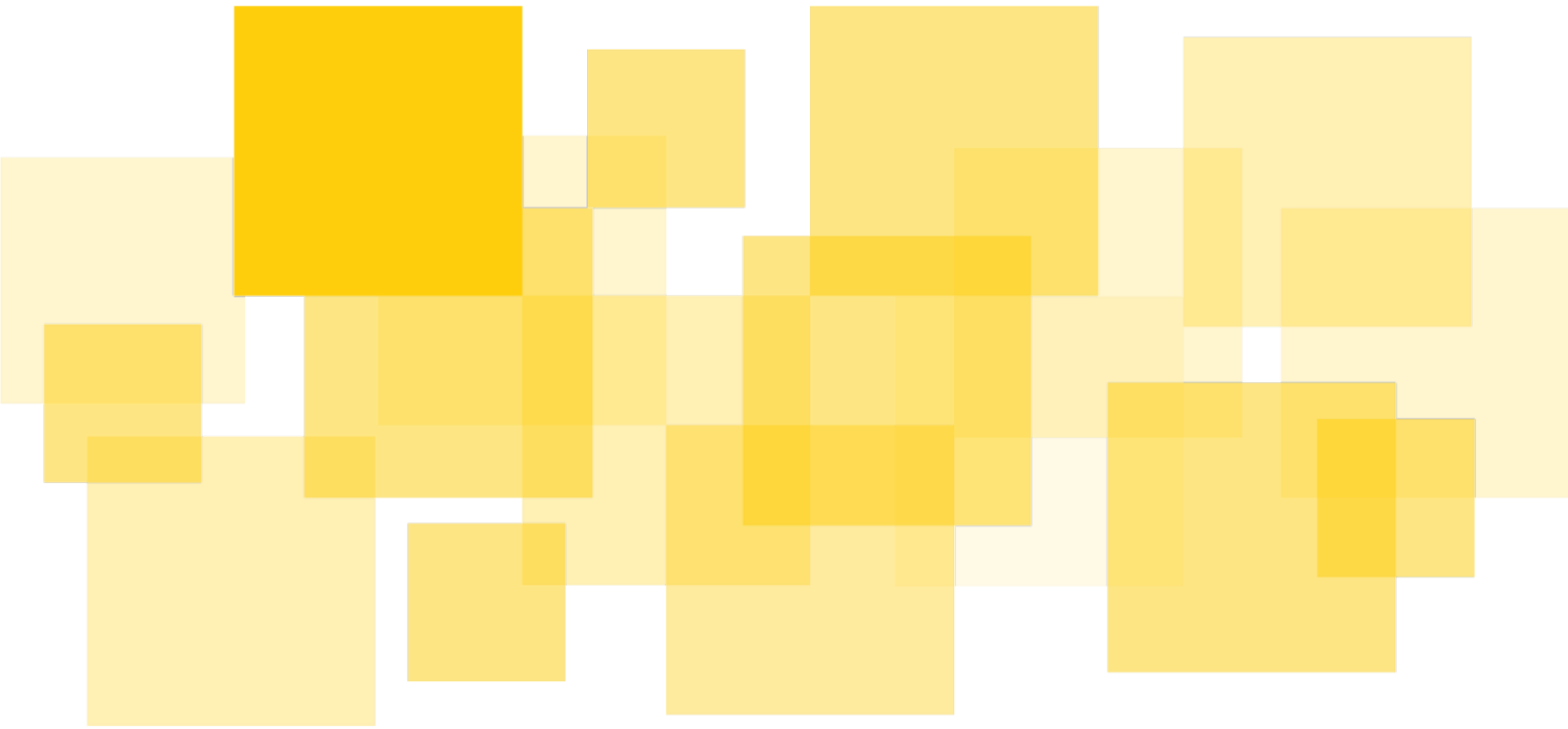
Security Audit Report

Algofi Lending V2

Delivered: 13 July 2022



Prepared for Algofi by

runtime verification

Contents

Summary . . . 2
Timeline . . . 3
Scope . . . 3
Disclaimer . . . 4
Methodology . . . 5
- Terminology . . . 6
- Properties and Invariants . . . 6

Findings . . . 9
- A1. Flash loans can be exploited to drain collateral . . . 9
- A2. Over repaying a loan can spawn ghost borrow shares . . . 9
- A3. Interest may be compounded when there is no borrow, blocking the protocol . . . 10
- A4. Liquidations can cause B-asset exchange rate to decrease . . . 10
- A5. Utilization ratio can grow beyond 100% . . . 11
- A6. Incorrect logic in final withdrawal . . . 12
- A7. Incorrect rewards calculation when compound interval is non-zero . . . 12

Informative Findings . . . 13
- B1. Obsolete foreign app in `Market.on_activate_market` . . . 13
- B2. Unnecessary check in `Market.on_user_opt_in` . . . 13
- B3. STBL Market and Vanilla Market have different borrow conditions . . . 13
- B4. Redundant check in `Manager.on_request_vault_transfer` . . . 14
- B5. Typos and suggestions in docs . . . 14

Summary

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

The objective was to review the contract's business logic and implementation in PyTeal and identify any issues that could potentially cause the system to malfunction or be exploited.

