// HALBORN

# SteakWallet

## Mobile App Pentest

Prepared by: **Halborn**

Date of Engagement: **March 4th, 2022 - April 7th, 2022**

Visit: **Halborn.com**

# DOCUMENT REVISION HISTORY

| VERSION | MODIFICATION | DATE | AUTHOR |
|---|---|---|---|
| 0.1 | Document Creation | 03/17/2022 | Pablo Gómez |
| 0.2 | Draft Review | 04/07/2022 | Gabi Urrutia |
| 1.0 | Remediation Plan | 07/26/2022 | Pablo Gómez |
| 1.1 | Remediation Plan Review | 07/29/2022 | Gabi Urrutia |
| 1.2 | Minor changes | 07/29/2022 | Pablo Gómez |

# 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 |
| Pablo Gómez | Halborn | Pablo.Gomez@halborn.com |

# EXECUTIVE OVERVIEW

## 1.1 INTRODUCTION

The SteakWallet app allows users to keep staking their crypto assets over a wide range of blockchain networks. In addition, they can operate with them as a regular Wallet and deposit or withdraw most of the top known cryptocurrencies and tokens.

SteakWallet engaged Halborn to conduct a security assessment on their mobile applications, both Android and iOS on March 4th, 2022 and ending on April 7th, 2022. The security assessment was scoped to Android and iOS SteakWallet Applications. The client team provided the application source code for Halborn to conduct security testing using tools to scan, detect, validate possible vulnerabilities and report findings at end of engagement.

Since attackers can find new attack vectors and ways to exploit information security, and penetration tests are based on manual human testing, it is worth noting that this assessment does not represent any guarantee that applications are completely secure. Nevertheless, the tools and methods used by attackers have been used to ensure the highest level of security.

## 1.2 AUDIT SUMMARY

The team at **Halborn** was provided a month for the engagement and assigned a full-time security engineer to audit the security of the assets in scope. The engineer is a blockchain and smart contract security expert with advanced mobile penetration testing, smart-contract hacking, and deep knowledge of multiple.

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

In summary, Halborn identified some security risks that were mostly addressed by the SteakWallet Team.

# 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 audit:

- Storing private keys and assets securely.
- Send/Receive tokens and assets securely to another wallet.
- Exposure of any critical information during user interactions with the blockchain and external libraries.
- 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.
- Fuzzing of all input parameters.
- Technology stack-specific vulnerabilities and Code Audit.
- Known vulnerabilities in 3rd party / OSS dependencies.

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.

| CRITICAL | HIGH | MEDIUM | LOW | INFORMATIONAL |
|---|---|---|---|---|

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

## 1.4 SCOPE

IN-SCOPE:

The **security assessment** was scoped to:

1. SteakWallet Mobile Applications
    (a) Applications in the scope:
        i. SteakWallet Github
            Monorepo URL: https://github.com/steakwallet/monorepo

OUT-OF-SCOPE:

External libraries.

# 2. ASSESSMENT SUMMARY & FINDINGS OVERVIEW

| CRITICAL | HIGH | MEDIUM | LOW | INFORMATIONAL |
|---|---|---|---|---|
| 0 | 0 | 2 | 3 | 2 |

LIKELIHOOD

IMPACT

| | | | | |
|---|---|---|---|---|
| | | | | |
| (HAL-03) | (HAL-02) | | | |
| | | | (HAL-01) | |
| | (HAL-04) | | | |
| (HAL-06)<br>(HAL-07) | | (HAL-05) | | |

# EXECUTIVE OVERVIEW

| SECURITY ANALYSIS | RISK LEVEL | REMEDIATION DATE |
|---|---|---|
| HAL01 - CERTIFICATE PINNING BYPASS FOR MULTIPLE DOMAINS | Medium | SOLVED - 07/26/2022 |
| HAL02 - LACK OF ANTI-TAMPERING MECHANISM | Medium | RISK ACCEPTED |
| HAL03 - MISCONFIGURED ATS (APP TRANSPORT SECURITY) | Low | SOLVED - 07/26/2022 |
| HAL04 - LACK OF ANTI-HOOK ANTI-DEBUG MECHANISMS | Low | SOLVED - 07/26/2022 |
| HAL05 - MMKV LOGS LEAKED | Low | SOLVED - 07/26/2022 |
| HAL06 - HARDCODED API KEYS | Informational | FUTURE RELEASE |
| HAL07 - DEFAULT SEED KEY ON RANDOMBYTESMODULE | Informational | FUTURE RELEASE |

