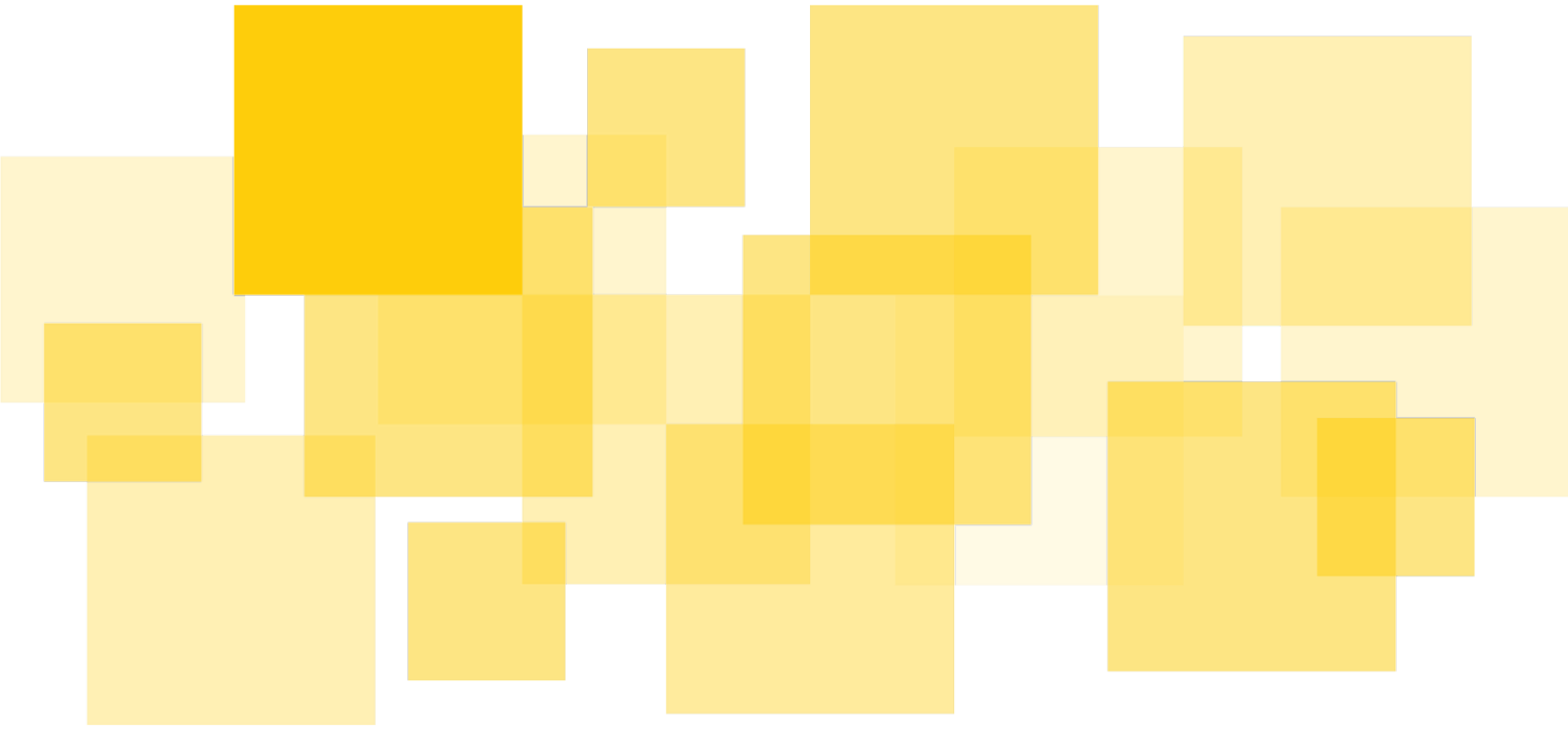
# Security Audit Report

## Algofi

Delivered: December 14, 2021



Prepared for Algofi by

runtime verification

# Contents

## Summary

Algofi engaged Runtime Verification Inc to conduct a security audit of the smart contracts implementing their decentralized lending market.

