be completely successful against a determined adversary. However, they can be used as part of a defense-in-depth strategy that seeks to minimize the impact and likelihood of such an attack, along with binary patching, local resource modification, method hooking, method swizzling, and heap modification.

Proof of concept:

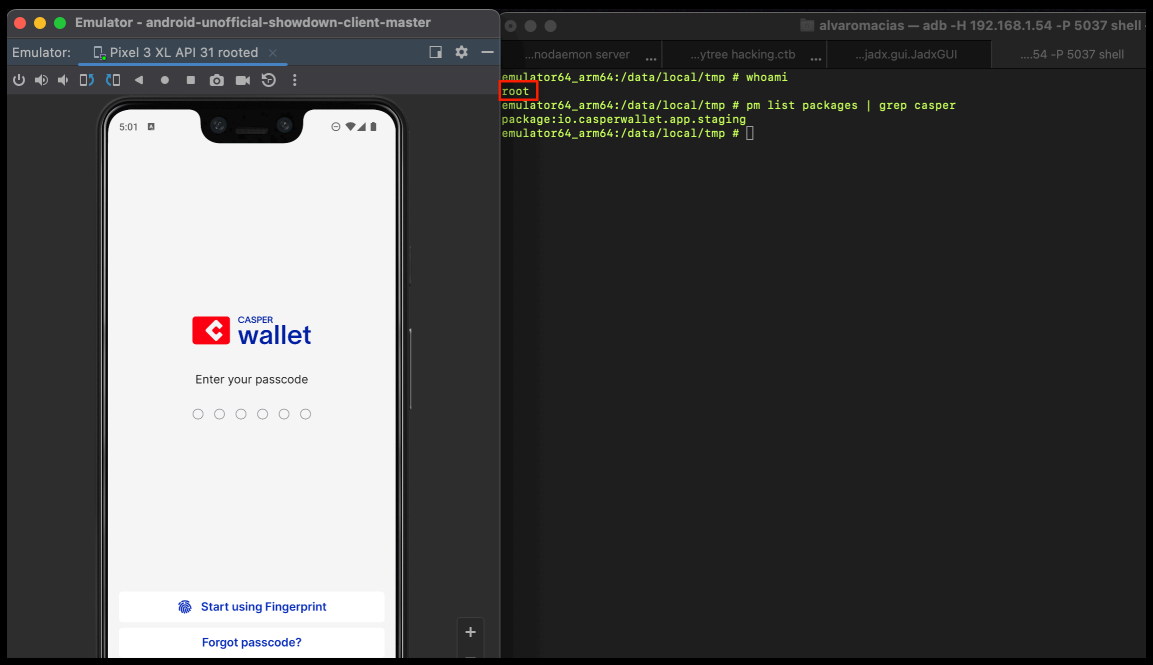


Figure 12: Application running on rooted device

CVSS Vector:

- CVSS:3.1/AV:P/AC:H/PR:H/UI:R/S:C/C:L/I:L/A:N

Risk Level:

Likelihood - 2
Impact - 2

Recommendation:

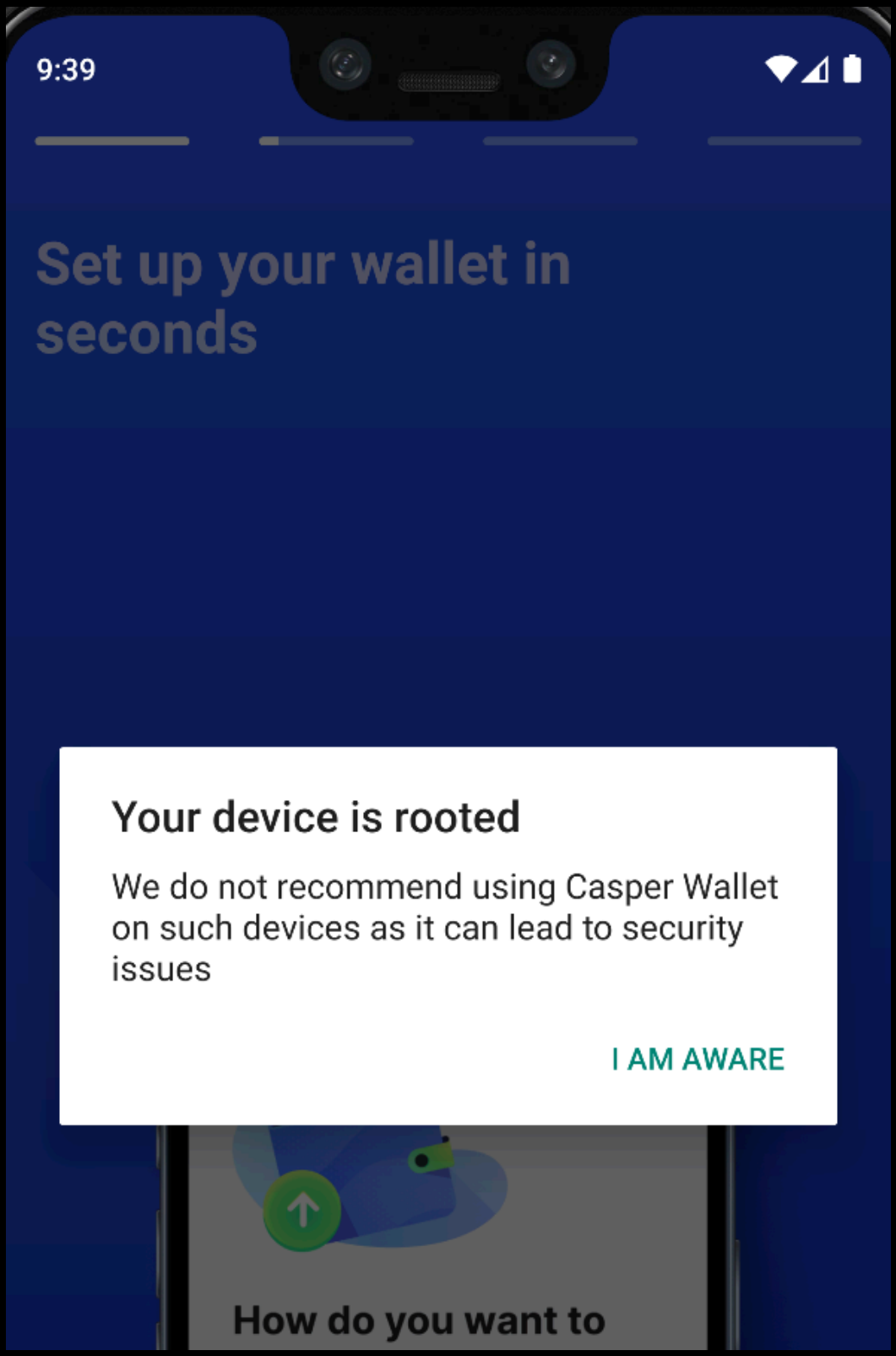
The application should detect rooting methods to prevent modifications to the app. As a security best practice, it is recommended to implement a mechanism to check the rooted status of the mobile device. This can be done either manually by implementing a custom solution or using libraries already built for this purpose. This can be done by searching for commonly known files and locations, checking file permissions and attempting to find common rooting services like SuperSU, Magisk or OpenSSH, for example.

