Least Authority
PRIVACY MATTERS

TezosKit Client

Security Audit Report

Tezos Foundation

Final Audit Report Version: 8 May 2020

Table of Contents

Overview

Background

Tezos Foundation has requested that Least Authority perform a security audit of TezosKit, a Swift based toolbox for interacting with the Tezos blockchain.

Project Dates

- **March 24 - April 12:** Initial Review *(Completed)*
- **April 13:** Delivery of the Initial Audit Report *(Completed)*
- **May 6 - 7:** Verification Review *(Completed)*
- **May 8:** Delivery of the Final Audit Report *(Completed)*

Review Team

- Mirco Richter, Cryptography Researcher and Engineer
- Nathan Ginnever, Security Researcher and Engineer
- Jehad Baeth, Security Researcher and Engineer

Coverage

Target Code and Revision

For this audit, we performed research, investigation, and review of the TezosKit followed by issue reporting, along with mitigation and remediation instructions outlined in this report.

The following code repositories are considered in-scope for the review:
- TezosKit: https://github.com/keefertaylor/tezoskit
  - ~ 5,000 of Swift code

Changes made to the Elliptic Curve Key Pair are considered in-scope, however, the Elliptic Curve Key Pair library is considered out of scope.

Specifically, we examined the Git revisions for our initial review:

```
5b24c098a15121aaa5cc59812faec8a679f9290b
```

For the verification, we examined the Git revision:

```
20d3ebde3eb91fca7c6e816f0c348be7d023984f
```

All file references in this document use Unix-style paths relative to the project's root directory.

Areas of Concern

Our investigation focused on the following areas:

- Attacks that impacts funds, such as draining or manipulating of funds;
- Mismanagement of funds via transactions;
- Secure communication between the nodes;
- Proper management of encryption and signing keys;
- Vulnerabilities within each component as well as secure interaction between the contracts and network components;

- Correctness of the implementation;
- Adversarial actions and protection against malicious attacks on the network;
- Economic incentives: ensure token economics (monetary incentives to punish bad behavior and reward good behavior) are included and functional;
- DoS/security exploits;
- Inappropriate permissions and excess authority;
- Data privacy, data leaking, and information integrity; and
- Anything else as identified during the initial analysis phase.

Findings

General Comments

Our team found the TezosKit code to be well organized and easy to read. It was sufficiently commented with an adequate amount of tests covering a substantial amount of the codebase. The documentation was comprehensive, accurate, and very easy to follow, which facilitated our team's ability to effectively and comprehensively review the code.

We also found that the project adhered to security best practices and standards, making it clear that security was strongly considered throughout the design and implementation. For example, there were no linting errors present and the codebase makes good use of SwiftLint to enforce Swift styles and conventions. This helps keep the codebase standardized so as to not introduce possible unique variations of style that could be potentially vulnerable. In addition, the cryptographic utilities such as encoding, decoding and compression follow all known standards.

Our team found it notable that the secure enclave generates Tz3 accounts that provide a unique extra layer of security to devices that contain an Apple T2 chip. The enclave separates the P256 private key from the device processor which makes its extraction much harder, as the private information never leaves the chip hardware and only a pointer is passed to the wallet software. This is an improvement over previous mobile wallets that support cryptocurrency that do not have a secure enclave.

Although the inability to extract the key provides extra security, it presents a challenge to backing up the Tezos account (Suggestion 6). Without the ability to back up a Tezos account, it is possible to lose the entire contents of the account in the case of system failures. Given that the origin of the constant parameters for the Secp256r1 or NIST-P256 curve are unknown, some cryptography experts express caution in using the P256 curve as they may have been selected as a back door. However, there is no known evidence that the parameters are compromised beyond attempts. Additionally, we found that Tezos primitive types are not represented in all detail in their Swift counterparts (Issue C and Suggestion 1).

Specific Issues

We list the issues we found in the code in the order we reported them. In most cases, remediation of an issue is preferable, but mitigation is suggested as another option for cases where a trade-off could be required.