# FINDINGS & TECH DETAILS

# 3.1 (HAL-01) CERTIFICATE PINNING BYPASS FOR MULTIPLE DOMAINS - MEDIUM

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 on iOS 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 counter measure, 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.

**SteakWallet** implements SSL Pinning, but it uses some methods with common

names on iOS, which can be easily bypassed when performing application hooking and method tracing. Nevertheless, on Android it has not been possible to intercept traffic.

Proof of concept:

1. Connect to the application using Frida and Objection

**Listing 1**

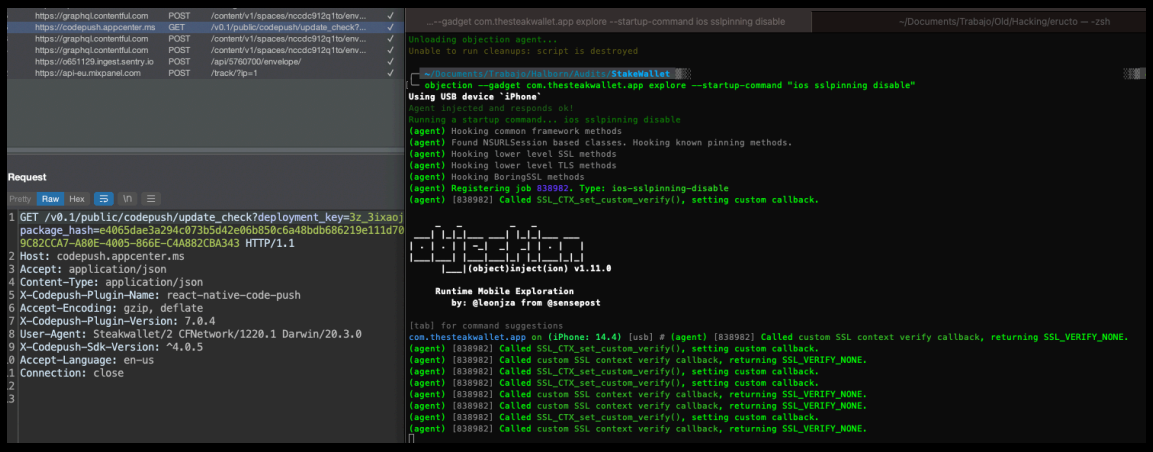
```
objection --gadget fi.thesteakwallet.app explore
```

2. Set the automatic certificate pinning bypass implemented by objection

**Listing 2**

```
ios sslpinning disable
```

As it can be seen below, the SSL_CTX_SetCustomVerify method is triggered and modified at runtime. In addition, plain traffic capture evidence is shown on left side:




Risk Level:

**Likelihood - 4**
**Impact - 3**

Recommendation:

It is recommended to prevent these actions by enforcing anti-tampering and anti-debugging mechanisms. This vulnerability is related to jailbreak and 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.
- Apple suggests pinning a CA public key by specifying it in the Info.plist file in App Transport Security Settings.
- TrustKit, an open-source SSL pinning library for iOS and macOS is available. It provides an easy-to-use API to implement pinning and has been deployed in many apps.

References:

- iOS Security Suite
- Configure Server Certificates - iOS
- OWASP Pinning Cheat Sheet
- Guidelines Towards Secure SSL Pinning in Mobile Applications

Remediation Plan:

**SOLVED:** Since the application keeps crashing when executing on rooted/-jalbroken devices or when using a patched app, it is a direct consequence of certificate pinning not being bypassable as before.

## 3.2 (HAL-02) LACK OF ANTI TAMPERING MECHANISMS - MEDIUM

Description:

With the difficulty of jailbreaking iOS and rooting Android devices increasing with each new version released, repacking and resigning applications for sideload on non-rooted devices has been a topic of considerable interest among security researchers recently.

