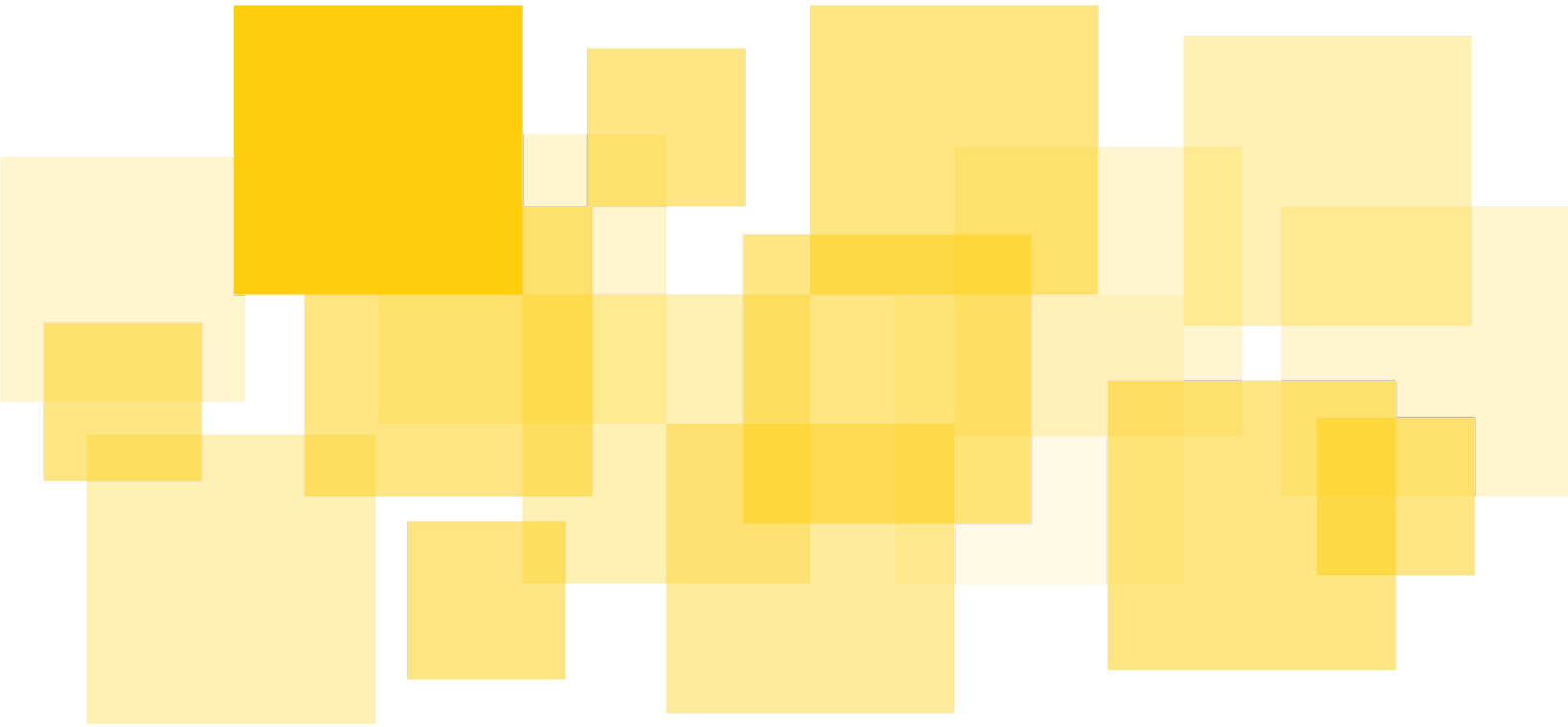
Security Audit Report

Folks Finance

Delivered: February 1, 2022



Prepared for Folks Finance by

runtime verification

Contents

Summary

Folks Finance engaged Runtime Verification to conduct a security audit of the smart contracts implementing lending- and borrowing-related operations in the Folks Finance application.

This was the second phase of the auditing engagement, after the design review phase. The objective of this second phase was to review the contracts' business logic and implementation in PyTeal and identify any issues that could potentially cause the system to malfunction or be exploited.

This second phase of the audit was conducted over a period of four weeks (December 6-17, 2021 and January 10-21, 2022). It led to identifying one critical issue, where an attacker could potentially drain the holdings of a dispenser contract, and a low-severity issue, where certain loan operations may get disabled when the price ratio of the token pair becomes extremely high. The audit also identified several informative findings. The issues and informative findings highlighted in the report were addressed.

The contracts' source code was very well structured and documented. We have enjoyed working with the team at Folks Finance, who have provided us with any assistance possible to make the audit process smooth and productive.