The objective 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.

The audit was conducted by Mihai Calancea over a period of 7 weeks (18 Oct 2021 - 03 Dec 2021).

The audit led to identifying 2 critical issues, 3 high severity issues, 2 medium severity issues and 4 informative findings and recommendations. One of the high severity issues was reported by Algofi. We generally found the protocol to be thoughtfully engineered and collaborated very well with the Algofi team.

## Scope

The audit was conducted on the following artefacts provided by Algofi:

- Manager Application (manager.py)
- Market Application (market.py)
- A small helper library (globals.py)

The audit focused on two commits in the Algofi GitHub repository:

- Commit 1: ea69095f2bbebb391ed5b9cf172d4dd43e2932ea. This is the initial commit provided by Algofi during week 2 of the audit.
- Commit 2: fd6b2a60f4beb759e9c456b556b1eb5ba7fdab8a. A later iteration containing patches for issues highlighted during the audit and other small changes. This commit was reviewed in the last two weeks of the audit.

This is a best-effort audit. In particular, the code implementing user reward allocation was not in scope.

## Methodology

Both smart contracts are implemented in PyTeal, a Python EDSL for writing TEAL programs. The generated TEAL programs are quite large and lack adequate mapping of code and comments from the PyTeal source. Because of this, the audit focused primarily on the PyTeal code.

A simplified high-level model of the protocol was written in C++. The model abstracts away the Algorand Virtual Machine (AVM) environment and focuses on the accounting logic. It was used to develop intuition about useful invariants and test against scenarios involving a large number of protocol operations.

The severity scale is mostly inspired by this classification. Issues that constitute more nuanced cases include additional justification.

# 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 contracts only, and make no material claims or guarantees concerning the contracts' operation post-deployment. The preparers of this report assume no liability for any and all potential consequences of the deployment or use of these contracts.

Smart contracts are still a nascent software arena, and their deployment and public offering carries substantial risk. This report makes no claims that its analysis is fully comprehensive, and recommends always seeking multiple opinions and audits.

This report is also not comprehensive in scope, excluding a number of components critical to the correct operation of this system.

The possibility of human error in the manual review process is very real, and we recommend seeking multiple independent opinions on any claims which impact a large quantity of funds.

# Issues and findings

Issues and findings are grouped by the most recent commit to which they apply. Both categories are sorted by severity.

# Commit 1

## A01. Flipped price scales in liqudation reward calculation (Critical)

### Context

Asset price values are stored multiplied by a scale factor. For example, if the stored price value for a certain asset `A` is `1500` and the scale factor is `1000`, the real price is `1.5` dollars per base unit of `A`. Importantly, different assets may have different scale factors. The liquidation process may involve two different assets (the liquidated asset and the reward asset), so the computations may involve different scale factors.

### Issue

The liquidation reward calculation mistakenly swaps the two scale factors involved, scaling asset `A`'s price down by asset `B`'s scale factor and vice-versa. If these factors are different, then during this calculation one price increases and the other decreases, by orders of magnitude, compared to reality.

- If the liquidated asset price increases and the reward asset price decreases, the liquidator's reward decreases substantially.
- If the liquidated asset price decreases and the reward asset price increases, the liquidator's reward increases substantially. This case is exploitable. A malicious actor can take a very large loan and then self-liquidate[1], effectively recovering their collateral by spending orders of magnitude less than its real price.

### Recommendation

Use the appropriate scale factors.

### Status

Solved.

[1] Self-liquidation on demand is arguably non-trivial, but can be plausibly achieved by an experienced attacker.