The audit has identified one critical issue A1. Flash loans can be exploited to drain collateral and one high severity issue A7. Incorrect rewards calculation when compound interval is non-zero , which could have been exploited with high probability. Both A1 and A7 were promptly patched by Algofi in the follow-up PRs #65, #66 and #67.

Other findings, namely A2, A3, A4, A5 and A6 are low-severity and are related to shallow liquidity situations and severe marked conditions. The Algofi team has made an effort to cover these edge-cases to improve the protocol's robustness.

We have also identified a number of informative findings, which do not present an attack scenario, but are still worth pointing out.

Timeline

The audit has been conducted over a period of 6 weeks, from May 30, 2022 to July 8, 2022.

Scope

The audit was conducted on the frozen commit 3895846381141555227faf8cb15de4367b3a5636, with the following files in scope:

Filename	Description
`contracts/manager.py`	Implements the Manager
`contracts/market.py`	Implements a Market for a particular asset
`contracts/stbl_market.py`	Implements the specialized Stabelecoin Market
`contracts/vault_market.py`	Implements the Vault
`contracts/wrapped_var.py`	Provides a unified interface for local and global state access
`contracts/globals.py`	Provides math and inner transaction issuing helper functions used across the codebase
`contracts/permissionless_sender_logic_sig.py`	Allows minimal sanity-checks for permissionless transactions

Additionally, we have reviewed the stub implementation of a price oracle (`contracts/test_oracle.py`), which we expect to be substituted with a real implementation upon deployment.

The `contracts/rewards_issuer.py` and `contracts/test_rewards_manager.py` remain out-of-scope due to audit priorities and time limitations.

Over the course of the audit, we have been also looking at test harness in `test/` and various parameters and off-chain helpers in `utils/contract_utils.py`, but we have not formally audited these files.

Disclaimer

This report does not constitute legal or investment advice. You understand and agree that this report relates to new and emerging technologies and that there are significant risks inherent in using such technologies that cannot be completely protected against. While this report has been prepared based on data and information that has been provided by you or is otherwise publicly available, there are likely additional unknown risks which otherwise exist. This report is also not comprehensive in scope, excluding a number of components critical to the correct operation of this system. This report is for informational purposes only and is provided on an ”as-is” basis and you acknowledge and agree that you are making use of this report and the information contained herein at your own risk. The preparers of this report make no representations or warranties of any kind, either express or implied, regarding the information in or the use of this report and shall not be liable to you or any third parties for any acts or omissions undertaken by you or any third parties based on the information contained herein.

Methodology

We have performed the audit in several stages:

Exploration At this initial phase, we have identified the priorities for the audit by browsing through the available documentation and the codebase, and by communicating with team at Algofi.

Manual Code Review We have employed a top-down approach, following the priorities identified at the Exploration phase.

First, we reviewed the Manager smart-contract in `contracts/manager.py`, paying special attention to the access control. Alongside the Manager, we reviewed the helper functions in `contracts/globals.py`, which perform extensive checks of transaction fields and are used consistently across the whole codebase. The `contracts/wrapped_var.py` was also reviewed at this stage, making sure that its interface coincides with our intuition.

Second, we worked through the vanilla Market in `contracts/market.py`, always referring back to the Manager to check that all interactions with the Market are checked by the Manager, when appropriate.

Finally, we went through the STBL and Vault market flavors to check that they are correctly specialized for their particular function.

Over the course of the review phase, we have identified a number of important Properties and Invariants that the contracts must maintain.

Modelling Being already familiar with the contract's inner workings, we have used our internal tooling to derive an executable model from the protocol's implementation in PyTeal. Then, we manually constructed a collection of Python wrapper modules to interact with the model, and consolidated these modules into a lightweight Python-based simulation environment.

Exploit Analysis and Validation During the Exploration, Review and Modelling phases, we have identified a number of potential exploits: access-control violations, potential rounding issues, arithmetic underflows and overflows, and invariant violations. At this stage, we employed the simulation environment constructed in the Modelling phase to produce execution traces and spreadsheet-style tables for exploit analysis.

