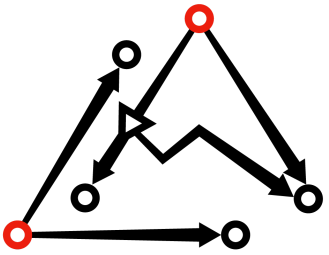

Least Authority

PRIVACY MATTERS

Kolibri Smart Contracts

Security Audit Report

Tezos Foundation

Final Report Version: 9 March 2021

Table of Contents

This audit makes no statements or warranties and is for discussion purposes only.

Overview

Background

The Tezos Foundation has requested that Least Authority perform a security audit of the Kolibri Smart Contracts. Kolibri is a set of Smart Contracts on Tezos which can be used to issue kUSD, a trustless, algorithmic stablecoin that is built on Collateralized Debt Positions of XTZ implementing the FA1.2 Standard.

Tezos is a decentralized blockchain that governs itself by establishing a true digital commonwealth. It facilitates formal verification, a technique which mathematically proves the correctness of the code governing transactions and boosts the security of the most sensitive or financially weighted smart contracts.

Project Dates

- **January 18 - February 12** Code review *(Completed)*
- **February 17**: Delivery of Initial Audit Report *(Completed)*
- **March 8:** Verification completed *(Completed)*
- **March 9:** Delivery of Final Audit Report *(Completed)*

Review Team

- Sajith Sasidharan, Security Researcher and Engineer
- Bryan White, Security Researcher and Engineer
- Nathan Ginnever, Security Researcher and Engineer

Coverage

Target Code and Revision

For this audit, we performed research, investigation, and review of the Kolibri Smart Contracts 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:

- https://github.com/Hover-Labs/kolibri/releases/tag/la-audit

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

`2454a0c1b80fa94a62bf6b67b3b4a6acb3db820d`

For the review, this repository was cloned for use during the audit and for reference in this report:

https://github.com/LeastAuthority/kolibri

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

In addition, any dependency and third party code, unless specifically mentioned as in-scope, were considered out of scope for this review.

Supporting Documentation

The following documentation was available to the review team:

- FA1.2 Standard (TZIP-7)

This audit makes no statements or warranties and is for discussion purposes only.

- Kolibri Documentation
- Article: "Hello, Kolibri"
- Article: "Problems with Balance"
- Article: "Smart Contract Vulnerabilities"

In addition, this audit report references the following documents:

- Article: "The Open Price Feed"
- Least Authority's Thanos Wallet Security Audit Report
- Least Authority's Taquito Security Audit Report
- A. A. Letichevsky, O. A. Letychevskyi, V. S. Peschanenko, 2012. "Insertion Modeling System". In: Clarke E., Virbitskaite I., Voronkov A. (eds.) *Perspectives of Systems Informatics. PSI 2011. Lecture Notes in Computer Science*, vol 7162. Springer, Berlin, Heidelberg. [LLP12]
- O. Letychevskyi, V. Peschanenko, V. Radchenko, M. Poltoratskyi, Y. Tarasich, 2019, "Formalization and Algebraic Modeling of Tokenomics Projects". *Proceedings of the 15th International Conference on ICT in Education, Research and Industrial Applications. Integration, Harmonization and Knowledge Transfer.Part III: 3nd International Workshop on Rigorous Methods in Software Engineering (RMSE 2019)*, pp. 577–584. [LPR+19]

Areas of Concern

Our investigation focused on the following areas:

- Correctness of the implementation;
- Adherence to the specification and best practices;
- Adversarial actions and other attacks on the smart contracts;
- Potential misuse and gaming of the smart contracts;
- Attacks that impacts funds, such as the draining or the manipulation of funds;
- Mismanagement of funds via transactions;
- Denial of Service (DoS) and security exploits that would impact the smart contracts intended use or disrupt the execution of the contract;
- Vulnerabilities in the smart smart contracts code;
- Protection against malicious attacks and other ways to exploit the smart contracts;
- 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

During the security audit, our team did not identify any security-critical *Issues* in the Kolibri system, however, we have proposed several *Suggestions* for improvement.

System Design

We commend the Kolibri team for their strong considerations for security, which are evident and demonstrated throughout Kolibri's system design.