ISSUE / SUGGESTION	STATUS
Issue A: Swift Memory Security Might Compromise Private Key Deletion	Resolved
Issue B: Undefined Behavior in the Tez Class	Resolved

Issue C: TezosKit Does Not Support Unbounded `nat` and `int` Types	Resolved
Issue D: Hex Seeded Key Initializer is Hard Coded to Ed25519 Curve	Resolved
Suggestion 1: Implement All Primitive Michelson Types in Separate Classes	Resolved
Suggestion 2: Improve Precision in Cryptography Documentation	Resolved
Suggestion 3: Review Dependencies Used in TezosKit	Resolved
Suggestion 4: Ensure Secure Environment for Running TezosKit Applications	Resolved
Suggestion 5: Minimal Amount of Overly Complex Code	Invalid
Suggestion 6: Warn Users of Incapability of Backing Up Enclave Key	Resolved

Issue A: Swift Memory Security Might Compromise Private Key Deletion

Location

Function `deleteKeyPair()` in /TezosKit/Crypto/EllipticCurveKeyPair/EllipticCurveKeyPair.swift

Synopsis

According to the expected behavior of the Swift programming language, function `deleteKeyPair()` might not fully erase the footprint of the private key in storage. This behavior in Swift is designed so that some parts of the memory are not controlled by the developers. In particular, copies of memory can be created at runtime that are uncontrollable and untrackable. Moreover, the operating system can move and copy memory without hindrance. As a result, this might allow an attacker to read the private key from RAM.

Impact

If successful, an attacker is able to access a private key and has full control over the wallet and its funds.

Feasibility

Low. Since locating the private key footprint in the RAM is difficult, it would also be difficult to carry out such an attack in a real world application. However, this is a general concern that has also been noted in the Apple Security Development Checklist.

Mitigation

Since this issue is based on the expected behavior of Swift, we recommend that the development team research potential mitigation strategies that align with industry best practices.

Status

Since research for a mitigation strategy by the development team drew inconclusive results, our team suggested adhering to Apple's Security Development Checklists, specifically as it pertains to the following:

“**Scrub (zero) user passwords from memory after validation:** Passwords must be kept in memory for the minimum amount of time possible and should be written over, not just released, when no longer needed. It is possible to read data out of memory even if the application no longer has pointers to it.”

Notifying implementers of TezosKit about security best practices adequately mitigates the issue, at this point. As a result, a comment alerting that there is a secret key stored in memory when using the `Wallet` class has been added to the code, linking to Apple's best practice suggestions. However, we are not certain this will be an appropriate long-term solution as this is a fundamental issue with SWIFT and it is hoped that it will be addressed at that level.

Verification

Resolved.

Issue B: Undefined Behavior in the Tez Class

Location

/TezosKit/Common/Models/Tez.swift

Synopsis

The `Tez` class uses `bignums` to represent the native token of the Tezos currency but can be initialized with `init(_ balance: double)`, which accepts negative values and results in undefined behavior.

Impact

We are not aware of any attack that can be based on this behavior. However, the type `Tez` does not behave according to the Michelson specifications, outlined in Michelson: the language of Smart Contracts in Tezos and Michelson Reference, which might lead to various unexpected problems with calculations.

Feasibility

The problem may arise whenever a developer wants to subtract a certain amount of `Tez` from another `Tez` and uses negative `Tez` and addition to achieve that.

Technical Details

Suppose a user has 2.1 `Tez` and wants to subtract 1.999999 `Tez` from this. The computation is 2.1 `Tez` - 1.999999 `Tez` = 0.100001 `Tez`. However, since `init(_ balance: double)` accepts negative `Tez`, the user can execute something like the following:

```
let tez1 = Tez(2.1)

let tez2 = Tez(-1.999999)

let result = tez1 + tez2
```

This gives the undefined value `result = 1,-899999` which is not a number. This happens because tez2 is the non number '-1.-999999'.

Remediation

Prevent initialization with negative `Tez`. According to the Michelson specifications, the type `Tez` should only be able to store positive values.

Status

A remediation strategy has been implemented so that the `Tez` class now uses `BigUInt` instead of `BigInt` internally to represent `Tez` decimals in addition to ensuring that the initializer `init(_ balance: double)` fails on any attempt to initialize a negative amount of `Tez`.

Verification

Resolved.

Issue C: TezosKit Does Not Support Unbounded `nat` and `int` types

Location

/TezosKit/Common/Michelson/IntMichelsonParameter.swift

Synopsis

According to the Michelson specifications, the types `nat` and `int` are unbounded and the size of the actual instances is only controlled by storage and gas cost. However, this is not reflected in TezosKit as it mixes both types and represents them internally as the Swift signed integer type `Int`. This is equivalent to `Int_32` on 32-bit platforms and `Int_64` on 64-bit platforms.