References:

- OWASP Tampering and Reverse Engineering
- OWASP Root Detection Methods
- OWASP Android Lack of binary protections
- Android SafetyNet Attestation API

Remediation Plan:

SOLVED: The Make solved this finding. Added notification to the user of increased risks of using the app on a rooted device.


9:39
Set up your wallet in seconds
Your device is rooted
We do not recommend using Casper Wallet on such devices as it can lead to security issues
I AM AWARE
How do you want to

4.6 (HAL-09) ANDROID - LACK OF ANTI-HOOK ANTI-DEBUG MECHANISMS - LOW

Description:

The tested application does not have any security features or mechanisms to prevent malicious actions, Anti Hook and Anti Debug mechanisms.

Proof of concept:

- Install Frida on the rooted phone. Frida for Android
- Use the Objection Tool to investigate the Anti-Hook mechanisms in the application. Objection
- Use the following command in the objection tool to investigate the rooted device.

Listing 17

```
objection -g <package-name> explore --startup-command "android
↳ hooking watch class_method <package-name>.MainActivity.onCreate"
```

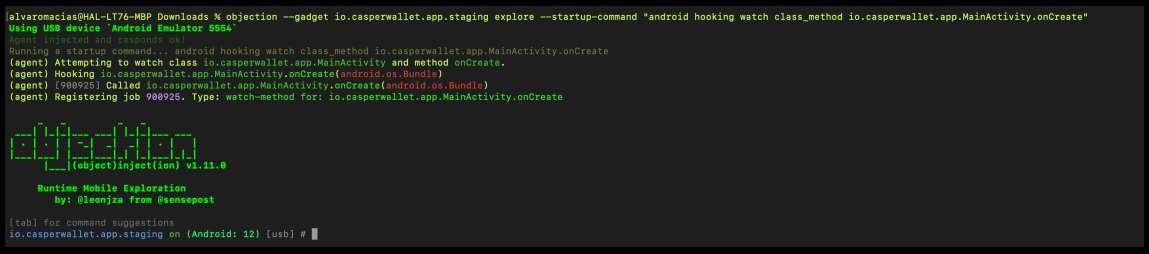


Figure 13: hooking onCreate method

You can see that the application does not terminate; therefore the application does not have anti-hook or anti-tampering mechanisms.

CVSS Vector:

- CVSS:3.1/AV:P/AC:H/PR:H/UI:R/S:C/C:L/I:L/A:N

Risk Level:

Likelihood - 2
Impact - 2

Recommendation:

Anti-Debug, Anti-Hook and Integrity Check mechanism (completed in the native code), will protect against injection of various types of scripts into it, i.e., Frida Gadgets. The application should not allow modifications in its operation.

References:

- OWASP Reverse Engineering and Tampering
- AppKnox anti debugging techniques

Remediation Plan:

SOLVED: The Make solved this finding. Added checks for hooks and debugs mods. In case of detection, a blocking screen is shown.

4.7 (HAL-10) ANDROID - BACKGROUND SCREEN CATCHING - LOW

Description:

Manufacturers want to provide device users with an aesthetically pleasing experience at application startup and exit, so they introduced the screenshot-saving feature for use when the application has been executed in the background. This feature may pose a security risk since a user can deliberately screenshot the application while sensitive data is being displayed. A malicious application that is running on the device and able to continuously capture the screen may also expose data. Screenshots are written to local storage, from which they may be recovered by a rogue application (if the device is rooted) or someone who has stolen the device.

For example, capturing a screenshot of a banking application may reveal information about the user's account, credit, transactions, and so on.

In this specific case of the android application, the mnemonic phrase can be captured while the application is not active.

Proof of concept:

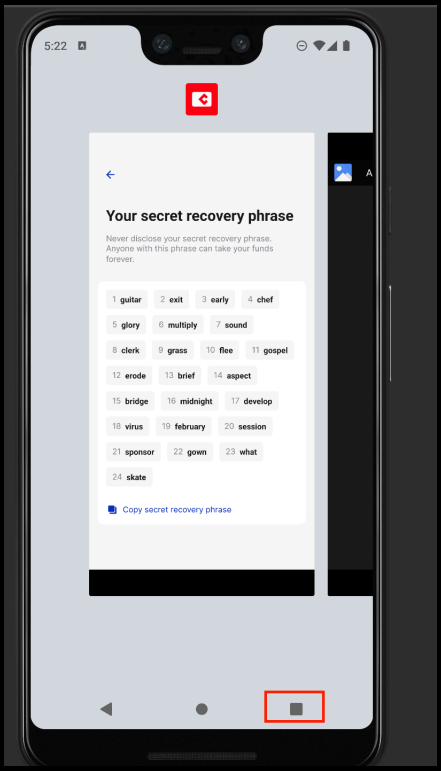


Figure 14: Background screen cache

Risk Level:

Likelihood - 1
Impact - 4

CVSS Vector:

- CVSS:3.1/AV:P/AC:L/PR:H/UI:N/S:U/C:L/I:N/A:N

Recommendation:

This vulnerability represents a minimal exposure to exploitation. Only the users of the mobile devices to which the attacker has access are affected by this vulnerability.

As a best practice, consider preventing run background screen caching if

the application displays sensitive data.

Remediation Plan:

SOLVED: The Make solved this finding. Added FLAG_SECURE and `setRecentsScreenshotEnabled(false)` to prevent screen catching.

4.8 (HAL-11) ANDROID - TRANSFER FUNCTIONALITY SHOULD REQUIRE PASSCODE - LOW

Description:

The crypto wallet application did not have the passcode protection implemented in the transfer wallet functionality.

The actions without rollback should be restricted with passcode/biometric auth to ensure that the user knows the risks and anyone other than the user could not use those functions.

Proof of concept:

- Go to Delete Account functionality located in Screens -> Send Tokens -> Confirming Sending.
- No protection is displayed in the transfer.

Risk Level:

Likelihood - 1
Impact - 3

CVSS Vector:

- CVSS:3.1/AV:P/AC:H/PR:L/UI:N/S:U/C:N/I:N/A:L

Recommendation:

Secure the functionality with the passcode to ensure that anyone other than the user could not use it.

Remediation Plan:

PENDING: The Make will solve this finding in a future release.

4.9 (HAL-12) ANDROID - DELETE FUNCTIONALITY SHOULD REQUIRE PASSCODE - LOW

Description:

The crypto wallet application did not have the passcode protection implemented in the delete wallet functionality.

The actions without rollback should be restricted with passcode/biometric auth to ensure that the user knows the risks and anyone other than the user could not use those functions.

Proof of concept:

- Go to Delete Account functionality located in Screens -> Accounts -> Delete Account.
- No protection is displayed in the delete account.

Risk Level:

Likelihood - 1
Impact - 3

CVSS Vector:

- CVSS:3.1/AV:P/AC:H/PR:L/UI:N/S:U/C:N/I:N/A:L

Recommendation:

Secure the functionality with the passcode to ensure that anyone other than the user could not use it.

Remediation Plan:

SOLVED: The Make solved this finding. Asked for pincode before opening delete wallet screen.