Goal

The goal of the audit was twofold:

1. Review the updated design documents and study the changes made since the design phase review was completed
2. Review the PyTeal implementation of the contracts to identify any discrepancies with the design and any potential code-level issues

The audit focused on trying to identify issues in the system's logic and its implementation that could potentially render the system vulnerable to attacks or cause it to malfunction. Furthermore, the audit highlighted informative findings that could be used to improve robustness, performance and/or efficiency of the implementation.

Scope

The audit was conducted on the following artifacts provided by Folks Finance:

- Application design documents (updated since the design review phase):
  - Protocol Overview
  - Economic Model
  - Technical Design
  - Change Log
- Smart contracts in PyTeal:
  - Dispenser smart contract (`dispenser_approval_program.py`)
  - Token pair smart contract (`token_pair_approval_program.py`)
  - Oracle adapter contract (`oracle_adapter_approval_program.py`)
  - Oracle smart contract (`oracle_approval_program.py`)
  - Library functions and state definitions used by the contracts above (contents of `common`, `dispenser` and `oracle_adapter` subdirectories)

The audit was based on commit `7f445aff68b3b76891197f6d83cd745697bac381` on Folks Finance's private GitHub repository. It was a best-effort audit. In particular, the liquidity approval program and the staking programs were not in scope.

Methodology

This is a best-effort, mostly manual audit of the contracts in scope. The codebase is relatively large and the logic that the contracts implement is non-trivial.

All contracts are implemented in PyTeal, a Python-embedded domain-specific language for writing TEAL programs. Given the complexity of the code, the audit focused primarily on the PyTeal code rather than the compiled version of the contracts in TEAL.

In the process of trying to build a deeper understanding of what the implementation does, a specification in a Python-like language was derived from the implementation of the dispenser contract (the largest contract amongst the contracts in scope). It abstracted away a few aspects of PyTeal, such as scratch memory, but modeled other details, such as transaction fields and global and local state access.

In addition to the specification above, we manually reviewed the PyTeal code against the design and the specification of the various transaction groups. This process involved a series of checks, including:

- Checking that values of transaction fields are checked if needed
- Checking that parameters that are given as user input is properly validated
- Investigating potential panic behaviors due to arithmetic problems
- Investigating the effects of rounding errors and whether they may cause problems
- Deriving possible contract invariants and manually reasoning about their validity

A combination of modelling and manual code review has enabled us to identify the issues highlighted in this report.

Over the whole timeline of the audit, we maintained a channel of communication with the team at Folks Finance. The discussions of the on-going progress and reporting of findings in realtime enabled us to be more effective and efficient.

For classifying findings, we have adopted a severity classification system that is inspired by the one of Immunefi.

Disclaimer

This report does not constitute legal or investment advice. The preparers of this report present it as an informational exercise documenting the due diligence involved in the secure development of the target contract, and make no material claims or guarantees concerning the contract's operation post-deployment. The preparers of this report assume no liability for any and all potential consequences of the deployment or use of this contract.

Smart contracts are a nascent software arena, and their deployment and public offering carries risk. This report makes no claims that its analysis is fully comprehensive. The possibility of human error in the manual review process is real. We recommend always seeking multiple independent opinions and audits.

Findings

Findings presented in this section are issues that can cause the system to fail, malfunction, and/or be exploited, and should be properly addressed.

A01. Draining of Funds from a Dispenser Contract (Critical)

Context

Each dispenser smart contract maintains a specific asset: either Algo or an Algorand Standard Asset (ASA), along with the f-asset and fr-asset ASAs (created by the contract itself) for that specific asset. For example, the Algo dispenser maintains the deposited Algos along with the fAlgo and frAlgo ASAs. The f-asset accrues interest with time, while the fr-asset represents rewards that are distributed to users to incentivize providing liquidity and participating in the protocol. Therefore, an A-dispenser for an asset A is able to receive A, fA and frA assets.

When a user makes a deposit, say an amount X of Algos, the user receives a corresponding amount M of fAlgo calculated by the protocol, which represents the deposited amount and the accrued interest. The user may also have earned frAlgo by providing liquidity or by borrowing and repaying loans (or through some other means external to the protocol, like buying frAlgo on a third-party exchange). The user may later decide to redeem his fAlgo tokens for an amount of Algo that will be equal to his original deposits of Algo plus the interest accrued.

Issue

The redeem logic does not check that the user is actually sending the expected f-asset, e.g. fAlgo, to the dispenser. Since an A-dispenser may receive its target asset A and the f-asset and fr-asset versions (fA and frA) of the asset (and nothing else), an attacker could transfer an amount of the target asset A or the fr-asset frA instead to the A-dispenser to attempt to redeem without sending the interest-accruing f-asset fA.