Impact

Since the TezosKit implementation of both `nat` and `int` is bounded by 32/64 bit signed integers, it is not possible to send parameters or read storage of those types from/to on-chain Tezos Smart contracts that exceed the storage capacities of Swifts `Int` type, leading to various boundary errors. For example, it is not possible to write the number 9,223,372,036,854,775,808 into a `nat type` storage of any Tezos Smart contract using TezosKit.

Remediation

Implement the `IntMichelsonParameter` class using `bignum` instead of `Int` internally. This does not lead to overflow errors, as the storage of Michelson Smart Contract is bounded by storage and gas costs.

Status

A remediation strategy has been implemented so that the `IntMichelsonParameter` class is now split into two representations: `IntMichelsonParameter` and `NatMichelsonParameter` of the Michelson types `int` and `nat`. In addition, `BinInt` and `BigUint` can now be used to account for big numbers.

Verification

Resolved.

Issue D: Hex Seeded Key Initializer is Hard Coded to Ed25519 Curve

Location

Initializers in /TezosKit/Common/Models/Wallet.swift

Synopsis

We found a bug that the curve value was hard coded, despite giving a choice for the user to select a curve from three choices: Ed25519, Secp256k1 and P256. As a result, the function did not respect the parameters selected by the user and forced the wallet to use the Ed25519 curve in all cases.

Impact

The user believes they have selected a particular elliptic curve and, as a result, believes they have selected different security properties.

Remediation

A commit has been added which resolves this by making the selection take effect. As a result, when the wallet or other application uses the `SecretKey` struct, it will initialize the `SecretKey` struct with the signing curve that is passed into the `init()` function, rather than having Ed25519 hard coded as the only signing curve that the struct would be initialized with.

Status

The above remediation was implemented prior to the completion of the security audit and delivery of the audit report.

Verification

Resolved.

Suggestions

Suggestion 1: Implement All Primitive Michelson types in Separate Classes

Location

/TezosKit/Common/Michelson/

Synopsis

The Michelson Specifications distinguish between various primitive types like `nat`, `int`, `address`, `string` and others. However, this is not strictly reflected in the corresponding `MichelsonParameters` in TezosKit. For example, there is no distinction between the `nat` and the `int` type in TezosKit, as both are handled in the `IntMichelsonParameter` class. Moreover TezosKit handles `string`, `timestamp`, `address` and the `key` type inside the `StringMichelsonParameter` class. In addition, `bytes`, `key_hash` and `chain_id` are handled inside the `BytesMichelsonParameter`.

Remediation

We suggest that TezosKit defines separate `MichelsonParameter` classes for all primitive Michelson types so they are strictly in line with Michelson's strong type system. In addition, some basic checks (like proper formats of the `key  type` or the `chain_id`) could be enforced on their `MichelsonParameter` classes.

Status

All Michelson types are now represented by appropriate `MichelsonParameter` classes:

`Date`: https://github.com/keefertaylor/TezosKit/pull/194
`Key`: https://github.com/keefertaylor/TezosKit/pull/195
`Nat`: https://github.com/keefertaylor/TezosKit/pull/196
`Signature/Address`: https://github.com/keefertaylor/TezosKit/pull/197
`Key_Hash/Chain_ID`: https://github.com/keefertaylor/TezosKit/pull/198

Verification

Resolved.

Suggestion 2: Improve Precision in Cryptography Documentation

Location

/TezosKit/Common/Models/Tez.swift
/TezosKit/Common/Models/Wallet.swift
/TezosKit/Crypto/SecretKey.swift

Synopsis

Overall, we found that the documentation is very good. However, we found minor imprecisions in the documentation of various functions. In particular, we found that the initializer `public init?(_ balance: string)` in class Tez, interprets the balance argument string as microTezos, not Tezos, which is slightly misleading given the name of the class. Similarly, we found slight imprecisions in the descriptions of the initializer `init?(secretKey: String, signingCurve: EllipticalCurve = .ed25519)` in class `Wallet`, where the argument `secretKey` requires a seed, not the actual `secretKey` interpreted as a string.

In addition, it was not immediately clear to us how the various constructors in the `Wallet` class relate to the Tezos key derivation, based on the “mnemonic, secret, password, email” scheme, used in this example.