At this stage we have confirmed a number of findings that we describe in detail further in the report.

Additional Code Review and Manual Reasoning In some cases, modelling and simulation provides too much information. To pin point the exact reason of some anomalies, we have employed manual pen-and-paper reasoning to check the suspicious scenarios.

Wrap-up Finally, we have performed one additional sweep over the codebase, reviewed a number of patches addressing the confirmed findings, and prepared the audit report.

Terminology

We introduce a number of terms that we use in the report. We assume that the reader of the report is familiar with the common blockchain terminology, as well as with the Algorand-specific concepts of rekeying, transactions, transaction groups, accounts, ASAs, and smart contracts. See Algorand Developer Docs for details on these concepts.

Protocol is the collection of smart contracts deployed on the Algorand blockchain that implements the Algofi Lending Market V2.

Asset is either Algos — the native currency of Algorand, or an Algorand Standard Asset (ASA).

Manager is the smart contract that controls the operation of the protocol. Most operations of the protocol require Manager's approval.

Market is the smart contract that implements the lending/borrowing functionality for a particular asset.

STBL Market is the smart contract that implements the STBL stablecoin

Vault is the smart contract that implements the Vault. A Vault allows locking Algos for participation in Algorand Governance, while keeping them liquid by counting them towards the user's total collateral in the protocol.

User is an Algorand account participating in the protocol. An **active user** is a user that is correctly opted into the Manager, and has an associated Storage Account.

User Storage Account is an Algorand account that stores the protocol state attached to a User: their participation in markets, balances, debt, and other data. Most of the times, we will refer to this account as "storage account", or just "storage". An **active storage** is a storage account that is opted into the Manager, rekeyed to the Manager, and has a user account associated with it.

Properties and Invariants

Over the course of the audit, we have identified a number of important invariants that the Protocol relies on. We did not formally prove that these invariants hold, but looking for their violations has been a major source of findings.

User-Storage mapping is bijective An active user corresponds to exactly one active storage, and an active storage corresponds to exactly one active user.

Active Storage is always owned by Manager A storage account must be rekeyed to the Manager upon opt-in, and remain to be owned by the Manager before it is closed out of the protocol.

Preamble is valid The preamble, if executed before a managed transaction, will satisfy the following properties:

- There are as many `"cup"` transactions as user market pages.
- If a user has several market pages, the first `"cup"` is responsible for page 0, the second for page 1, and the rest progress successively, with no gaps.
- Every market the user participates in is accounted for exactly once.
- The first foreign account (user storage) is the same for the whole preamble.
- The second argument (target market) is the same for the whole preamble
- The last preamble transaction's second argument (target market) is the same that the current market's `ApplicationID`.

These are checked by the preamble transaction's themselves, and
the `Market.User.verify_manager_transactions` function.

Borrows Share Circulation Validity In a market, the borrow share circulation always stays in sync with the borrowed amount. Borrow share circulation is 0 if there is no borrow, and vice-versa. More formally:

$$borrow\_shares = 0 \iff borrow = 0 \tag{1}$$

B-asset Circulation Validity In a market, the yield bearing asset's B circulation always stays in sync with the supply. B-asset circulation is 0 if there is supply is zero, and vice-versa. More formally:

$$circulation(B) = 0 \iff supply = 0 \tag{2}$$

B-asset Exchange Rate Must Never Decrease The yield beading asset B of a market accrues the lenders' interest, and its exchange rate to the market's underlying asset U must either stay the same or increase after every interaction with the protocol:

$$\forall t_1, t_2, \text{ such that } t_2 > t_1 : \frac{supply(U, t_2)}{circulation(B, t_2)} \geq \frac{supply(U, t_1)}{circulation(B, t_1)} \tag{3}$$

Utilization ratio ranges from 0% to 100% At any point in time, the protocol utilization ratio, i.e. the fraction of the borrowed to the supplied assets, must range from 0 to 1. More formally:

$$\forall t : 0 \leq \frac{borrowed_t}{supplied_t} \leq 1, \text{where } supply_t = cash_t + borrowed_t - reserves_t \tag{4}$$

Findings

A1. Flash loans can be exploited to drain collateral

Severity: Critical, Difficulty: Low, Probability: High

Description

The protocol lends assets to any opted-in account if the loan is taken and repaid within the same transaction group. The `on_flash_loan` function checks that the final transaction of the group repays the loan and covers the fee. However, the protocol does not identify the repayment with any specific transaction in the group, and opens an attack scenario.