4.10 (HAL-13) ANDROID - CERTIFICATE PINNING BYPASS - INFORMATIONAL

Description:

Certificate pinning is the process of associating the backend server with a particular X.509 certificate or public key, instead of accepting any certificate signed by a trusted certificate authority (CA). After storing ("pinning") the server's certificate or public key, the mobile app will subsequently connect only to the known server. Withdrawing trust from external CAs reduces the attack surface (after all, there are many cases of CAs being compromised or tricked into issuing certificates to impostors).

The certificate can be pinned and hardcoded in the app or retrieved at the time the app first connects to the backend. In the latter case, the certificate is associated ("pinned" to) the host when the host is first seen. This alternative is less secure because attackers intercepting the initial connection can inject their certificates.

The target application has not correctly implemented SSL pinning when establishing a trusted connection between the mobile applications and the back-end web services. Without enforcing SSL pinning, an attacker could man-in-the-middle the connection between mobile applications and back-end web services. This allows an attacker to sniff user credentials, session ID, etc. Certificate pinning is used in modern applications to prevent users from intercepting and analyzing HTTP traffic. Using this method, an application can verify the server's certificate and, in case there is a Man-in-The-Middle, not trust any other certificate than the one stored as default. There are many ways to perform this security countermeasure, and taking it in place does not ensure that a motivated attacker will be able to bypass it in time, but it does represent the first wall of defense against HTTP attacks.

However, in the case of **Make Casper Wallet** android, although it implements SSL pinning, it uses methods with common names and does not implement anti-hooking mechanisms, which allows attackers to bypass this protection

and make it possible to steal the authentication token used in requests as well.

Proof of concept:

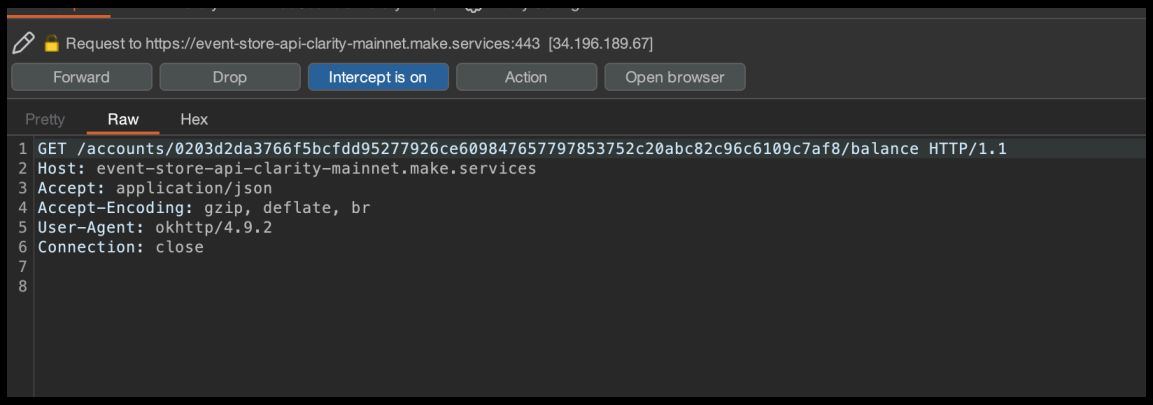


Figure 15: Request intercepted.

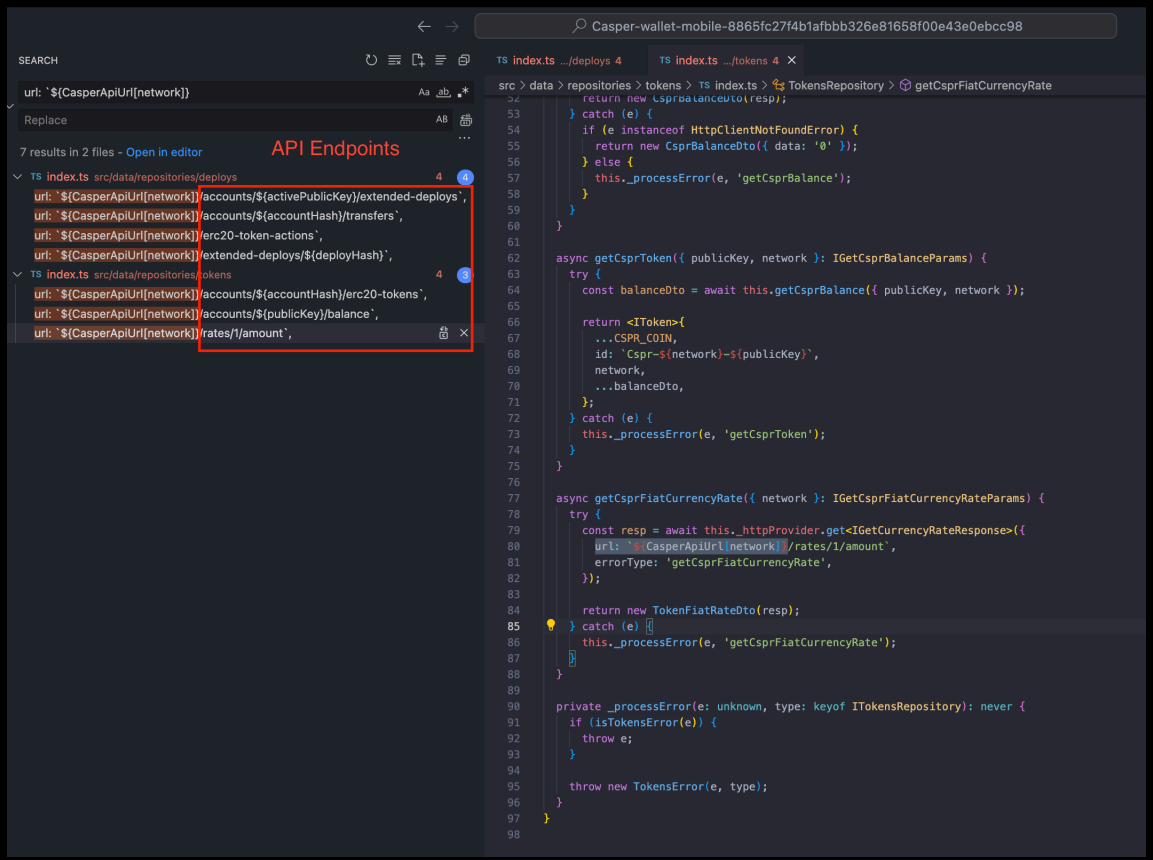


Figure 16: API endpoints from source code.