For iOS devices, due to several code signing applications implemented in the kernel, sideloading applications is restricted on non-jailbroken devices. This is done to prevent malicious actors from distributing and running untrusted code on the devices of unsuspecting users. This code signing enforcement, with Apple's AppStore application review process, has significantly reduced the distribution of malicious applications to iOS users.

On Android, there are fewer protection measures, since the full APK can be downloaded directly from memory or from many web services that allow any user to download these files.

All that being said, Halborn Team has been able to successfully make modifications and execute the Android APK and iOS IPA files.

Proof Of Concept (iOS):

1. 'Run the following commands:

**Listing 3**

```
./Clutch --dump com.thesteakwallet.app
unzip SteawWallet.ipa
```

2. Change some source file in Payload folder

3. Create an Empty Project on XCode and install it on the device to generate the provisioning files (embedded.mobileprovision)
4. Run the following commands

**Listing 4**

```
fastlane sigh resign SteakWallet.ipa --signing_identity "Apple Development: pablo.gomez@halborn.com" -p embedded.mobileprovision
```

5. Use **ideviceinstaller** or **ios-deploy** to install the application on the device.

Proof Of Concept (Android):

1. Download the app from the memory or from the APK repository
2. Modify files within the application
3. Run the following commands

**Listing 5**

```
keytool -genkey -v -keystore my-release-key.keystore -alias halborn_signed -keyalg RSA -keysize 2048 -validity 10000
apktool d ${app_name}.apk
apktool b -f -d ${app_name}
jarsigner -verbose -sigalg SHA1withRSA -digestalg SHA1 -keystore my-release-key.keystore ${app_name}/dist/${app_name}.apk halborn_signed
jarsigner -verify -verbose -certs ${app_name}/dist/${app_name}.apk
~/Library/Android/sdk/build-tools/32.1.0-rc1/zipalign -p -v 4 ${app_name}/dist/${app_name}.apk signed-${app_name}.apk
```

Risk Level:

**Likelihood - 2**
**Impact - 4**

Recommendation:

Use the checksums, digital signatures, and other validation mechanisms to help detect file tampering. When an attacker attempts to manipulate the application, the correct checksum is not preserved and this could detect and prevent illegitimate execution. Note that such techniques are not foolproof and can be bypassed by a sufficiently motivated attacker. Checksum, digital signature and other validation techniques increase the amount of time and effort an attacker must successfully spend to breach the application. An application can silently wipe its user data, keys, or other important data wherever tampering is detected to further challenge an attacker.

References:

- iOS Tampering and Reverse Engineering
- iOS Platform Security & Anti-tampering Swift Library
- Cydia Impactor
- Repacking iOS Applications

Remediation Plan:

**RISK ACCEPTED:** The SteakWallet team accepted the risk of this finding.

## 3.3 (HAL-03) MISCONFIGURED ATS (APP TRANSPORT SECURITY) - LOW

Description:

Application Transport Security (ATS) was introduced by Apple in iOS 9 and adds and controls another security layer regarding secure connections. ATS is used by iOS to protect connections and prevent devices and applications to connect using insecure protocols, certificates, and cipher suites. Within this additional protection layer, there is a configuration field named NSAllowsArbitraryLoads that can be used to exempt or permit the use of HTTP (non ciphered connections) for some specific domains.

Setting this flag to **True** or **YES** allows an application to send all the information in plain text and eases attackers to perform Man-in-The-Middle attacks. Furthermore, Apple forces developers to clearly justify the use of this flag to accept the app publishing on the AppStore. If it is not clearly justified, it can be rejected.