Scenario

An attacker can form a transaction group that includes several consecutive requests for a flash loan of *the same amount*. The final transaction of this group will repay one of the loans, with the remaining loans remaining unrepaid, transferring the funds from the market to the attacker.

Recommendation

Enforce the flash loan call to be the first transaction of the group, thus disallowing several calls within the same group.

Status

The issue has been addressed by Algofi in a follow-up PR #65.

A2. Over repaying a loan can spawn ghost borrow shares

Severity: N/A, Difficulty: High, Probability: Low

Description

In `on_repay_borrow`, the over repay branch does not handle the final repay exactly at the limit, like the under repay does. If the user over repays the last borrow in the market, the `underlying_borrowed` becomes zero, while `borrow_share_circulation` may remain positive, breaking Invariant 1.

Scenario

See A3. Interest may be compounded when there is no borrow, blocking the protocol for a protocol denial-of-service induced by this issue.

Recommendation

Verify that the Invariant 1 is maintained upon repaying a borrow.

Note that the `recalculate_underlying_to_borrow_share` function checks that the borrow is non-zero, which is the right thing to do. This function is a perfect candidate for logging both the borrow share circulation and total borrow to facilitate runtime monitoring of Invariant 1

Status

The issue has been addressed by Algofi in a follow-up PR #68.

A3. Interest may be compounded when there is no borrow, blocking the protocol

Severity: High, Difficulty: High, Probability: Low

Description

A market's interest rate should only be compounded if there's non-zero borrow. The `compound_interest` subroutine relies on the Borrow Share Validity Invariant 1 to check this condition (market.py, line 760). The violation of the invariant causes a division-by-zero error in `new_implied_borrow_index` on every subsequent interest compound. Since `compound_interest` is executed before any lending function, the market will become non-functional.

Recommendation

The issue will be mitigated by implementing a fix for A2. Over repaying a loan can spawn ghost borrow shares .

As a defensive programming measure, we suggest also checking that `underlying_borrow` is non-zero before compounding interest (market.py, line 760).

Status

The issue has been addressed by Algofi in a follow-up PR #68.

A4. Liquidations can cause B-asset exchange rate to decrease

Severity: Low, Difficulty: High, Probability: Low

Description

Recall that, in a market, the B-asset B to underlying asset U exchange rate is calculated as:

$$\frac{supply(U)}{circulation(B)}, \text{where } supply(U) = cash(U) + borrow(U) - reserves(U)$$

When an unhealthy loan is liquidated, a portion of liquidation incentive that goes into the reserves, denominated in U, is rounded in the reserves' favor, while the liquidator's reward, which is denominated in B, is effectively rounded down. In other words, a tiny amount of B-tokens is burned at a rate higher than the actual exchange rate, violating Invariant 3.

Recommendation

We recommend taking a closer look at the upper bound of the amount of B tokens burned as liquidation fee. We suspect this amount to always be very low, but we have not carried out a formal upper bound proof. If the underlying tokens are very expensive and have few decimals, the liquidator may carry a significant loss.

Status

The issue does not need immediate addressing.

A5. Utilization ratio can grow beyond 100%

Severity: High, Difficulty: High, Probability: Low

Description

Recall that a market's utilization ratio is defined as:

$$UR = \frac{borrowed}{supplied}, \text{where } supply = cash + borrowed - reserves$$

If all cash is borrowed, the formula turns into:

$$UR = \frac{borrowed}{borrowed - reserves}$$

While *borrowed* grows with interest, the reserves grow with both interest and protocol fees. Notably, the `seize_collateral` action is a special case in which the reserves are payed a high fee, while borrow does not grow at all. In adverse market conditions, it is very likely that liquidations will start happening, and the reserves will be pumped with fees, causing the denominator to shrink, and UR to grow significantly beyond 100%. This, in turn, will cause the interest rate to skyrocket, making the market conditions even more adverse.

Furthermore, since the interest rate depends on the utilization ratio quadratically, it may at some point overflow, blocking the protocol completely.

Recommendation

At the very least, we suggest capping the utilization ratio at 100%, or any other reasonable upper-bound. Suggesting a different interest rate model for adverse market conditions is

beyond the scope of this audit, but we do recommend to consider more in-depth economic modelling to ensure the protocol remains robust in such scenarios.

Status

Addressed. Utilization ratio and interest rates caps added in PR #69.