1. Connect to the application using Frida and Objection

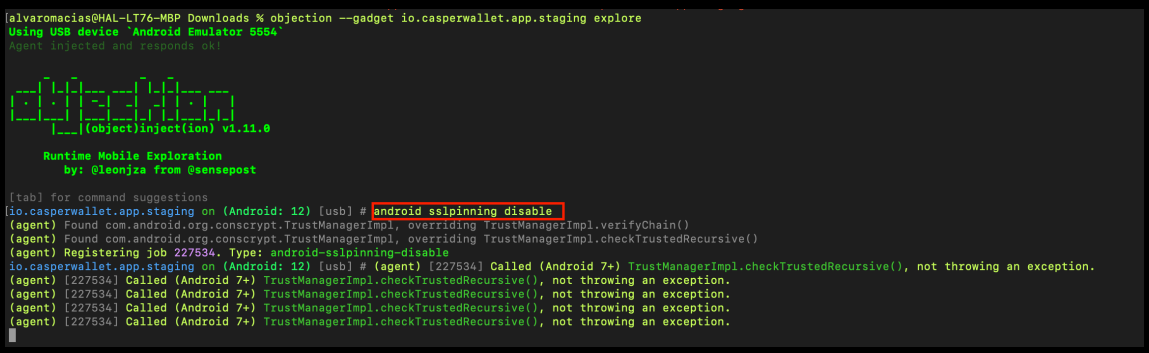
Listing 18

```
objection --gadget <package-name> explore
```

2. Set the automatic certificate pinning bypass implemented by objection

Listing 19

```
android sslpinning disable
```



As it can be seen above, the certificatePinner.check() method of OkHTTP and CheckTrustedRecursive() of TrustManagerImpl are triggered and modified at runtime:

Risk Level:

Likelihood - 2
Impact - 1

Recommendation:

It is recommended to prevent these actions by enforcing anti-tampering and anti-debugging mechanisms. This vulnerability is related to rooting detection and anti-debug and anti-tampering (following). Having methods that cannot be triggered by name and anti-hooking, debugging and rooting detection mechanisms should be enough to start preventing certificate pinning bypass. Additionally, an application should follow the following best practices:

- Set an HTTP Public Key Pinning (HPKP) policy that is communicated to the client application and/or supports HPKP in the client application, if applicable.
- Use of security frameworks such as Android SafetyNet to avoid method hooking and rooted devices

Reference:

- Android Certificate Pinning
- OWASP Pinning Cheat Sheet
- Android Code Obfuscation
- Guidelines Towards Secure SSL Pinning in Mobile Applications

Remediation Plan:

PENDING: The Make will solve this finding in a future release.

FINDINGS & TECH DETAILS IOS

5.1 (HAL-14) iOS - SENSITIVE DATA IN SNAPSHOT - MEDIUM

Description:

During the analysis, it has been observed that sensitive data like seed-phrase and private-key can be saved as a snapshot in iOS. Whenever you press the home button, iOS takes a snapshot of the current screen to be able to do the transition to the application in a much smoother way. However, if sensitive data is present in the current screen, it will be saved in the image (which persists across reboots). These are the snapshots that you can also access, by double tapping the home screen to switch between apps.
Unless the iPhone is jailbroken, the attacker needs to have access to the device unblocked to see these screenshots. By default, the last snapshot is stored in the application's sandbox in Library/Caches/Snapshots/ or Library/SplashBoard/Snapshots folder and can leak the sensitive details related to the user's wallet.

Proof of concept:

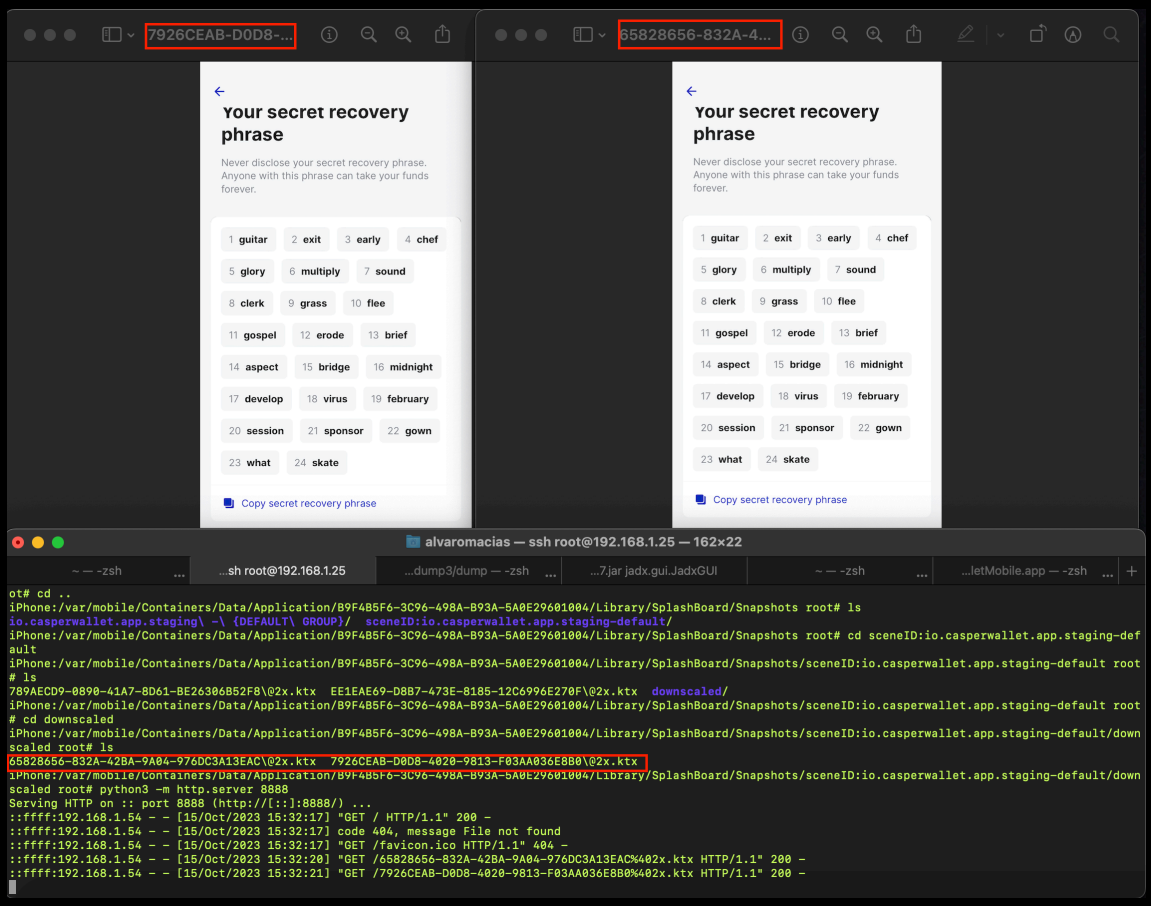


Figure 17: Snapshot in iOS contains sensitive details

Risk Level:

Likelihood - 2
Impact - 5

CVSS Vector:

- CVSS:3.1/AV:P/AC:L/PR:N/UI:N/S:U/C:H/I:N/A:N

Recommendation:

It is recommended to prevent sensitive data leaks in snapshots. One way to prevent this bad behavior is to put a blank screen or remove the sensitive data before taking the snapshot using the ApplicationDidEnterBackground()

function. The following is a sample remediation method that will set a default screenshot.

Swift:

Listing 20

```
private var backgroundImage: UIImageView?

func applicationDidEnterBackground(_ application: UIApplication) {
    let myBanner = UIImageView(image: #imageLiteral(resourceName:
"overlayImage"))
    myBanner.frame = UIScreen.main.bounds
    backgroundImage = myBanner
    window?.addSubview(myBanner)
}

func applicationWillEnterForeground(_ application: UIApplication)
{
    backgroundImage?.removeFromSuperview()
}
```

Objective-C:

Listing 21

```
@property (UIImageView *)backgroundImage;

- (void)applicationDidEnterBackground:(UIApplication *)application
 {
    UIImageView *myBanner = [[UIImageView alloc] initWithImage:@"
overlayImage.png"];
    self.backgroundImage = myBanner;
    self.backgroundImage.bounds = UIScreen.mainScreen.bounds;
    [self.window addSubview:myBanner];
}

- (void)applicationWillEnterForeground:(UIApplication *)
application {
    [self.backgroundImage removeFromSuperview];
}
```

This sets the background image to overlayImage.png whenever the application is back grounded. It prevents sensitive data leaks because overlayImage.png will always override the current view.