Kolibri uses the Tezos token (XTZ) as collateral in Collateralized Debt Positions (CDPs). A CDP contract is referred to as an `oven`, against which users can borrow kUSD, a USD soft-pegged stablecoin (i.e. currency that is collateralized by USD).The smart contracts implement access control and state transition checks, which are used appropriately throughout the system and is consistent with established best practices.

This audit makes no statements or warranties and is for discussion purposes only.

This is demonstrated in the use of proxy and factory contracts for the oven. Proxy contracts allow for upgradability in an otherwise immutable blockchain, which provides a significant advantage when fixes need to be implemented. Factory contracts in the Kolibri system have a security function, such that they ensure that all ovens are printed in a deterministic, trusted manner and are registered to a visible registry for later use. This promotes resolving trust issues in a decentralized environment. As a result, these design features allow oven contracts, which hold user funds, to be trustworthy and immutable.

Other examples of conforming to security best practices are the careful handling of callbacks and the decentralization of permissions in the Tezos environment. In addition, we examined the constructors and setup functions of the smart contracts and the corresponding SmartPy-provided `init` function and identified no issues.

Message Passing and Callbacks

Tezos uses a callback-style message passing pattern and orders callbacks by Breadth-First Search (BFS), in contrast to the direct message callback-style used by Ethereum. While this may potentially lead to confusion on how developers can secure their smart contracts against attacks, such as callback authorization bypass and call injection, the Kolibri team has mitigated these vulnerabilities by ensuring that the contract interactions do not require passing data back to the caller. In cases that the data is required to be passed back to the callback function, the function is properly permissioned and handles the returned data in an expected and orderly way.

In addition, we looked at the smart contract interactions for any issues caused by the BFS ordering that have been present in Tezos. We verified that the permission and state machine transition checks in place ensure the callbacks happen as expected.

Liquidation Mechanism

Kolibri utilizes compound interest accrual to incentivize users to close their positions and uses a system of interest rate indices to compute compound interest on an unbounded number of the oven contracts, as described in the project documentation. Furthermore, accrued interest is paid out to the stability fund through the developer fund, which secures the network by liquidating underwater ovens while rational actors would not liquidate At the protocol level, the network is secured by a liquidation mechanism, which is leveraged as an incentive mechanism, providing stability to the stablecoin price. This mechanism is also automatically activated to recover ovens that are underwater. However, since this mechanism is funded by the stability fund, it is not a viable standalone solution and is intended as a backup only. As a result, users are increasingly incentivized to liquidate undercollateralized ovens as there is a negative correlation between the collateralization ratio and the net payout, which prioritizes the liquidation of the highest risk ovens in the network. The network's liquidation of undercollateralized positions through this liquidation mechanism reduces the burden on the stability fund and produces competition between prospective liquidators, thus sustaining the value of the stablecoin.

Governance

Governance functionality and launch limitations also demonstrate consideration for security. Importantly, the oven's functionality is modified to only be accessible by the owner of the oven and not the governance contract. As a result, smart contracts holding user funds cannot be seized by any governance mechanism. Furthermore, the modifiers for governance and permissions restrict execution to governance and special roles as expected. These roles modify critical functions that handle global states, such as fees, pausing the system, or proxy endpoints being changed for upgradability reasons.

Oracle Price Feed

We considered the oracle and its source of price feed data and found that Kolibri's implementation of Harbinger is robust in how it introduces price feeds to the smart contracts. The Kolibri team's solution considers the trust trade-offs of decentralized oracles and their role in Decentralized Finance (DeFi),

This audit makes no statements or warranties and is for discussion purposes only.

which is based on the Compound Protocol's Open Price Feed. This approach does not rely on an on-chain price oracle, such as Uniswap, which has been previously prone to arbitrage attacks. Instead, it pulls data from centralized exchanges that are likely harder to manipulate than Automated Market Maker (AMM) pools, thus reducing this potential attack vector.

Scope

The audit scope encompassed all components of the Kolibri implementation, including a comprehensive review of the SmartPy smart contracts and TypeScript SDK. However, we identified several out-of-scope dependencies and components that directly impact the overall security of the Kolibri system. We recommend separate, independent security audits or further analysis by subject matter experts, for the following components, as detailed below.

