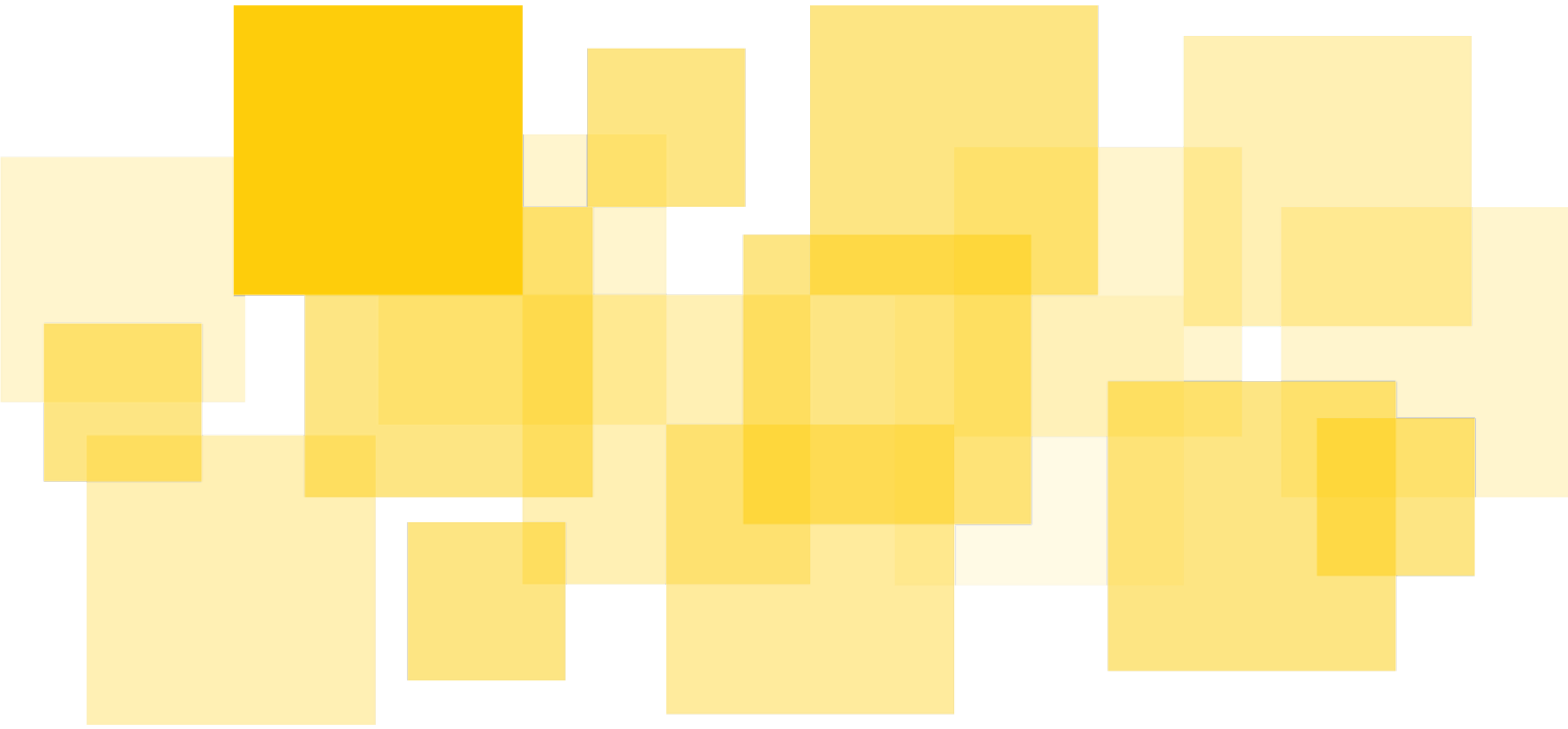
Security Audit Report

Tinyman AMM v2

Delivered: November 23, 2022



Prepared for Tinyman by

runtime verification

Contents

Summary

Tinyman engaged Runtime Verification to conduct a security audit of their Tinyman AMM v2 smart contracts implementing a decentralized exchange.

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

During the early stages of the audit one critical issue was self-reported by the team: A1. Flash loans and swaps can be exploited to drain asset reserves. The vulnerability was due to insufficient input validation in flash operations that could have enabled an attacker to borrow funds from a high-value pool and repay the same amount but to a low-value or a 0-value pool. The audit also identified several informative findings B1, B2 and B3. These findings do not enable any attack scenario but are still worth pointing out as they target improving the robustness of the implementation and the code's readability and maintainability.

All issues and suggestions raised by the audit, and highlighted in this report, were promptly fixed by Tinyman in subsequent commits (which were also reviewed as part of this audit).