Remediation Plan:

SOLVED: The Make solved this finding. Added applicationWillResignActive and applicationDidBecomeActive to AppDelegate.mm with an animated filling of the screen with white color on the app in the background.

5.2 (HAL-15) iOS - SENSITIVE INFORMATION EXPOSURE VIA IOS CLIPBOARD - LOW

Description:

The iOS application allows users to copy the seed phrase to the iOS pasteboard. This can introduce potential security risks, as other malicious apps could potentially access and compromise sensitive information stored in the clipboard. Additionally, if the user has iCloud clipboard enabled, the seed phrase could be accessible on other devices connected to the same iCloud account.

If a malicious actor gains access to the user's seed phrase, they could potentially gain access to the user's cryptocurrency wallets or other accounts that rely on that seed phrase for authentication. This could result in the theft of funds or other sensitive data, as well as damage to the user's reputation and trust in your application.

Proof of concept:

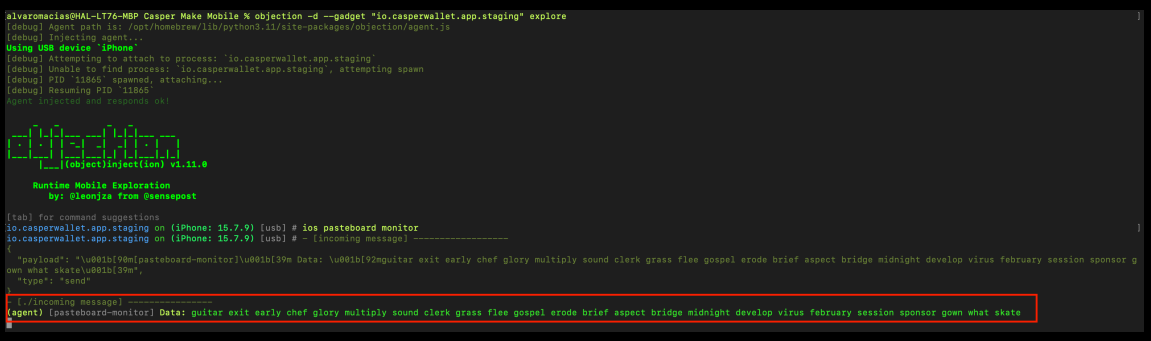


Figure 18: Mnemonic phrase captured from iOS Pasteboard

Risk Level:

Likelihood - 2
Impact - 3

CVSS Vector:

- CVSS:3.1/AV:N/AC:H/PR:H/UI:R/S:U/C:H/I:H/A:N

Recommendation:

To mitigate this vulnerability, it is recommended to consider disabling the ability to copy the seed phrase to the clipboard. Instead, alternative methods can be implemented for users to securely store their seed phrase, such as:

- Allowing users to export the seed phrase as an encrypted file, which can then be stored on an external storage device or a secure cloud storage service.
- Integrating your app with hardware wallets or other secure storage solutions to store the seed phrase.
- Encouraging users to write down the seed phrase on a piece of paper and store it in a secure location.

Reference:

- OWASP Sensitive Data Exposure

Remediation Plan:

PENDING: The Make will solve this finding in a future release.

5.3 (HAL-16) iOS - DUMP CLEAR TEXT MNEMONICS FROM MEMORY - LOW

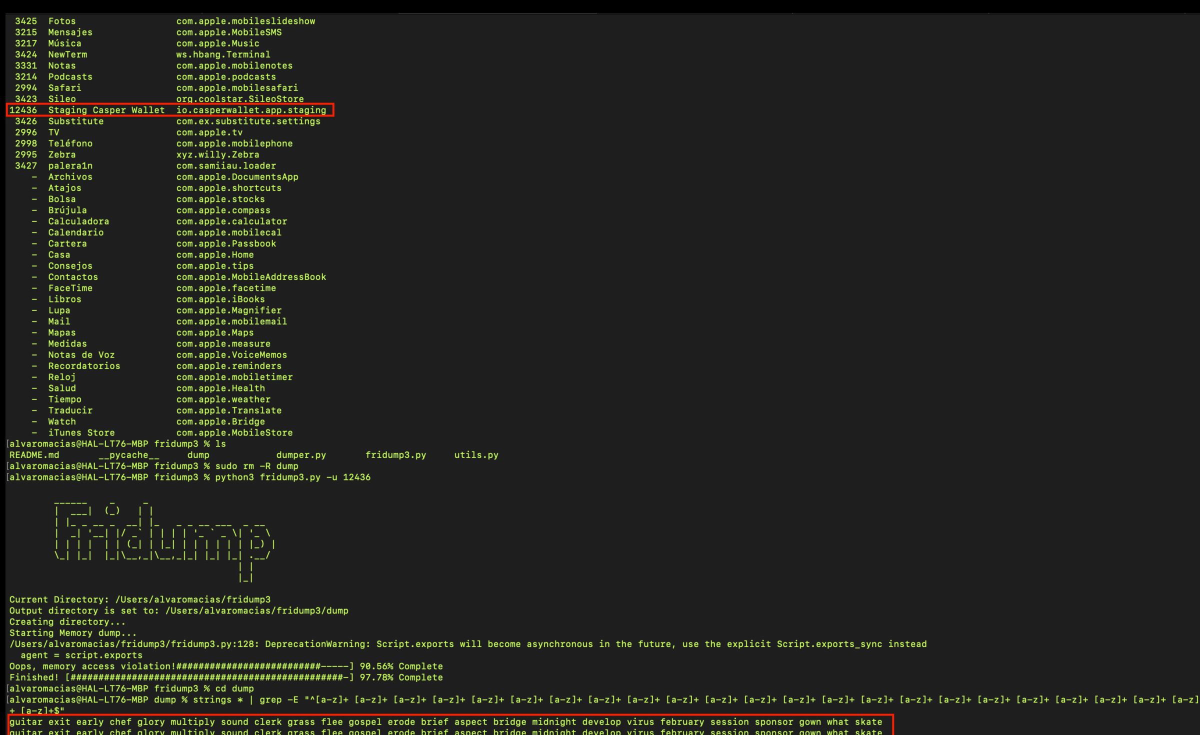
Description:

It was possible to dump the mnemonic phrase from the memory of the application and find the mnemonic pattern with regex. As there were no checks against the jailbroken devices, it made possible to dump the running app in memory and extract the mnemonics from it.

Note: In the application, Fridump was used to dump memory. Our goal was to dump the memory of the application and find the mnemonic pattern with regex.