We also found one slight error in the comments in `SecretKey.swift` on line 119, where the comment mentions that the input parameter is of type hexadecimal `String`, when it is of type `[Uint8]` bytes.

Remediation

Improve documentation by checking for and correcting these imprecisions.

Status

The minor errors in the code comments have been corrected. The TezosKit development team has also added a seed input initializer for the wallet to help clarify the difference between seed and secret keys used in Tezos.

Verification

Resolved.

Suggestion 3: Review Dependencies Used in TezosKit

Location

Dependencies managed by Carthage in /TezosKit/Cartfile
EllipticCurveKeyPair dependencies in /TezosKit/Crypto/EllipticCurveKeyPair/

Synopsis

The CryptoSwift 0.14.0 library used by TezosKit is several releases behind (latest release at the time of writing this report is 1.3.1), in which many fixes, improvements and improved integration with Swift 5 has been incorporated. In addition, EllipticCurveKeyPair source code files were copy-pasted into TezosKit codebase.

Mitigation

Review and upgrade release versions of used dependencies when feasible. Use a dependency manager instead of copy-pasting if possible.

Status

The TezosKit development team updated versions of all old dependencies used by the library.

Verification

Resolved.

Suggestion 4: Ensure Secure Environment for Running TezosKit Applications

Synopsis

Allowing applications using TezosKit to run on a Jailbroken iOS makes it less secure and more prone to attacks. Jailbroken phones have no access control on root files, hence rendering the sandbox model useless.

Mitigation

Programatically check if the application using TezosKit is running on a Jailbroken phone and prevent it from executing sensitive operations. Checking Jailbreak can be done by using multiple methods including checking relevant file changes or checking if Cydia is installed.

Status

The TezosKit development team has implemented a function that detects and stops communication with the Tezos Network on Jailbroken devices.

Verification

Resolved.

Suggestion 5: Minimal Amount of Overly Complex Code

Location

/TezosKit/TezosNode/TezosNodeClient.swift

/TezosKit/Extensions/PromiseKit/TezosNode/TezosNodeClient+Promises.swift

Synopsis

Our team ran the TezosKit codebase against the Codebeat static analyzer tool. This tool generally looks at the complexity of the code with deep assignment branch conditions, cyclomatic complexity or control flow paths, lines of code, arity, maximum block nesting, and code duplication.

The results have shown that the default critical threshold of six function arguments was exceeded in multiple places in the locations listed. There is one warning of block nesting too deep with a depth of four in `public func forgeSignPreapplyAndInject()`.

Remediation

While these reports do not reveal an immediate security vulnerability, it is suggested that code block depth never exceeds three levels, and that function arguments remain below a number that makes the function unwieldy to use or understand. These are minor issues that are present in a few locations so the impact is very minimal. Reducing these numbers could slightly improve code quality.

Status

In response to this suggestion, the TezosKit development team has noted that careful consideration has been given to the default thresholds provided by the static analyzer tool. They have reduced stack depth and parameter counts where they feel it is reasonable to do so and have stated that they consider code readability of higher priority than the potential complexity implications that are not present in this case and adhering to a default provided by tooling.

Given that the thresholds for code complexity are not standard, we agree that no additional changes are needed and that maintaining readability is more desirable than reduced complexity in this case.

Verification

Invalid.

Suggestion 6: Warn Users of Incapability of Backing Up Enclave Key

Location

/TezosKit/Examples/SecureEnclave/ViewController.swift

Synopsis

As mentioned in this detailed article by the author of the TezosKit, the secure enclave does not have the ability to export the P256 private key that it uses to generate the Tz3 accounts. If the T2 chip malfunctions, then access to that account will be lost and there will be no way to recover the account. The severity of this issue does not seem to be broadcast to a user that is creating an account with the secure enclave. Users may not understand the importance of being able to recover cryptocurrency keys.

Mitigation

Add a warning message to the creation UI of the secure enclave key that will inform the user that their key will have no options for recovery if their device malfunctions. A link to the article or a provided guide on how to use the enclave as a multisig signer could be a useful feature that will prevent misunderstanding and loss of access in the future.

Status

The TezosKit development team has implemented a warning message on the example UI of the wallet. Furthermore, they have added comments to the example in an effort to address this issue for implementers of secure enclave key storage in the future.