There are two ways in which this can potentially be exploited:

1. The attacker attempts to redeem by sending an amount X of A to the dispenser. In this case, the dispenser returns an amount $M > X$ of the asset A back to the attacker. The amount M includes interest earned since the deposit was made. This means that the attacker can repeat this process indefinitely to slowly drain the dispenser of its holdings of A (as time needs to pass for interest to accrue). Note that this scenario is not possible with Algo-dispensers since the logic checks that the transaction is actually an asset transfer transaction.
2. The attacker attempts to redeem by sending an amount X of frA to the dispenser. In this case, the dispenser returns an amount M of asset A to the attacker. In situations where the value of A is greater than that of frA, the attacker will have a financial incentive to repeat this process as long as he is able to obtain frA (through the protocol

or otherwise), which could potentially result in fully draining the dispenser's funds. Note that this works also with Algo-dispensers.

Note that in both scenarios, the attacker would still keep his stash of fA tokens.

Recommendation

In the `on_redeem` logic of the dispenser contract, check that the asset ID in the asset transfer made by the user (Txn indexed 1 in the redeem transaction group) matches the asset ID of the f-asset of the dispenser.

Status

This issue has been addressed.

A02. Blocking Certain Loan Operations When the Token Pair Price Ratio is Extreme (Low)

Context

There are certain operations that require the price feed of an oracle. They are: borrowing, increasing the borrowed amount, reducing the collateral and liquidating a loan. For example, to perform a borrow operation, the application needs to get the current price ratio of the collateral asset (with price C) to the asset being borrowed (with price B). This price ratio is provided by the oracle contract, through the oracle adapter contract.

The oracle adapter, when invoked with a pair of asset IDs arguments, calculates two quantities:

1. The conversion rate between the collateral asset and borrow asset: C/B
2. The number of decimals the conversion rate value has, which is 18 if $C < B$, or $18 - \lceil \log_{10}(C/B) \rceil$ otherwise.

These two quantities are made available to later transactions in the transaction group through shared scratch memory slots.

Issue

There is no ceiling or log functions that are readily available in PyTeal, so the expression above that calculates the decimal places in the conversion rate is computed using a counter-controlled `for` loop that counts how many orders of magnitude the price C is larger than the price B, and deducts this value from 18. The loop counter counts down from 18 down to 0 (so the loop body is executed at most 19 times).

In Algorand, an asset can have at most 19 decimal places, which is the maximum number of decimals representable in a uint64. Therefore, in the most extreme case, it is possible to have the price C be more than 19 orders of magnitude larger than B, e.g. $C = 1.0000000000000000001 \times 10^{19}$ and $B = 1$. In this case, evaluating the expression

$\lceil \log_{10}(C/B) \rceil$ means that the for loop iterates 19 times through its body and decrements its counter after each iteration. Since the counter begins at 18, the last iteration of the loop will cause underflow and execution will panic.

Notice that in this isolated case, as long as the price ratio C/B stays this extremely high, the protocol will be unable to process any borrow, increase-borrow, reduce-collateral or liquidate operations for this pair (with C the price of the collateral asset and B the price of the borrow asset). Other pairs are not affected.

This issue is classified "Low" because it represents a case that is unlikely to occur in practice for the typical types of assets to be supported by the application.

Recommendation

Augment the loop condition with a conjunct that checks that the loop counter is strictly greater than 0. This means though in the extreme case presented above, there may be loss of precision, but this is expected if the price ratio is that extreme.

Generally, token pairs with such extreme price ratios should not be supported, and a plan should be put in place to deal with existing pairs whose price ratios become extreme.

Status

This issue has been addressed.

Informative Findings

Findings presented in this section do not necessarily represent any flaw in the code itself. However, they indicate areas where the code may need some external support or areas where the code deviates from best practices.

B01. Locking Up Small Amounts of User Funds in Some Extreme Cases due to Rounding

Description

This finding was mostly self-reported. It was initiated by a discussion of allowing the user to supply 0 amounts for deposits, redeems, borrows and other other user-initiated operations. When these cases succeed, they reduce to updating interest rates and indices, for which there exists an operation `on_update_indexes` in the dispenser contract that allows users to do just that. So the initial question was that why allow 0-value deposits, redeems, borrows … etc, and the main reason was that to essentially keep the logic simple and avoid having to complicate it unnecessarily.