Proof of concept:

Using fridump, dump the memory of the application running. Find the mnemonic pattern with regex over the dumped files.



```
 3425  Fotos                com.apple.mobileslideshow
 3215  Mensajes             com.apple.MobileSMS
 3217  Música               com.apple.Music
 3424  NewTerm              ws.hbang.Terminal
 3331  Notas                com.apple.mobilenotes
 3214  Podcasts             com.apple.podcasts
 2994  Safari               com.apple.mobilesafari
 3423  Sileo                org.coolstar.SileoStore
12436  Staging Casper Wallet  io.casperwallet.app.staging
 3426  Substitute           com.ex.substitute.settings
 2996  TV                   com.apple.tv
 2998  Teléfono             com.apple.mobilephone
 2995  Zebra                xyz.willy.Zebra
 3427  palera1n             com.samiiau.loader
    -  Archivos             com.apple.DocumentsApp
    -  Atajos               com.apple.shortcuts
    -  Bolsa                com.apple.stocks
    -  Brújula              com.apple.compass
    -  Calculadora          com.apple.calculator
    -  Calendario           com.apple.mobilecal
    -  Cartera              com.apple.Passbook
    -  Casa                 com.apple.Home
    -  Consejos             com.apple.tips
    -  Contactos            com.apple.MobileAddressBook
    -  FaceTime             com.apple.facetime
    -  Libros               com.apple.iBooks
    -  Lupa                 com.apple.Magnifier
    -  Mail                 com.apple.mobilemail
    -  Mapas                com.apple.Maps
    -  Medidas              com.apple.measure
    -  Notas de Voz         com.apple.VoiceMemos
    -  Recordatorios        com.apple.reminders
    -  Reloj                com.apple.mobiletimer
    -  Salud                com.apple.Health
    -  Tiempo               com.apple.weather
    -  Traducir             com.apple.Translate
    -  Watch                com.apple.Bridge
    -  iTunes Store         com.apple.MobileStore
alvaromacias@HAL-LT76-MBP fridump3 % ls
README.md       __pycache__     dump            dumper.py       fridump3.py     utils.py
alvaromacias@HAL-LT76-MBP fridump3 % sudo rm -R dump
alvaromacias@HAL-LT76-MBP fridump3 % python3 fridump3.py -u 12436

Current Directory: /Users/alvaromacias/fridump3
Output directory is set to: /Users/alvaromacias/fridump3/dump
Creating directory...
Starting Memory dump...
/Users/alvaromacias/fridump3/fridump3.py:128: DeprecationWarning: Script.exports will become asynchronous in the future, use the explicit Script.exports_sync instead
  agent = script.exports
Oops, memory access violation!##########################-----] 90.56% Complete
Finished! [################################################-] 97.78% Complete
alvaromacias@HAL-LT76-MBP fridump3 % cd dump
alvaromacias@HAL-LT76-MBP dump % strings * | grep -E "^[a-z]+ [a-z]+ [a-z]+ [a-z]+ [a-z]+ [a-z]+ [a-z]+ [a-z]+ [a-z]+ [a-z]+ [a-z]+ [a-z]+ [a-z]+ [a-z]+ [a-z]+ [a-z]+ [a-z]+ [a-z]+ [a-z]+ [a-z]+ [a-z]+ [a-z]+ [a-z]
+ [a-z]+$"
guitar exit early chef glory multiply sound clerk grass flee gospel erode brief aspect bridge midnight develop virus february session sponsor gown what skate
guitar exit early chef glory multiply sound clerk grass flee gospel erode brief aspect bridge midnight develop virus february session sponsor gown what skate
```

Figure 19: Memory dump of Make Casper application

Since the mnemonic was in cleartext when the user copied it, the severity has been downgraded.

Risk Level:

Likelihood - 1
Impact - 3

CVSS Vector:

- CVSS:3.1/AV:P/AC:H/PR:H/UI:R/S:U/C:H/I:N/A:N

Recommendation:

It is recommended to have protected measures against jailbreak detection and some open-source tools like Frida in the application to prevent loading/running the application if these tools are detected on the device. The crypto wallet application should erase the mnemonics from the memory after a wallet delete. This ensures that users can maintain the protection of the wallet during regular use.

Reference:

- iOS Tampering and Reverse Engineering
- OWASP Jailbreak Detection Methods
- IOSSecuritySuite

Remediation Plan:

PENDING: The Make will solve this finding in a future release.

5.4 (HAL-17) iOS - TRANSFER FUNCTIONALITY SHOULD REQUIRE PASSCODE - LOW

Description:

The crypto wallet application did not have the passcode protection implemented in the transfer wallet functionality.

The actions without rollback should be restricted with passcode/biometric auth to ensure that the user knows the risks and anyone other than the user could not use those functions.

Proof of concept:

- Go to Delete Account functionality located in Screens -> Send Tokens -> Confirming Sending.
- No protection is displayed in the transfer.

Risk Level:

Likelihood - 1
Impact - 3

CVSS Vector:

- CVSS:3.1/AV:P/AC:H/PR:L/UI:N/S:U/C:N/I:N/A:L

Recommendation:

Secure the functionality with the passcode to ensure that anyone other than the user could not use it.

Remediation Plan:

PENDING: The Make will solve this finding in a future release.

5.5 (HAL-18) iOS - DELETE FUNCTIONALITY SHOULD REQUIRE PASSCODE - LOW

Description:

The crypto wallet application did not have the passcode protection implemented in the delete wallet functionality.

The actions without rollback should be restricted with passcode/biometric auth to ensure that the user knows the risks and anyone other than the user could not use those functions.

Proof of concept:

- Go to Delete Account functionality located in Screens -> Accounts -> Delete Account.
- No protection is displayed in the delete account.

Risk Level:

Likelihood - 1
Impact - 3

CVSS Vector:

- CVSS:3.1/AV:P/AC:H/PR:L/UI:N/S:U/C:N/I:N/A:L

Recommendation:

Secure the functionality with the passcode to ensure that anyone other than the user could not use it.

Remediation Plan:

SOLVED: The Make solved this finding. Asked for pincode before opening delete wallet screen.

5.6 (HAL-19) iOS - LACK OF CERTIFICATE PINNING - LOW

Description:

Certificate pinning is the process of associating the backend server with a particular X.509 certificate or public key, instead of accepting any certificate signed by a trusted certificate authority (CA). After storing ("pinning") the server's certificate or public key, the mobile app will subsequently connect only to the known server. Withdrawing trust from external CAs reduces the attack surface (after all, there are many cases of CAs being compromised or tricked into issuing certificates to impostors).

The certificate can be pinned and hardcoded in the app or retrieved at the time the app first connects to the backend. In the latter case, the certificate is associated ("pinned" to) the host when the host is first seen. This alternative is less secure because attackers intercepting the initial connection can inject their certificates.

The target application has not implemented SSL pinning when establishing a trusted connection between the mobile applications and the back-end web services. Without enforcing SSL pinning, an attacker could man-in-the-middle the connection between mobile applications and back-end web services. This allows an attacker to sniff user credentials, session ID, etc. Certificate pinning is used in modern applications to prevent users from intercepting and analyzing HTTP traffic. Using this method, an application can verify the server's certificate and, in case there is a Man-in-The-Middle, not trust any other certificate than the one stored as default. There are many ways to perform this security countermeasure, and taking it in place does not ensure that a motivated attacker will be able to bypass it in time, but it does represent the first wall of defense against HTTP attacks.