Finally, the audit confirms a few highlights of the Tinyman AMM v2 contracts:

1. The Approval contract, which is the main contract implementing the AMM protocol, is immutable once deployed since the "delete application" and "update application" functions are explicitly disabled by the contract's logic.
2. The special "Manager" accounts can only manage protocol fees and withdraw donations. They do not have authority over anything else in the system.
3. When bootstrapping a pool, the Approval contract creates the pool's liquidity asset so that the asset is not in any way revocable or freezable, and cannot be centrally managed by any account.
4. There is no mechanism for any account (including the "Manager" accounts) to close out a pool or remove funds from a pool (beyond protocol fees and donations, which are claimable by a designated "Manager" account).

Goal

The goal of the audit was twofold:

1. Review the protocol's specifications document
2. Review the Tealish and TEAL implementations of the contracts to identify any discrepancies with the specification 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, readability and maintainability of the implementation.

Timeline and Scope

The audit has been conducted over a period of 5 weeks, from September 22, 2022, to October 31, 2022.

The audit was based on code hosted at Tinyman's private GitHub repository. The audit was initially conducted on the commit `b247dd94c3436386d3a770b5829b434411cbe519`. Some of the later fixes that were made during the audit were also reviewed as part of the audit. These fixes were included in the commits listed below:

1. Commit 2d01fcedb959124c63b92def2644022d67effb07
2. Commit 41f85674b4e91b23add30e6bf84a63530ac4223a
3. Commit 7cbf9b47cebda5339268cf870cdad174f38351df

The audit included in scope both the source artifacts in Tealish:

- The Approval program `contracts/amm_approval.tl` (source)
- The Clear-state program `contracts/amm_clear_state.tl` (source)
- The Pool program template `contracts/pool_template.tl` (source)

and the generated artifacts in TEAL:

- The Approval program `contracts/build/amm_approval.teal` (generated)
- The Clear-state program `contracts/build/amm_clear_state.teal` (generated)
- The Pool program template `contracts/build/pool_template.teal` (generated)

Over the course of the audit, we have also looked at testing scripts in `tests/` for general guidance, 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 that 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, reviewing other audit reports of similar systems (especially ones with flash loan capabilities), and communicating with the team at Tinyman.

Manual Code Review: Following the priorities identified during the Exploration phase, we have employed a top-down approach, focusing first on the interface through which users/contracts may interact with the main approval application, and working our way through the details of how the logic of the application is implemented, including its mathematical formulas. We highlight below the major stages of the manual code review process.

Authorization Model: We first reviewed the lifecycle of a pool, what operations on a pool the approval contract allows, and in which state, and who is authorized to initiate these operations, paying special attention to the consequences of rekeying the pool to the Approval contract.

Contract Interface: We then reviewed the checks the contract implements to ensure that interactions with the application and the pools are conformant to the defined interface. To make this review more systematic (and thus more comprehensive), we devised and followed two different approaches:

1. The first approach was manual, where we built a table for each transaction in the transaction group defining a slice of the interface to the application. The table lists all the relevant fields of the transaction, and for each field, identifies the semantic restrictions (requirements) imposed by the application's logic and whether these restrictions

are sufficiently checked by the implementation. We built such tables for all transaction groups that define the application's interface. This approach helped identify or confirm missing checks (see the Appendices for more details).

2. The second approach uses automatic model generation and verification using Alloy. We developed a formal specification of transaction groups and restrictions on their transaction fields as implemented by the contract, and then specified a number of correctness properties as logical formulas. We used this specification with Alloy to automatically generate all valid models (transaction groups) and then check them against the correctness properties. This method helped identify unusual or unexpected transaction group configurations that we then investigated (see the Appendices for more details).

Arithmetic Analysis: Next, we analyzed the implementation of the protocol's mathematical formulas (as specified in its documentation) as statements and arithmetic expressions in the contract. This included analyzing three fundamental aspects of arithmetic expressions in Tealish/TEAL:

1. Arithmetic overflow, underflow, and division-by-zero errors
2. Integer type conversion errors (byte arrays to unsigned 64-bit integers)
3. Precision and rounding-down effects (due to truncating integer division and square root operations)

Translation Validation: Finally, we reviewed the TEAL code generated by the Tealish transpiler (source-to-source compiler) to validate the translation from the Tealish contracts to the corresponding TEAL contracts. This was important to avoid having to trust the Tealish transpiler codebase. This translation validation stage was performed mostly manually, through the following steps:

1. Extract manually the name-to-slot map that the transpiler generated for the contract (with scoping info)
2. Using the inline comments, manually verify that the contract's statements were translated correctly
3. Verify completeness/correctness of the inline comments using a small Python script we wrote for this task

At this stage, we have confirmed a number of findings that we describe in detail further in the report. We classify findings using our own severity classification system.