Halborn Team has detected that the iOS application has the flag NSAllowsArbitraryLoads set to **YES** and has the localhost domain as an exception as follows:

```
<string>v2</string>
<key>LSRequiresIPhoneOS</key>
<true/>
<key>MinimumOSVersion</key>
<string>13.0</string>
<key>NSAppTransportSecurity</key>
<dict>
        <key>NSAllowsArbitraryLoads</key>
        <true/>
        <key>NSExceptionDomains</key>
        <dict>
                <key>localhost</key>
                <dict>
                        <key>NSExceptionAllowsInsecureHTTPLoads</key>
                        <true/>
                </dict>
        </dict>
</dict>
<key>NSCameraUsageDescription</key>
<string>Steakwallet is requesting access to your camera to scan address or WalletC
```

Since NSExceptionAllowsInsecureHTTPLoads is set for localhost domain, it seems to be a misconfiguration on the parent NSAllowsArbitraryLoads, which should be set as **False** or **NO**

Risk Level:

**Likelihood - 1**
**Impact - 4**

Recommendation:

It is recommended to use the NSAllowsArbitraryLoads flag always set to **False** or **NO** and set exceptions using the fields NSExceptionDomains in order to **except the using of HTTPS**.

References:

- NSAllowsArbitraryLoads Docs
- OWASP Testing Network Communication

Remediation Plan:

**SOLVED**: Verified that misconfigured parameters are now correctly set as follows:

## 3.4 (HAL-04) LACK OF ANTI-HOOK ANTI-DEBUG MECHANISMS - LOW

Description:

The tested application does not have properly set security features or mechanisms to prevent malicious actions, Anti Hook and Anti Debug. It should be noted that the Debug.isDebuggerConnected() method has been detected in the Android app and there are some virtualization tests, but the test team has been able to successfully execute the objection and Frida, indicating that these anti-debugging are not being properly managed.

Example Command:

- Install Frida on the Android or iOS 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 6**

```
objection --gadget "com.thesteakwallet.app" explore
```

- Run the following code on the objection.

**Listing 7**

```
com.thesteakwallet.app on (iPhone: 14.4) [usb] # ios
↳ nsuserdefaults get
```

- You can see that an application does not terminate; therefore the application has no anti-hook or anti-tamper mechanisms.

Risk Level:

**Likelihood - 2**
**Impact - 2**

Recommendation:

Anti-Debug, Anti-Hook and Integrity Check mechanism (completed in 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.

References:

- iOS Platform Security & Anti-tampering Swift Library
- Owasp MSTG
- Android Google SafetyNet

Remediation Plan:

**SOLVED:** Due to the fact that the application keeps crashing when executing it on rooted/jalbroken devices or using a patched app, it is a direct consequence that the app implements anti-debug/anti-hook mechanisms.

## 3.5 (HAL-05) MMKV LOGS LEAKED - LOW

Description:

Android logs can be used to debug applications and represent a useful tool to analyze an application behavior. In some cases, developers and third-party software leaks information that could be used by attackers to deeply understand and exploit the solutions developed.

In this specific case, Halborn has detected in Android logs the use of MMVK functionality. This piece of software is used to encrypt and decrypt the files stored in:
- Android: **/data/user/0/fi.steakwallet.app/files/mmkv**
- iOS: **/var/mobile/Containers/Data/Application//Documents/mmkv**

Since the names of these files are secure.wallets.zustand or zustand.default, it is clear that these files are managed by Zustand.

Halborn has not been able to decrypt these files, but they seem to be encrypted using the keys stored on the keychain/keystore. Since that keys are stored using good practices, this vulnerability has been marked as Low.

Proof Of Concept:

1. Set the Debugging Options (USB Debugging) on Android device
2. Use LogCat to check logs from the device

**Listing 8**

```
1 adb logcat | grep -i steakwallet
```

```
09 D RecentTasksList: dumpTasks# id=108, cmp=ComponentInfo{fi.steakwallet.app/fi.steakwallet.app.MainActivity}, uid=0, locked=false, isAppLocked=false;
09 D BiometricService: Package: fi.steakwallet.app Authenticator ID: 0 Modality: 2 Reported Modality: 2 Status: 1
13 D SoLoader: libturbomodulejsijni.so not found on /data/data/fi.steakwallet.app/lib-main
13 D SoLoader: libturbomodulejsijni.so found on /data/data/fi.steakwallet.app/lib-0
13 D SoLoader: libreact_nativemodule_core.so not found on /data/data/fi.steakwallet.app/lib-main
13 D SoLoader: libreact_nativemodule_core.so found on /data/data/fi.steakwallet.app/lib-0
14 D RecentTasksList: dumpTasks# id=108, cmp=ComponentInfo{fi.steakwallet.app/fi.steakwallet.app.MainActivity}, uid=0, locked=false, isAppLocked=false;
16 D SoLoader: libreactnativeblob.so not found on /data/data/fi.steakwallet.app/lib-main
16 D SoLoader: libreactnativeblob.so found on /data/data/fi.steakwallet.app/lib-0
15 I MMKV    : Installing MMKV JSI Bindings for MMKV root directory: /data/user/0/fi.steakwallet.app/files/mmkv
15 I MMKV    : <MMKV.cpp:207::initializeMMKV> root dir: /data/user/0/fi.steakwallet.app/files/mmkv
15 I MMKV    : <MemoryFile.cpp:97::open> open fd[0x8c], /data/user/0/fi.steakwallet.app/files/mmkv/zustand.default
15 I MMKV    : <MemoryFile.cpp:97::open> open fd[0x8e], /data/user/0/fi.steakwallet.app/files/mmkv/zustand.default.crc
15 I steakwallet.ap: Background young concurrent copying GC freed 21265(846KB) AllocSpace objects, 7(1000KB) LOS objects, 0% free, 11MB/11MB, paused 308us total 141.603ms
16 D SoLoader: libimagepipeline.so not found on /data/data/fi.steakwallet.app/lib-main
16 D SoLoader: libimagepipeline.so found on /data/data/fi.steakwallet.app/lib-0
16 D SoLoader: libjnigraphics.so not found on /data/data/fi.steakwallet.app/lib-main
16 D SoLoader: libjnigraphics.so not found on /data/data/fi.steakwallet.app/lib-0
16 D SoLoader: libjnigraphics.so not found on /data/data/fi.steakwallet.app/lib-1
16 D SoLoader: libjnigraphics.so not found on /data/data/fi.steakwallet.app/lib-2
16 D SoLoader: libjnigraphics.so not found on /data/app/~~jtMZIkQJH8dG33Gg9MPUTQ==/fi.steakwallet.app-b1fpR_fWX677fyINBvrf5w==/lib/arm64
15 I steakwallet.ap: Background concurrent copying GC freed 138663(3887KB) AllocSpace objects, 8(1024KB) LOS objects, 49% free, 13MB/26MB, paused 244us total 197.079ms
11 D BiometricService: Package: fi.steakwallet.app Authenticator ID: 0 Modality: 2 Reported Modality: 2 Status: 1
14 D BiometricService: Package: fi.steakwallet.app Authenticator ID: 0 Modality: 2 Reported Modality: 2 Status: 1
```

Risk Level:

**Likelihood - 3**
**Impact - 1**

Recommendation:

It is recommended to not prompt any kind of logging information in production releases. It can be used in the ‘ProGuard framework or modify the source code to achieve this.

References:

- Android Shrink Code
- Android ProGuard
- Android LogCat

Remediation Plan:

**SOLVED**: No log found regarding sensitive MMKV information.

## 3.6 (HAL-06) HARDCODED API KEYS - INFORMATIONAL

Description:

API keys are used in mobile applications and other services to connect to third-party solutions to share or gather information, interact with user data and send statistics to aggregators and analysis tools. These API keys can be considered as sensitive information since they allow impersonating developers and products sending information or mixing fake data within these statistics.

Halborn has been able to decompile Android source code and found an API key and MIXPANEL tokens:
- COVALENT_API _KEY
- DEV_MIXPANEL_TOKEN
- MIXPANEL_TOKEN (prod)

Since API key found is used for querying crypto prices, this vulnerability has been set as Low. Nevertheless, an attacker could use the mix panel tokens to generate fake data and send it to SteakWallet project as is discussed on this link.