Proof of concept:

1. The Portswigger certificate of Burp Suite was installed on the iOS device.
2. The App continued working as expected.
3. It was possible to intercept the traffic without any certificate pinning bypass:

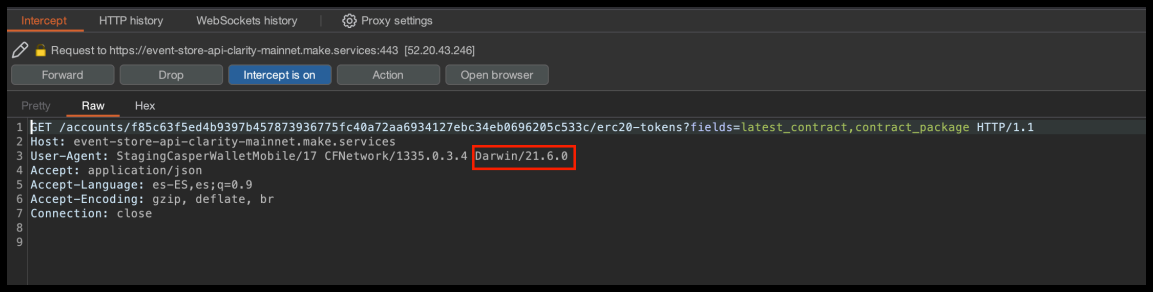


Figure 20: Traffic captured

Risk Level:

Likelihood - 2
Impact - 2

CVSS Vector:

- CVSS:3.1/AV:N/AC:H/PR:N/UI:N/S:U/C:L/I:N/A:N

Recommendation:

The application should implement a certificate pinning to only trust known certificates, to prevent as much as possible attackers with high privileges over the device to capture the traffic.

Remediation Plan:

PENDING: The Make will solve this finding in a future release.

5.7 (HAL-20) iOS - LACK OF JAILBREAK DETECTION MECHANISM ON THE iOS APPLICATION - LOW

Description:

Anti-jailbreak mechanisms are not used in the iOS application. These mechanisms can help mitigate reverse engineering, application modification, and unauthorized versions of mobile applications to some extent, but few if any will be completely successful against a determined adversary. However, they can be used as part of a defense-in-depth strategy that seeks to minimize the impact and likelihood of such an attack, along with binary patching, local resource modification, method hooking, method swizzling, and heap modification.

Screenshot:

Risk Level:

Likelihood - 2
Impact - 2

CVSS Vector:

- CVSS:3.1/AV:P/AC:H/PR:H/UI:R/S:C/C:L/I:L/A:N

Recommendation:

The application should not allow any modifications in its operation.

- Obfuscated code.
- Add Frida and other open-source jailbreak detection tools.
- Implement jailbreak detection.

Reference:

- iOS Tampering and Reverse Engineering
- objection - Runtime Mobile Exploration
- iOS Platform Security & Anti-tampering Swift Library

Remediation Plan:

SOLVED: The Make solved this finding. Added notification to the user of increased risks of using the app on a jailbroken device.

5.8 (HAL-21) iOS - LACK OF ANTI-HOOK ANTI-DEBUG MECHANISM ON THE APPLICATION - LOW

Description:

The tested application does not have any security features or mechanisms to prevent malicious actions, Anti Hook and Anti Debug.

Example Command:

- Install Frida on the jailbroken phone. Setup Jailbroken Device
- Use the Objection Tool to investigate the Anti-Hook mechanisms in the application. Objection
- Use the following command in the objection tool to investigate the Jailbroken device.

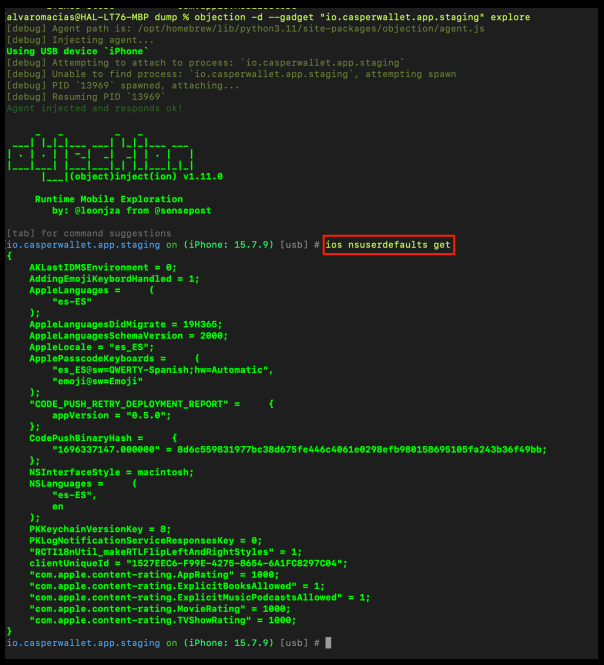
Listing 22

```
objection --gadget "<package name>" explore
```

- Run the following code in the objection.

Listing 23

```
# ios nsuserdefaults get
```



- You can see an application does not terminate; therefore, the application does not have anti-hook or anti-tampering mechanisms.

Risk Level:

Likelihood - 2
Impact - 2

CVSS Vector:

- CVSS:3.1/AV:P/AC:H/PR:H/UI:R/S:U/C:L/I:L/A:N

Recommendation:

Anti-Debug, Anti-Hook and Integrity Check mechanism (completed in the native code), which will protect against injection of various types of scripts into it, i.e., Frida Gadgets. The application should not allow modifications in its operation.

Reference:

- Owasp MSTG
- IOS Security Suite

Remediation Plan:

SOLVED: The Make solved this finding. Added check for debug mods. In case of detection blocking, a screenshot is shown.

5.9 (HAL-22) iOS - BACKGROUND SCREEN CATCHING - LOW

Description:

Manufacturers want to provide device users with an aesthetically pleasing experience at application startup and exit, so they introduced the screenshot-saving feature for use when the application has been executed in the background. This feature may pose a security risk since a user can deliberately screenshot the application while sensitive data is being displayed. A malicious application that is running on the device and able to continuously capture the screen may also expose data. Screenshots are written to local storage, from which they may be recovered by a rogue application (if the device is rooted) or someone who has stolen the device.

For example, capturing a screenshot of a banking application may reveal information about the user's account, credit, transactions, and so on.

In this specific, the mnemonic phrase can be captured while the application is not active.

Proof of concept:

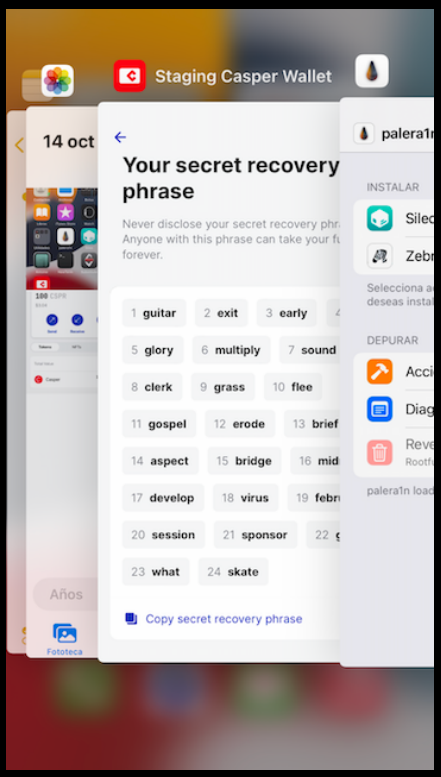


Figure 21: Background screen cache

Risk Level:

Likelihood - 1
Impact - 4

CVSS Vector:

- CVSS:3.1/AV:P/AC:L/PR:H/UI:N/S:U/C:L/I:N/A:N

Recommendation:

This vulnerability represents a minimal exposure to exploitation. Only the users of the mobile devices to which the attacker has access are affected by this vulnerability.