This raised the question of whether scenarios where the user is getting 0 amounts in return should be rejected. For example, if a user deposits 1 Algo into the Algo dispenser, they are likely to get 0 fAlgo in return due to rounding with integer division (as the deposit interest index is likely to be greater than 1), which could result in locking up 1 Algo in the dispenser. This led to conducting a thorough investigation of cases that could to locking up of small amounts of tokens when the user supplies correspondingly small input amounts.

We summarize below the different cases where this is possible.

1. `on_deposit`: As given in the example above, it is possible that the user gets a 0 amount of fA in return when depositing a small amount of an asset A, due to integer division and the fact that the deposit interest index is likely greater than one.
2. `on_borrow`: It is possible that the user applies a non-zero amount of collateral and receives back a 0 loan in the case that the conversion rate is 0. This can be problematic as when they come back to reclaim their collateral, their borrow balance will be 1 (effectively locking up a portion of the collateral)..
3. `on_reduce_collateral`: It is possible that the user specifies a non-zero amount and receives 0 collateral back in one rather extreme case, which happens when the escrow's collateral balance happens to be exactly equal to one plus the amount of collateral that corresponds to the borrow balance.

Status

The amounts that could potentially be locked up in these cases are very small. The `on_borrow` case was deemed the only case for which a check is to be added as a protection mechanism.

Generally, these cases can only be problematic in practice with assets with 0 decimals and

low total supply, so the value of an individual basic unit is potentially high. Given that there won't be too many of those kinds of assets, and since dispensers are approved and setup only by the platform owner or the DAO, having a warning message to users that the may lose some funds if they deposit very small amounts of an asset or may have to pay more interest if they borrow very small amounts should be sufficient.

B02. Missing Non-critical Safeguards That Would Improve Robustness

Description

There are a few privileged operations that are executed by the platform owner address (or later the DAO), such as creating and setting up dispensers, creating and setting up new token pairs and adding new asset data to an oracle. Since these are privileged operations, it is assumed that only proper contracts are created and setup, which is a reasonable assumption. Nevertheless, given the importance of setting up contracts with the correct parameter values and the proper links, it is advisable to have the logic check and ensure proper setup and creation of these contracts. This can protect against unintentional mistakes that could potentially result in ill-configured contracts. In some cases, misconfigurations can potentially manifest themselves during execution as panic behaviors due to arithmetic problems.

We summarize below the properties to be checked when creating/updating these contracts (some were self-reported after this issue was raised):

- The Dispenser Contract:
  - `U_OPT < 1e14`
  - `RF + SRR <= 1e14`
  - `EPS >= 1e14`
  - Adding a token pair (`on_add_token_pair_opt_in` and `on_add_token_pair`):
    i. The `TokenPairLinkAddr` in Txn0 and Txn1 is the same as the one in Txn 2 and Txn3
    ii. The borrow dispenser passed as a foreign argument in Txn0 is the same app that is being called in txn2 (Similarly, the collateral dispenser passed in Txn2 is the one called in Txn0)
    iii. The `TokenPair` contract passed as a foreign application in Txn0 and Txn2 are the same
- The Token Pair Contract:
  - `S1 <= S2 <= S3`
  - `S1 <= 1e14`
  - `S2 <= 1e14`
  - `S3 >= 1e14`
  - `on_creation`: The two dispenser application IDs given as arguments should be checked to ensure that they refer to two distinct contracts
- The Oracle Contract:
  - When adding a price-timestamp value to the oracle (`on_add` in the oracle approval program), if the asset already exists, the existing value for this asset is overwritten,

which is probably not the intention, since there is already an update transaction defined for this purpose. To avoid inadvertently modifying existing values, it is advised that the state is checked first if the asset already exists, and if so the transaction fails.

Status

This issue has been addressed.

B03. Minor Inconsistencies in the Documentation

Description

The source code of the contracts is amply documented using consistently structured PyTeal comments, which is a feature of the code that has been extremely useful during the audit. However, there were a few inconsistencies in the documentation that we thought should be highlighted to eliminate any confusion when reading the code. We mention the most important ones below:

1. The decimal specifications of `div14` in `math.py` given in its documentation are too restrictive. `div14` is (correctly) called by other functions with the second argument not necessarily having 14dp, which is what the specification seems to require (e.g. the call to `div14` in `calc_utilization_ratio`). The specification should just be generalized to make no assumptions about the decimal specs of the inputs, so something like: if `n1` is Xdp and `n2` is Ydp, then the result is (X+14-Y)dp.
2. In `calc_delta_time`, the comment states that the result should have 14dp, but it should instead have dp matching those of `latest_timestamp()`, namely 0 in this case.

Status

These comments have been addressed.