Proof Of Concept:

1. Use a decompiling method such as MobSF framework or dex2jar tool
2. Go to **fi/steakwallet/app/BuildConfig.java**

BuildConfig.java

```java
package fi.steakwallet.app;
/* loaded from: classes.dex */
public final class BuildConfig {
    public static final String API_URL = "https://api-beta.steakwallet.fi";
    public static final String APPLICATION_ID = "fi.steakwallet.app";
    public static final String BUILD_TYPE = "release";
    public static final String COVALENT_API_KEY = "ckey_[redacted]f";
    public static final boolean DEBUG = false;
    public static final String DEV_MIXPANEL_TOKEN = "1[redacted]";
    public static final String ENV = "prod";
    public static final String FLAVOR = "prod";
    public static final String MIXPANEL_TOKEN = "f[redacted]";
    public static final String TITLE = "Steakwallet";
    public static final int VERSION_CODE = 61;
    public static final String VERSION_NAME = "2.0.8";
}
```

Risk Level:

**Likelihood - 1**
**Impact - 1**

Recommendation:

It is recommended to not store sensitive information in local storagenor hardcoded in source code as it can be read by anyone who has access to the device or any application or malware installed on a rooted/jailbroken device. Sensitive data hardcoded locally on the device should always be encrypted and stored within the application sandbox. It is recommended to encrypt sensitive data in the iOS Keychain and Android Keystore (similar minimum version requirements).

References:

- Mixpanel Security Discussion
- OWASP Testing Data Storage
- OWASP Testing Data Storage - 2
- Keychain Item Access
- Keychain Services
- Keystore System

Remediation Plan:

**PENDING**: Postponed due to informational status and not related with sensitive information. Going forward, the SteakWallet team will keep them on its backend and retrieve them using a verified signature.

## 3.7 (HAL-07) DEFAULT SEED KEY ON RANDOMBYTESMODULE - INFORMATIONAL

Description:

Random modules are used for a wide range of functionalities and applications. They allow developers to generate random numbers or words, in most of the cases, from a seed from which comes all the derivated values. Consecuently, having the same seed, may generate the same list of pseudo-random values.

In this case, the Halborn Team has detected that java's RandomBytesModule is used. This module has its seed hardcoded by default, and it has not being changed, nor derived from time or other secure generator.

Proof Of Concept:

1. Use a decompiling method such as MobSF framework or dex2jar tool
2. Go to **bitgo/randombytes/RandomBytesModule.java**

RandomBytesModule.java

```java
package com.bitgo.randombytes;

import android.util.Base64;
import com.facebook.react.bridge.Callback;
import com.facebook.react.bridge.ReactApplicationContext;
import com.facebook.react.bridge.ReactContextBaseJavaModule;
import com.facebook.react.bridge.ReactMethod;
import java.security.SecureRandom;
import java.util.HashMap;
import java.util.Map;
/* loaded from: classes.dex */
public class RandomBytesModule extends ReactContextBaseJavaModule {
    private static final String SEED_KEY = "seed";

    public RandomBytesModule(ReactApplicationContext reactApplicationContext) {
        super(reactApplicationContext);
    }

    private String getRandomBytes(int i) {
        byte[] bArr = new byte[i];
        new SecureRandom().nextBytes(bArr);
        return Base64.encodeToString(bArr, 2);
    }

    @Override // com.facebook.react.bridge.BaseJavaModule
    public Map<String, Object> getConstants() {
        HashMap hashMap = new HashMap();
        hashMap.put(SEED_KEY, getRandomBytes(4096));
        return hashMap;
    }

    @Override // com.facebook.react.bridge.NativeModule
    public String getName() {
        return "RNRandomBytes";
    }

    @ReactMethod
```

Risk Level:

**Likelihood - 1**
**Impact - 1**

Recommendation:

It is recommended to use a secure random generator based on time, user behavior or, at least, modify the random seed to not generate the same random sequence in most of the cases.