SmartPy Compiler

The smart contracts are implemented in SmartPy that compiles into Michelson. In reviewing the SmartPy CLI to determine the possibility for the generation of untrustworthy Michelson code, we were unable to determine with certainty that the compiler binary corresponds to its published source code. As a result, there are no checks or guarantees against the compiler introducing malicious code. Thus, we recommend a security audit of the SmartPy compiler (Suggestion 4).

Thanos Wallet DApp Module

We examined the SDK to determine the way in which it accesses the network and noted that it uses the `thanos-wallet/dapp` module, a light wrapper for the Taquito library used by Kolibri for RPC communication. While Least Authority has audited both Taquito (see audit report) and the Thanos Wallet (see audit report), the scopes of those audits did not include the `dapp` Module, which we recommend be audited for potential vulnerabilities (Suggestion 6).

Token Economics

We investigated the economics of liquidation to consider its effectiveness as an incentivization mechanism to recover underwater `ovens`. While a full economic investigation is not in scope for this audit, we did not identify any issues in our preliminary investigation. However, we recommend that the token economics be further reviewed by a team of subject matter experts, with the expertise to run a tokenomics simulation, which can help determine complex situations that may potentially lead to vulnerabilities (Suggestion 7).

Formal Verification

In addition to manual code reviews, concurrent distributed systems benefit from formal verification, which is less prone to human error in identifying potential vulnerabilities in the core logic of the smart contracts. We recommend that the Kolibri team explore opportunities to conduct formal verification of the smart contracts and the high-level logic (Suggestion 5).

Code Quality

The code base is well organized and demonstrates a clear separation of concerns. Each module is defined, allowing an easy understanding of the system and facilitating a more efficient review for potential security issues.

We examined the tests and deploy scripts to better understand how the system integrates the components beyond the supplied documentation and found that the smart contracts and the SDK include sufficient test coverage.

This audit makes no statements or warranties and is for discussion purposes only.

We found that the SDK code conforms to TypeScript standards and best practices. We also found that the smart contracts adhere to the SmartPy Reference Manual. Given that SmartPy is still in the early stages of development, best practices and style guides enforced by linters, static analyzers, and other testing frameworks have not yet been firmly established.

Documentation

The project documentation is comprehensive, well-organized, and easy to read. We did not find any major inconsistencies between the smart contract code and the project documentation. However, we identified areas for minor improvement, including correcting typos in the project documentation (Suggestion 1) and better definition of terminology in the liquidation mechanism documentation (Suggestion 2).

The smart contract and SDK code is sufficiently commented and easy to follow. The comments in the SDK tests are particularly helpful, as they describe the behavior being tested along-side the test code, which facilitates easy code maintenance and review.

Specific Issues & Suggestions

We list the issues and suggestions found during the review, 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
Suggestion 1: Correct Typos in Documentation	Unresolved
Suggestion 2: Improve Liquidation Documentation	Unresolved
Suggestion 3: Eliminate Overlapping Error Constants	Unresolved
Suggestion 4: Audit SmartPy Compiler and CLI	Unresolved
Suggestion 5: Implement Formal Verification	Unresolved
Suggestion 6: Audit the Thanos Wallet App Module	Unresolved
Suggestion 7: Run a Tokenomics Simulation	Unresolved

Suggestions

Suggestion 1: Correct Typos in Documentation

Location

/pull/2

/pull/1

Synopsis

We identified several typos and a broken link in the project documentation, as described in the pull requests above. We suggest further proof-reading of the documentation to aid in avoiding potential

This audit makes no statements or warranties and is for discussion purposes only.

misunderstandings caused by mistakes in the text and to help to ensure that information is being conveyed clearly and adequately.

Mitigation

We recommend fixing the identified typos and the broken link, in addition to including a proof-reading stage into the documentation editing process for all future changes to the project documentation.

Status

The Kolibri team has acknowledged this suggestion and stated that they intend to implement the suggested changes to the documentation in the near future. As a result, the suggested mitigation remains unresolved at the time of this verification.

Verification

Unresolved.