Invariants Analysis: Over the course of the code review, we have identified a number of important properties and invariants that the contracts must maintain. This included two main classes of properties:

1. Function pre-conditions (assumptions that must be met for the function's correctness) and post-conditions (properties that the function guarantees)
2. Contract invariants (properties that should hold in the contract's state before and after the execution of a contract call)

We manually analyzed these properties against the implementation, using a paper-and-pencil style of reasoning where we produced informal but detailed arguments of why the implementation satisfies these properties (see the Appendices for more details).

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: the collection of smart contracts deployed on the Algorand blockchain that implements the Tinyman AMM v2.

Asset: either Algos — the native currency of Algorand (with asset identifier 0), or an Algorand Standard Asset (ASA – with a non-zero asset identifier).

Approval Contract: the smart contract that controls the operation of the protocol. It contains most of the logic of the protocol.

Pool Template: the smart signature template that can be instantiated to create new pools in the protocol. Upon bootstrapping, a pool created with this template is rekeyed to the approval contract account.

Issued Pool Tokens: the portion of the total supply of pool tokens that represents liquidity provided by poolers to the pool. It is exactly the sum of the *circulating pool tokens* (tokens belonging to users) and a small amount of tokens (1,000 tokens) that is locked in the pool.

Managers: Algorand accounts, distinct from the approval contract account and pool smart signature accounts, that are authorized for certain privileged operations in the protocol, namely fee management and claiming of protocol fees and donations.

Users: Algorand accounts (including contract accounts) that are distinct from managers, the approval contract account, and pool smart signature accounts. Users may initiate various non-privileged operations in the protocol.

Tealish: A domain-specific language for writing smart contracts in Algorand that "transpiles" into TEAL (see the Tealish GitHub repository for more information).

Properties and Invariants

Over the course of the audit, we have identified a number of important properties and invariants that the Protocol relies on. We did not formally prove that these invariants hold, but we gave informal arguments for why they indeed hold. Analyzing invariants has been a major source of findings.

Note: Unbound variables below refer to pool state variables.

Function Pre- and Post-conditions

update_price_oracle(): this function updates the cumulative price of an asset stored in the pool's state.

Pre-conditions: If the pool has issued tokens, the pool's reserves of both assets cannot be zero.

```
issued_pool_tokens > 0 -> asset_1_reserves > 0 && asset_2_reserves > 0
```

Post-conditions: The post-state cumulative asset price of an asset "i" is larger than or equal to its pre-state cumulative asset price.

```
if (issued_pool_tokens && time_delta)
    asset_i_cumulative_price' > asset_i_cumulative_price
else
    asset_i_cumulative_price' == asset_i_cumulative_price
```

check_pool_token_value(): this function checks that pool tokens do not decrease in value. This function is called just after adding or removing liquidity.

Pre-conditions: the initial and final values for issued_pool_tokens are non-zero.

Post-conditions:_ None.

check_invariant(): this function checks the pool's constant product invariant, which is that the product of the reserves cannot decrease. This function is called just after a swap or a flash swap operation.

Pre-conditions: the pooler fee amount for an asset is less than or equal to the asset's reserves.

Post-conditions:_ None.

calculate_fixed_input_fee_amounts(input_amount): This function calculates the `total_fee`, `poolers_fee` and `protocol_fee` of a fixed-input swap given the input amount.

Pre-conditions:

1. `protocol_fee_ratio > 0`
2. `total_fee >= protocol_fee`

Post-conditions:_

1. `total_fee == poolers_fee + protocol_fee`
2. If `input_amount > 0`, then `total_fee < input_amount`
3. If `input_amount < 10000 / total_fee_share`, then `total_fee == 0`

calculate_fixed_output_fee_amounts(swap_amount): This function calculates the `total_fee`, `poolers_fee` and `protocol_fee` of a fixed-output swap given the amount to be swapped. It has the same invariants, preconditions, and postcondition (1) as above

calculate_fixed_input_swap(input_supply, output_supply, swap_amount): This function calculates the `output_amount` of a fixed-input swap.

Pre-conditions: `swap_amount > 0`

Post-conditions: `output_amount < output_supply`

calculate_fixed_output_swap(input_supply, output_supply, output_amount): This function calculates the input `swap_amount` (without fees) for a fixed-output swap.

Pre-conditions: `output_amount < output_supply`

Post-conditions: `swap_amount > 0`

asset_amount = get_balance(N, asset_id): This function calculates the current available balance of a given asset in account N.