References:

- Java Secure Random Generator
- React RandomBytesModule Used
- SecureRandom Properly Seeded

Remediation Plan:

**PENDING**: The Steakwallet team will change the affected library in a future release.

# PERFORMED TESTS

# 4.1 Testing Application Binary Protection

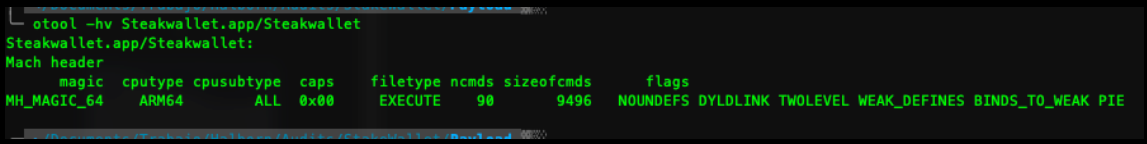
Unlike an Android application, an iOS binary can only be disassembled, not decompiled. A Mach-O binary file is the app binary of an iOS application. It is the machine code or executable file that runs on an iPhone. Complete reverse engineering of an iOS application to produce the source code is not possible. However, specified parts of library or object files can be dumped using existing tools. This is where Otool comes in.

Proof Of Concept - iOS:

1. Run the following command on the jailbroken device.

**Listing 9**

```
otool -hv SteakWallet
```




2. In the image above, we can clearly see that ASLR is enabled. When ASLR is disabled in an iOS application, certain memory structures and modules will not be randomly placed, creating the potential of a Buffer Overflow.

3. Next, the following command will examine Stack Smashing Protection.

**Listing 10**

```
otool -I -v SteakWallet | grep stack
```

```
└─ otool -I -v Steakwallet| grep stack
0x0000001004d8718  1948 ___stack_chk_fail
0x0000001004d9444  2463 _sigaltstack
0x0000010100619250  1949 ___stack_chk_guard
0x000000010061c478  1948 ___stack_chk_fail
0x000000010061cd40  2463 _sigaltstack
```

## 4.2 Result

An application’s binary protection mechanisms are configured correctly.

# 4.3 Testing Keychain Secrets

Description:

Encryption keys are used to protect sensitive information in applications. In blockchain-related mobile apps, these keys can be used to encrypt and protect one of the most important assets over this kind of applications: The mnemonic phrases. An attacker that can access this information could manipulate and take control over every wallet action. It would be possible to access to the keychain and use encryption keys to decrypt the secret files, likely stored on /var/mobile/Containers/Data/Application/<BUNDLE_ID>/Documents/mmkv.

Halborn's Team has not been able to decrypt files, but encryption keys exposure may lead to a loss of integrity and confidentiality. In addition, these keys are protected via biometric authentication.

Proof of Concept (iOS):

- 1. Connect to the app via objection

**Listing 11**

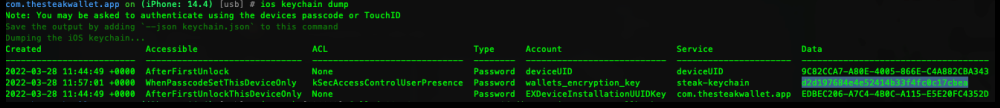
```
objection --gadget com.thesteakwallet.app explore
```

- 2. Access to the keychain stored values

**Listing 12**

```
ios keychain dump
```

- 3. Get the wallets encryption key secret



Proof of Concept (Android):

- 1. Connect to the app via objection

**Listing 13**

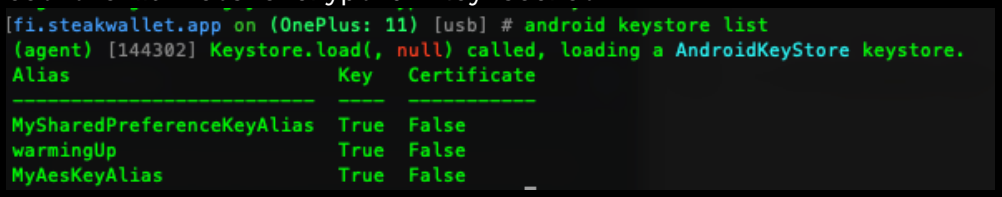
```
objection --gadget fi.steakwallet.app explore
```

- 2. Access to the keychain stored values

**Listing 14**

```
android keystore list
```

- 3. Get the wallets encryption key secret




Result:

The application's keystore and keychain mechanisms are correctly configured.

THANK YOU FOR CHOOSING

// HALBORN