A6. Incorrect logic in final withdrawal

Severity: Low, Difficulty: High, Probability: Low

The final withdrawal sends the remaining supply to the lender that burns the last B tokens in circulation, decrements the market's cash by the remaining supply.

$$supply = cash + borrowed - reserves$$

However, it's possible that the *supply* has a non-zero *borrowed* component at this stage. In such a case, the final withdrawal can fail (if there isn't sufficient cash at all) or dip into the reserves.

Description

A7. Incorrect rewards calculation when compound interval is non-zero

Severity: High, Difficulty: Low, Probability: High

Description

Rewards are issued to users in proportion to their borrow at any given time. However, because it is not possible to iterate over users in Algorand, it's necessary to implement necessary bookkeeping lazily: only update the user's reward index when the user actually interacts with the market.

This update is done in the `on_cycle` function, which is generally concerned with recomputing exchange rates, interest rates and global reward indices, and is triggered at intervals larger than `min_compound_interval`. While the necessary updates for the user must use fresh global reward indices, it is incorrect to only execute the user-level reward indices updates when `on_cycle` is triggered, because in this case it can happen that a user interacts with their borrow position without updating their rewards share (if the `on_cycle` function is still on cooldown and isn't triggered).

Scenario

A user interacts once with the market to make a small borrow. After a long period of no interaction, they make a very large borrow, but taking care to time it such that `on_cycle` is not triggered and their reward index is not recomputed. Then interact minimally with

the market taking care to trigger `on_cycle` this time and trick the protocol into believing that the user had been holding his current large borrow since their first interaction with the protocol, thus greatly increasing the amount of rewards allocated.

Status

Addressed in PR #66 and PR #67.

Informative Findings

B1. Obsolete foreign app in `Market.on_activate_market`

On line 456, `Global.current_application_id()` is supplied as a foreign app for an opt-in inner transaction into the Manager. The foreign apps array is not accessed in the Managers's handler `on_market_opt_in`, and the app id is received from `Global.caller_app_id()`.

B2. Unnecessary check in `Market.on_user_opt_in`

The `Manager.on_user_market_opt_in` makes `send_opt_in_txn` from Storage to Market, and supplies Storage as a foreign account. The Market handles this request in `Market.on_user_opt_in` and checks the following:

```
# verify sender is storage account
MagicAssert(Txn.sender() == Txn.accounts[1]),
```

I.e. that the sender (Storage) is the same that the first foreign account, which is set to be Storage by the manager. Since Market also checks that this transaction is set by Manager, this check will always pass.

It seems that it either can be removed, or instead the user's account can be passed as `Txn.accounts[1]` and the check should then assert that the Storage's address in user's local state matches the sender.

B3. STBL Market and Vanilla Market have different borrow conditions

On `Market.on_borrow`, the line 987 checks that

```
# verify sufficient cash on hand
MagicAssert(self.get_underlying_cash_on_hand() >= underlying_to_borrow),
```

Where `cash_on_hand == cash - reserves`.

The stablecoin market in `STBLMarket.on_borrow` checks on line 142:

```
# verify market has sufficient underlying to lend
MagicAssert(self.underlying_protocol_reserve.get() -
self.underlying_reserves.get() > underlying_to_borrow),
```

It may make sense to unify these conditions to `>=` for consistency.

B4. Redundant check in `Manager.on_request_vault_transfer`

On line 480 of `manager.py`, the check:

```
MagicAssert(Txn.sender() == self.market.address.value()),
```

does not check anything, because `self.market.address` is uniquely determined by `self.market_app_id_store`, which is set to be `Global.caller_app_id()`, turning the check into a tautology.

The security of this endpoint is only enforced by the check that the market is registered (line 482). If it was not for market registration check, an attacker could steal Vaults by crafting a smart contract that would call the manager, since `Txn.sender()` and `AppParam.address(market_app_id)` are always the same address, because `market_app_id` is `Global.caller_app_id()`.

Additionally, the comment on line 478 seems to be irrelevant.

B5. Typos and suggestions in docs

- In Manager, Set Active Market Registration the second item in "Checks" is spurious.
- Docs for Manager, Asset Optin don't mention the payment transaction to fund the storage with minimum balance.
- In Manager, Calculate User Position, the "Index" filed should be `Manager App ID`, not `Market App ID`.
- In Market, Add Underlying Collateral, must require the app call to market to include Manager as a foreign app for `sender_owns_account` check to work. Same applies to Add B Asset Collateral and Repay Borrow.
- Market, Claim Rewards is not documented.
- Market, Liquidate and Seize Collateral under-specifies the foreign arrays. The Liquidation transaction should make available the Seize Market's app ID and account, and the Seize transaction should make available the Liquidate Market app's ID and account, Seize Oracle app ID and account, and the Manager app.