## A02. Missing exchange logic in liquidation reward calculation (High)

### Context

During the liquidation process, the liquidator is rewarded with part of the liquidated user's collateral. These assets are effectively transferred to the liquidator's collateral balance for the same market. Only bank assets can act as collateral, so this reward must be calculated in bank assets as well.

### Issue

The code computes the reward amount in underlying assets, but does not convert this amount into bank assets. Because the bank-to-underlying exchange rate is generally greater than `1`, the liquidator receives more collateral than they're entitled to.

### Recommendation

Adjust the calculation, taking into account the bank-to-underlying exchange rate for the relevant market.

### Status

Solved.

## A03. Incorrect access of repaid amount in liquidation reward calculation (High)

### Context

Several operations require users to send assets to the protocol. Algos constitute an edge case in this regard because they can only be sent via `Payment` transactions. All other assets are sent via `AssetTransfer` transactions. The payment amount is given by the field `amount()` in the first case and the field `asset_amount()` in the latter case. This means that the code needs to treat Algos differently. The protocol generally handles this correctly, using wrappers which extract the amount from the appropriate field according to the expected asset type.

### Issue

The liquidation reward calculation does not use such a wrapper, but accesses the `asset_amount()` field of the liquidator's repay transaction directly. If this transaction was actually a `Payment` made in Algos, this field will return 0, convincing the protocol that it should not award anything to the liquidator, as it seems they haven't repaid any debt.

### Recommendation

Use a similar wrapper variable instead of accessing transaction fields directly. Notably, the existing wrapper is not the correct choice, because it computes the expected transaction type based on the current application. At the moment of reward calculation the current application is the reward market, while the repayment transaction should be checked against the repayment market.

### Status

Solved.

## A04. Incorrect check for expected transaction type in some operations (High) (Self-reported)

### Context

As mentioned in the previous issue, the protocol must account for the different transaction type expected in the case of the Algo market.

### Issue

The protocol checked that incoming transactions for `add_collateral` and `burn` operations in the Algo market are `Payment`s. But these operations actually require "bank" Algos, which can only be sent via `AssetTransfer` transactions. Thus, these two operations always failed on the Algo market.

This issue was reported by Algofi, shortly after providing Commit 1.

### Status

Solved.

## A05. Impact of rounding on interest calculation (Medium)

### Context

The protocol must account for the interest accrued when computing a user's borrowed amount at a given time. It does this by:

- Keeping a monotonically increasing `borrow_index` which accumulates interest rates over time. This is a global value which gets updated every few seconds on average.
- Updating a user's borrowed amount *lazily*. For each user `U`, the protocol tracks `user_borrowed_amount[U]` (the most recently computed value for the user's borrowed amount) and `user_previous_index[U]` (the value of the borrow index at the moment of the most recent recalculation). When user `U` interacts with their borrow position (either by borrowing more or repaying some debt), their amount is recalculated

as `user_borrowed_amount[U] = (user_borrowed_amount[U] * borrow_index) / user_previous_index[U]` and `user_previous_index[U]` is set to `borrow_index`[2].

Aditionally, the protocol tracks the `underlying_borrowed` quantity: the total borrowed amount in the market, including accrued interest. This variable is updated when a user interacts with their borrow position, but the interest is calculated globally, after every update of the `borrow_index`.

In the following, note that users can choose to borrow `0` amount. The only effect of such an operation is that the user's borrowed amount is recalculated. We say that a user `touches` their borrow position when performing this operation.

### Issue

Let's analyze what happens to the interest in two slightly different scenarios:

- In both scenarios user Alice borrows amount `X` at timestamp `0`. Consider two other timestamps `0 < T1 < T2`. Let `index(T)` be the value of `borrow_index` at timestamp `T`.
- In `Scenario 1` user Alice touches her borrow at time `T2` only. The resulting borrowed amount at time `T2` is $\frac{X * index(T2)}{index(0)}$.
- In `Scenario 2` user Alice touches her borrow at times `T1` and `T2`. The resulting borrowed amount is $\frac{\frac{X * index(T1)}{index(0)} * index(T2)}{index(T1)}$.