Verification

Resolved.

Recommendations

We commend the TezosKit team for addressing all of the *Issues* and *Suggestions* stated above, prior to the follow up verification by the auditing team.

Additionally, we recommend some effort be invested into further refining the handling of the cryptography, such as cleaning up the documentation to be more precise and improving the curve selection. This can be done by implementing P256 or removing the fatal error when initializing a private key on P256 or by documenting and separating the initialization of this curve’s private key on the enclave or external to the enclave. Furthermore, we recommend investigating additional backup methods and incorporating backup key documentation for implementers of the TezosKit library, in the case of deletion of the private key in Swift as the keys can be stored in multiple locations.

Overall, the approach to security in TezosKit is commendable and we encourage the team to continue making it a priority as development continues, updates are made, and new features are introduced.

About Least Authority

We believe that people have a fundamental right to privacy and that the use of secure solutions enables people to more freely use the Internet and other connected technologies. We provide security consulting services to help others make their solutions more resistant to unauthorized access to data and unintended manipulation of the system. We support teams from the design phase through the production launch and after.

The Least Authority team has skills for reviewing code in C, C++, Python, Haskell, Rust, Node.js, Solidity, Go, and JavaScript for common security vulnerabilities and specific attack vectors. The team has reviewed implementations of cryptographic protocols and distributed system architecture, including in cryptocurrency, blockchains, payments, and smart contracts. Additionally, the team can utilize various tools to scan code and networks and build custom tools as necessary.

Least Authority was formed in 2011 to create and further empower freedom-compatible technologies. We moved the company to Berlin in 2016 and continue to expand our efforts. Although we are a small team, we believe that we can have a significant impact on the world by being transparent and open about the work we do.

For more information about our security consulting, please visit https://leastauthority.com/security-consulting/.

Our Methodology

We like to work with a transparent process and make our reviews a collaborative effort. The goals of our security audits are to improve the quality of systems we review and aim for sufficient remediation to help protect users. The following is the methodology we use in our security audit process.

Manual Code Review

In manually reviewing all of the code, we look for any potential issues with code logic, error handling, protocol and header parsing, cryptographic errors, and random number generators. We also watch for areas where more defensive programming could reduce the risk of future mistakes and speed up future audits. Although our primary focus is on the in-scope code, we examine dependency code and behavior when it is relevant to a particular line of investigation.

Vulnerability Analysis

Our audit techniques included manual code analysis, user interface interaction, and whitebox penetration testing. We look at the project's web site to get a high level understanding of what functionality the software under review provides. We then meet with the developers to gain an appreciation of their vision of the software. We install and use the relevant software, exploring the user interactions and roles. While we do this, we brainstorm threat models and attack surfaces. We read design documentation, review other audit results, search for similar projects, examine source code dependencies, skim open issue tickets, and generally investigate details other than the implementation. We hypothesize what vulnerabilities may be present, creating Issue entries, and for each we follow the following Issue Investigation and Remediation process.

Documenting Results

We follow a conservative, transparent process for analyzing potential security vulnerabilities and seeing them through successful remediation. Whenever a potential issue is discovered, we immediately create an Issue entry for it in this document, even though we have not yet verified the feasibility and impact of the issue. This process is conservative because we document our suspicions early even if they are later shown to not represent exploitable vulnerabilities. We generally follow a process of first documenting the suspicion with unresolved questions, then confirming the issue through code analysis, live experimentation, or automated tests. Code analysis is the most tentative, and we strive to provide test code, log captures, or screenshots demonstrating our confirmation. After this we analyze the feasibility of an attack in a live system.

Suggested Solutions

We search for immediate mitigations that live deployments can take, and finally we suggest the requirements for remediation engineering for future releases. The mitigation and remediation recommendations should be scrutinized by the developers and deployment engineers, and successful mitigation and remediation is an ongoing collaborative process after we deliver our report, and before the details are made public.

Responsible Disclosure

Before our report or any details about our findings and suggested solutions are made public, we like to work with your team to find reasonable outcomes that can be addressed as soon as possible without an overly negative impact on pre-existing plans. Although the handling of issues must be done on a case-by-case basis, we always like to agree on a timeline for resolution that balances the impact on the users and the needs of your project team. We take this agreed timeline into account before publishing any reports to avoid the necessity for full disclosure.