Suggestion 2: Improve Liquidation Documentation

Location

/documentation/public/liquidation/overview.md

Synopsis

Under the "Liquidation" header, the second point describes how the liquidation fee is applied using undefined terminology:

> "percentage based **liquidation fee** assessed on the assets."

This description does not state explicitly how the liquidation fee is applied. It is possible to attain a more complete description from the example further below in `overview.md`, however, this requires unnecessary effort on the part of the reader.

Mitigation

We recommend that specifications be as explicit and unambiguous as possible, in order to avoid unintentional differences in implementation caused by differing interpretations. We suggest the following improvements.

Change the existing text by implementing one of the following suggestions:

1. Add a definition for all terms used ("assessed", "assets"); **or**
2. Restate the description using terms that are already defined.

   For example, use:
   "The liquidator repays all outstanding kUSD tokens, plus an additional liquidation fee of 10% of the outstanding kUSD."

   Instead of:
   "The liquidator repays all outstanding kUSD tokens, plus an additional percentage based liquidation fee assessed on the assets."

The application could also be described in a more formal manner (e.g. as a mathematical expression). This can be an alternative or an addition to the corrections suggested in point 1, above.

This audit makes no statements or warranties and is for discussion purposes only.

Status

The Kolibri team has acknowledged this suggestion and stated that they intend to implement the suggested changes to the documentation in the near future. As a result, the suggested mitigation remains unresolved at the time of this verification.

Verification

Unresolved.

Suggestion 3: Eliminate Overlapping Error Constants

Location

/smart_contracts/common/errors.py#L73-L77

Synopsis

The module `smart_contracts/common/errors.py` defines some constants to represent errors. The constants `TOKEN_INSUFFICIENT_BALANCE` and `TOKEN_UNSAFE_ALLOWANCE_CHANGE` are assigned the same value. This could potentially lead to confusion for error messages raised from `token.py`, which could result in incorrect or misdirected actions aimed at mitigating errors.

Mitigation

We recommend exploring a solution that automatically numbers constants. If such a solution cannot be implemented, a manual review should eliminate the same value being assigned to multiple constants.

Status

The Kolibri team has responded that they do not intend to fix the overlapping error code as it would require redeploying the entire system and acknowledged that they are willing to accept this trade-off. If the opportunity allows, we recommend that the Kolibri team reconsider implementing this mitigation in the future.

Verification

Unresolved.

Suggestion 4: Audit SmartPy Compiler and CLI

Location

https://smartpy.io/

https://smartpy.io/cli/

https://gitlab.com/SmartPy/smartpy

Synopsis

Kolibri smart contracts are built using the SmartPy compiler and CLI. Kolibri's SmartPy code compiles to Michelson code, which gets deployed on the Tezos blockchain. While it is possible to review the Michelson code to make sure that no malicious code has been introduced by SmartPy, It is a challenging task for human readers to make sense of Michelson code that contains complex business logic. This complexity can result in missed security issues and vulnerabilities. As a result, trusting the SmartPy compiler and CLI is security-critical.

The SmartPy compiler was out-of-scope and we did not perform an in-depth analysis as part of this review. Given that the compiler is a crucial dependency of the Kolibri system, we recommend that it be

This audit makes no statements or warranties and is for discussion purposes only.

examined more thoroughly. While SmartPy appears to be well-engineered and suitable for the Kolibri system, some aspects of SmartPy are a cause for concern, as noted below.

SmartPy has not been audited, however, TQ Tezos have indicated that an audit will be completed in the future but did not provide a specific time frame for the review. In addition, SmartPy has a semi-closed development model and appears to be managed by a single contributor or a small team. The commits to SmartPy's public repository are made by a pseudonymous committer. As a result, the contributor(s) have the ability to feed harmful code into the compiler, presenting a significant risk.

In addition, SmartPy CLI has no versioned releases. There have been a series of revisions to the web-based SmartPy.io, although it is unclear what changes each revision has introduced. At present, the public repository contains four code commits and no tags. As a result, it is difficult to verify which version of SmartPy produced the corresponding Michelson code.