Mathematically, these two values are equal after cancelling terms. However, the protocol uses integer division, which truncates the result. Because of this, the second scenario will generally produce a lower interest value. We must entertain the possibility that some users will choose to touch their position very often in order to cancel a large portion of their interest via rounding errors. Moreover, because the interest for the `underlying_borrowed` quantity is computed independently, we can expect it to diverge from the actual sum of individual borrowed amount. This discrepancy may cause long-term problems.

This hypothesis was tested using the simplified model developed during the audit and an extreme version of the comparison above, in which `Scenario 2` involved hundreds of thousands of touch operations instead of two. It was concluded that the impact of this behavior on interest accrual might be significant and this fact was brought to the attention of Algofi.

### Status

Algofi acknowledged the issue and spent time to estimate its impact on the protocol. They concluded that the impact of users manipulating their interest calculation is non-zero, but not very significant. However, due to the added discomfort of having the total amount differ from the actual sum of borrows, they decided to change the borrow book-keeping model so as to mitigate these issues. This is the main change done in the second commit analyzed during the audit and will be mentioned in a further point.

[2] Several details are omitted for clarity, including the fact that `borrow_index` is scaled to mitigate rounding errors.

## A06. Missing check on user collateral cap during liquidation (Informative)

### Context

The protocol implemented a user collateral cap, which limited the amount of collateral an individual user can hold.

### Issue

Because liquidators are rewarded in collateral, the protocol *should* check that no liquidation can put the liquidator over the user collateral cap. This check was overlooked.

### Status

Algofi decided to eliminate the user collateral cap entirely.

# Commit 2

## B01. Incomplete checks for transaction group validity (Critical)

### Context [3]

Most operations on the protocol require relatively complex transaction groups, which are subject to a large number of security checks. To avoid duplication of effort, most of the checks are done during the evaluation of the first transaction in the group, even though they might target any transaction in the group. A relevant detail is that `ApplicationCalls` which aren't `NoOps` are handled very early, before any other checks (such as transaction group logic), and `CloseOuts` are approved by default.

### Issue

The checks made by the protocol during transaction group validation are generally very thorough and often offer redundancy. However, it seems that it would've been possible for some malicious users to circumvent some of these checks, in a couple of ways:

- Transactions which satisfy `on_completion() == CloseOut` are approved by the code before making any checks on the transaction group. This is largely mitigated by the fact that the first transaction in a group checks that all transactions are `NoOp`. However, the problem is that the first transaction itself might be `CloseOut`. A similar thing can happen with `ClearState`.
- Some checks might be skipped simply because transactions which should trigger them aren't sent to the manager, but to a different application. The protocol checks that all transactions which should be sent to the manager have the correct application id, but this check is *only* done on the market application. The protocol supports operations which don't interact with any markets (e.g claim rewards), so these might be vulnerable in this regard.

Successful exploits of this weakness could've resulted in attackers performing operations on arbitrary accounts on the protocol. Whether this exploit can go through successfully on the target commit has not been confirmed, but any hypothetical safeguards are likely coincidental and cannot compensate for the flagged design issue.

### Recommendation

- Check all relevant application ids in the manager as well. These checks can be included among the "first transaction checks".
- Check that the first transaction will surely be fully evaluated. This can be done by checking that it is a `NoOp` and its application id is correct. This check should be performed by *all* (group) transactions evaluated by the manager.

[3] This issue was *partially* discussed and addressed before analysis of Commit 2. We choose to include it here and omit the detailed timeline for clarity of exposition.

### Status

Solved. Algofi have implemented the above recommendations.