As a best practice, consider preventing run background screen caching if

the application displays sensitive data.

Remediation Plan:

SOLVED: The Make solved this finding. Added `applicationWillResignActive` and `applicationDidBecomeActive` to `AppDelegate.mm` with an animated filling of the screen with white color on the app in the background.

ANNEX I

6.1 Mobile App Security Testing Methodology

Local Authentication:

During local authentication, an app authenticates the user against credentials stored locally on the device. In other words, the user "unlocks" the app or some inner layer of functionality by providing a valid PIN, password or biometric characteristics such as face or fingerprint, which is verified by referencing local data. Generally, this is done so that users can more conveniently resume an existing session with a remote service or as a means of step-up authentication to protect some critical function.

On Android, there are two mechanisms supported by the Android Runtime for local authentication: the Confirm Credential flow and the Biometric Authentication flow.

Fingerprint authentication on iOS is known as Touch ID. The fingerprint ID sensor is operated by the SecureEnclave security coprocessor and does not expose fingerprint data to any other parts of the system. Next to Touch ID, Apple introduced Face ID: which allows authentication based on facial recognition. Both use similar APIs on an application level, the actual method of storing the data and retrieving the data (e.g. facial data or fingerprint related data is different).

The tests performed include enforced password/PIN strength requirements, 2FA, and re-authentication to ensure that the application correctly enforced password/PIN strength requirements, 2FA is functioning as expected and re-authentication is correctly being prompted during sensitive operations.

Data Storage:

Android Description

Protecting authentication tokens, private information, and other

sensitive data is key to mobile security. In this chapter, you will learn about the APIs Android offers for local data storage and best practices for using them.

The guidelines for saving data can be summarized easily: public data should be available to everyone, but sensitive and private data must be protected, or, better yet, kept out of device storage.

iOS Description

The protection of sensitive data, such as authentication tokens and private information, is key for mobile security. In this chapter, you'll learn about the iOS APIs for local data storage, and best practices for using them.

As little sensitive data as possible should be saved in permanent local storage. However, in most practical scenarios, at least some user data must be stored. Fortunately, iOS offers secure storage APIs, which allow developers to use the cryptographic hardware available on every iOS device. If these APIs are used correctly, sensitive data and files can be secured via hardware-backed 256-bit AES encryption.

The tests performed include storage of application data, data encryption, Keychain/Keystore access control, clipboard, sensitive data stored, snapshots, and many more to ensure that all stored passwords/PINs are properly encrypted, private keys are securely stored in an encrypted format, the proper access control is implemented to limit the accessibility of stored sensitive data, the clipboard is automatically cleared, etc.

Network Communication:

Almost every Android/iOS app acts as a client to one or more remote services. As this network communication usually takes place over untrusted networks such as public Wi-Fi, classical network based-attacks become a potential issue.

Most modern mobile apps use variants of HTTP-based web services, as these protocols are well-documented and supported.

The tests performed include traffic inspection, HTTPS communications, and cache to ensure that the application should use HTTPS or another secure protocols, the application does not store sensitive information in logs, cache, or analytics data that could potentially be accessed by other applications or attackers.

Cryptographic APIs:

iOS Description

Apple provides libraries that include implementations of the most common cryptographic algorithms. Apple's Cryptographic Services Guide is a great reference. It contains generalized documentation of how to use standard libraries to initialize and use cryptographic primitives, information that is useful for source code analysis.

Android Description:

Android cryptography APIs are based on the Java Cryptography Architecture (JCA). JCA separates the interfaces and implementation, making it possible to include several security providers that can implement sets of cryptographic algorithms. Most of the JCA interfaces and classes are defined in the java.security.* and javax.crypto.* packages. In addition, there are Android-specific packages android.security.* and android.security.keystore.*.

KeyStore and KeyChain provide APIs for storing and using keys (behind the scene, KeyChain API uses KeyStore system). The tests performed included

The tests performed include cryptographic methods' usage, and memory analysis to ensure the application uses a secure and up-to-date algorithm, and key material is not leaked in memory and is securely wiped from memory after use.

Anti-Reversing Defenses:

Description

The lack of these measures does not cause a vulnerability - instead, they

are meant to increase the app's resilience against reverse engineering and specific client-side attacks.

The tests performed include jailbroken/rooted devices' detection, reverse engineer tasks on the code, binary manipulation, and debug scenario to ensure that the application detects a jailbroken iOS device or a rooted Android device and responds accordingly, obfuscation techniques are employed to make reverse-engineering more difficult, mechanism to detect binary tampering or modification are implemented, etc.

Tampering and Reverse Engineering:

iOS Description

iOS reverse engineering is a mixed bag. On one hand, apps programmed in Objective-C and Swift can be disassembled nicely. In Objective-C, object methods are called via dynamic function pointers called "selectors", which are resolved by name during runtime. The advantage of runtime name resolution is that these names need to stay intact in the final binary, making the disassembly more readable. Unfortunately, this also means that no direct cross-references between methods are available in the disassembler and constructing a flow graph is challenging.

Android Description

Android's openness makes it a favorable environment for reverse engineers. In the following chapter, we'll look at some peculiarities of Android reversing and OS-specific tools as processes.

Android offers reverse engineers big advantages that are not available with iOS. Because Android is open-source, you can study its source code at the Android Open-Source Project (AOSP) and modify the OS and its standard tools any way you want. Even on standard retail devices, it is possible to do things like activating developer mode and sideloading apps without jumping through many hoops. From the powerful tools shipping with the SDK to the wide range of available reverse engineering tools, there are a lot of niceties to make your life easier.

The tests performed include injecting snippets with Frida to test the application's defense, assessment errors to examine the application's

response to ensure that the application detects these tools and responded as expected, and error messages should not disclose sensitive information or information that could aid an attacker.

Input Validation:

For any publicly accessible data storage, any process can override the data. This means that input validation needs to be applied the moment the data is read back again.

The tests performed include different input injections and how the application handles the malicious inputs to ensure that the application sanitize and validate all user inputs before processing to prevent potential attacks, mitigating the risk of injection attacks.

Server-Side APIs:

Attacks targeting APIs are one of the most serious security threats facing businesses, as they provide direct access to sensitive data and functionalities. And attackers have become aware of the popularity of APIs and the existence of critical vulnerabilities in these interfaces. The problem is that web applications remain the primary target of attacks and APIs now represent 90% of the attack surface of web applications. Thus, APIs have become one of the main attack vectors, with devastating financial consequences for the companies that bear the costs.

The tests performed include server-side API testing, and high-load tests in the backend to ensure that the API is not vulnerable to vulnerabilities such as business logic vulnerabilities, access control, authentication, etc. and the application should handle unexpected volumes of data or requests gracefully, without crashing or becoming unresponsive and not observing any application crashes or slowdowns.

THANK YOU FOR CHOOSING

// HALBORN