Installing SmartPy CLI requires trusting a shell script downloaded from the Internet, using the command `sh <(curl -s https://smartpy.io/cli/install.sh)`. This command sets up a working SmartPy installation only on macOS. While it is intended to successfully set up a working SmartPy installation of Linux, we have found the SmartPy CLI installation succeeded on an Arch machine but failed to install on a standard Debian stable machine. Although this failure itself is not a security concern, it suggests that SmartPy is potentially under tested.

Finally, while the `smartpyc` binary installed on macOS appears to work without issues, it cannot be considered secure as there is little visibility to how it was built or if it corresponds to the published sources.

Mitigation

We suggest a comprehensive audit of the SmartPy compiler and CLI. Furthermore, the SmartPy team should be encouraged by its community of users to expand testing and to improve their development and release practices to provide more visibility and transparency about contributors and versioning.

Although we understand these suggestions may not be immediately actionable and may require significant resources, malicious code introduced by the compiler is a significant risk with potentially subversive consequences. In his seminal 1984 Turing Award Lecture, Reflections on Trusting Trust, Ken Thompson describes a Trojan horse built into a C compiler. The software tools we use today are far more complex, thus being able to trust tools is pertinent, as evidenced by Solidity's List of Known Bugs.

Status

The Kolibri team has responded that there are ongoing efforts within the Tezos community to prioritize a security audit of the Smarty compiler and CLI, although it has not been conducted at the time of this verification.

Verification

Unresolved.

Suggestion 5: Implement Formal Verification

Location

/master/smart_contracts/

This audit makes no statements or warranties and is for discussion purposes only.

Synopsis

In addition to manual code reviews, concurrent distributed systems benefit from formal verification, which is less prone to human error and is likely to uncover potential security issues.

Facilitating formal verification is a design feature of Michelson. Runtime Verification has announced that they are working with the Tezos Foundation to develop a formal verification framework for Michelson, which is promising and demonstrates progress towards formal verification becoming more easily accessible to the Tezos ecosystem.

Mitigation

Create a formal specification and implement formal verification of the Kolibri system. For example, a similar use case for MakerDAO has been implemented in collaboration with Runtime Verification.

Status

The Kolibri team has responded that they are not currently considering formal verification of the Kolibri smart contracts, but that they may reconsider formal verification in the future.

Verification

Unresolved.

Suggestion 6: Audit the Thanos Wallet dapp Module

Location

https://github.com/madfish-solutions/thanoswallet-dapp#readme

Synopsis

The Kolibri SDK utilizes the `thanos-wallet/dapp` module, a light wrapper for the Taquito library, for RPC communication. While Least Authority has audited both Taquito (see audit report) and the Thanos Wallet (see audit report), the audit scope did not include the dapp Module.

Mitigation

We recommend a security audit of the Thanos Wallet dapp module to check for potential vulnerabilities and to ensure that the interaction of the components function as intended.

Status

The Kolibri team has responded that they do not intend to pursue a follow up audit of the Thanos dapp module, since an audit of the Thanos Wallet has already been completed. However, while the Thanos Wallet has been audited by our team, the dapp module was out of scope. To the best of our knowledge, an audit of the dapp module has not been completed and we recommend that one be conducted by an independent auditing team.

Verification

Unresolved.

Suggestion 7: Run a Tokenomics Simulation

Synopsis

According to the Kolibri team, formal economic analysis of the Kolibri tokenomics has not been completed. Simulation is commonly used for such analyses, in order to test whether assumptions that the incentives rely on are sustainable under realistic conditions, as opposed to strictly statistical analysis.

Simulation models are better suited to take into account the emergent possibilities that can result from a bottom-up approach. Agent-based models are known for being particularly effective (e.g. "An Analysis of the Market Risk to Participants in the Compound Protocol"; Agent-Based Simulations of Blockchain protocols illustrated via Kadena's Chainweb). In addition, Insertion modeling has appeared in the literature [LLP12] [LPR+19].

Mitigation

We recommend a formal economic analysis of the economic assumptions of the Kolibri system.

Status

The Kolibri team has responded that they are actively pursuing a formal economic analysis, however, they have not identified a timeframe for completion.

Verification

Unresolved.

This audit makes no statements or warranties and is for discussion purposes only.

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.