## B02. Incorrect check for non-zero denominator in issued borrow shares (Medium)

### Context

The upgraded borrow model assigns users "shares" of the total borrow of the protocol, similarly to how bank assets function as shares of the market supply. The following code fragment calculates the number of borrow shares issued when a new borrow is made[4].

```
calc_user_borrow_shares_to_issue =
    If(outstanding_borrow_shares > 0,
        to_borrow * outstanding_borrow_shares / underlying_borrowed + 1,
        to_borrow * 1000
      )
```

Here `outstanding_borrow_shares` counts the total number of active borrow shares in the market. If this value is `0`, the user is issued shares at a default rate of `1000` shares per borrowed unit. Otherwise, the user is issued shares at the current share rate and the result is always incremented to avoid unfavorable rounding.

### Issue

The above code assumes that `outstanding_borrow_shares > 0` implies `underlying_borrowed > 0` to avoid division by zero. Although intuitive, this implication does not hold. There exist degenerate cases in which some users still possess borrow shares but the total borrow is actually 0. A simple example:

- Alice borrows `1` unit. This is the first operation on the protocol, Alice is issued `1000` shares.
- Bob borrows `0` units. Bob is issued `1` share due to rounding.
- Alice repays `1`. Now we have `underlying_borrowed = 0`[5], but Bob maintains his share.

Of course, other similar scenarios are possible. Although they are arguably improbable, it is important to note that *should* they occur, the effect on the protocol is significant: any subsequent borrow operation would panic due to division by `0` and there would be no way to unblock the market without updating its code.

[4] The code is simplified for clarity.

[5] Technically Alice overpays here because she only owes `(1000/1001)` units. Although the protocol returns the difference in general, here it is too small and the full payment is accepted.

### Recommendation

Check explicitly that `underlying_borrowed` is non-zero. Consider rounding using the ceiling function instead of incrementing by default.

### Status

Solved.

## B03. Possible accelerated growth of bank asset exchange ratio (Informative)

### Context

Bank assets function as shares of the market supply. Exchanging bank assets for underlying assets and vice-versa requires division so the computation is subject to rounding.

### Finding

Due to the rounding involved in exchanging bank assets, some degenerate scenarios exist in which the bank-to-underlying exchange ratio grows much quicker than one would expect in a natural scenario. For example, consider a scenario in which the current bank-to-underlying exchange ratio is slightly above `1` and the number of circulating bank units is very small. A user can now try to mint a bank asset by spending `1` unit of underlying. The protocol would accept their payment, but because the ratio is greater than `1`, this would result in the user receiving `0` bank units. So this operation effectively increments the market supply, but keeps the number of circulating bank units constant. Repeating this on a massive scale at the start of the protocol lifecycle can significantly increase the exchange ratio. This is unpleasant and can cause long-term problems.

### Status

Algofi plans to seed all markets with a significant amount of bank asset units, ensuring a relatively high denominator for the exchange ratio at all times.

## B04. Possible deflation of borrow share value (Informative)

### Finding

Similarly to the previous issue, malicious actors could try to leverage the rounding done by the protocol and issue borrow shares without increasing the total borrow. This results in borrow shares losing value and effectively transferring part of the global borrow to accounts that choose to follow this strategy. Conceivably, this can be done on a massive scale with small enough increments so as not to require collateral deposits (due to rounding). Because markets will generally operate with relatively high amounts of shares, it is improbable that such attacks can have impact. Nevertheless, we think the point warrants attention.

### Status

Acknowledged. Additionally, Algofi have banned operations with `0` amounts and operations which would mint or issue `0` amounts.

## B05. Extraneous market supply cap check while adding collateral (Informative)

### Context

The protocol implements a market supply cap, which limits the amount of supply a market can hold at any given time.

### Finding

While processing the adding of new collateral by a user, the protocol counts the newly added collateral as supply and checks it against the market supply cap. It is incorrect to do so because a market's supply is not increased by adding collateral, but rather by minting the bank assets which are deposited as collateral. So the cap check should only be made during the mint operation (this is done correctly).

### Recommendation

Remove this check.

### Status

Solved.