Pre-conditions: For Algos, `min_balance(N) <= balance(N)`

Post-conditions: None.

Contract Invariants

1. Protocol fee parameters never go beyond their prescribed ranges of values:
   - `1 <= total_fee_share <= 100`
   - `3 <= protocol_fee_ratio <= 10`
2. The cumulative price of an asset is non-decreasing:

   ```
   asset_i_cumulative_price' >= asset_i_cumulative_price
   ```

3. There are issued pool tokens if and only if the asset reserves are non-zero

   ```
   issued_pool_tokens == 0
     iff asset_1_reserves == 0
     iff asset_2_reserves == 0
   ```

4. The number of issued pool tokens is bounded above by the square root of the product of the asset reserves:

   ```
   issued_pool_tokens <= sqrt(asset_1_reserve * asset_2_reserve)
   ```

5. The circulating pool tokens are the issued tokens that are not locked in the pool:

```
if issued_pool_tokens > 0
    circulating_pool_tokens == issued_pool_tokens - LOCKED_POOL_TOKENS
else
    circulating_pool_tokens == 0
```

6. There must always be enough asset balance to cover the reserves and the protocol fees, unless the protocol is in the midst of a flash loan or a flash swap:

   Outside of a flash loan or a flash swap,

```
get_balance(asset_i_id) >= asset_i_reserves + asset_i_protocol_fees
```

7. There must always be enough pool token balance to cover what is yet to be issued:

```
if issued_pool_tokens == 0
    get_balance(pool_token_asset_id) == POOL_TOKEN_TOTAL_SUPPLY
else
    get_balance(pool_token_asset_id) >=
       POOL_TOKEN_TOTAL_SUPPLY - issued_pool_tokens + LOCKED_POOL_TOKENS
```

8. The minimum algo balance of the approval app contract must be maintained at all times.

Findings

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

A1. Flash loans and swaps can be exploited to drain asset reserves (self-reported)

[Severity: Critical, Difficulty: Low]

Description

A flash loan (and similarly a flash swap) is implemented using a pair of symmetric transactions: an opening transaction `flash_loan` that initiates the loan and transfers the loan amount to the user, and a closing transaction `verify_flash_loan` that checks that the loan has been properly closed. Both transactions need to specify the same pool account from which the loan is to be taken and then repaid with fees. However, this check that ensures that the transactions use the same pool account was missing, effectively enabling taking a loan from a high-value pool and then repaying the loan to a low-value pool. This also applies to flash swaps.

Scenario

An attacker could follow the following steps to mount an attack exploiting this vulnerability as follows:

1. Create arbitrary, 0-value assets A1 and A2
2. Create a pool for the asset pair A1-A2 and bootstrap it.
3. Create a flash loan (or flash swap) transaction group with:
   - an opening transaction requesting a loan of X goBTC tokens from the goBTC-Algo pool,
   - a payment transaction of X + Fee tokens of asset A1 to the A1-A2 pool,
   - a closing transaction verifying the repayment amount X + Fee tokens of asset A1.

Recommendation

(Self-made) Add statements asserting that the opening and closing transactions of a flash loan or a flash swap use the same pool account address.

In the `flash_loan` logic, add the following assertion:

```
assert(Gtxn[verify_flash_loan_txn_index].Accounts[1] == Txn.Accounts[1])
```

Similarly, in a the `flash_swap` logic, add the following assertion:

```
assert(Gtxn[verify_flash_swap_txn_index].Accounts[1] == Txn.Accounts[1])
```

Status

The issue has been addressed in 2d01fcedb959124c63b92def2644022d67effb07.

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.

B1. Tighten the lock check assertion

The approval application maintains a pool local state variable `lock` that, when set, disables any AMM operation on the pool except a `verify_flash_swap` operation until the lock is reset. While the lock is reset, all operations, except a `verify_flash_swap`, are enabled.

Currently, when executing an AMM operation, the application's logic checks the lock using the following assertion:

```
assert( (Txn.ApplicationArgs[0] == "verify_flash_swap")
        || (app_local_get(1, "lock") == 0) )
```

The assertion, however, allows for both disjuncts to be true, so that we may be executing a `verify_flash_swap` while the pool is unlocked, which is not intended. This cannot be exploited though, as the only way to successfully execute a `verify_flash_swap` is if we have had previously executed a `flash_swap`, which unconditionally locks the pool. Nevertheless, we recommend tightening this assertion for robustness, maintainability and readability of the protocol's implementation.

This has been address in commit 41f85674b4e91b23add30e6bf84a63530ac4223a, and the new assertion is:

```
assert( app_local_get(1, "lock") ==
        (Txn.ApplicationArgs[0] == "verify_flash_swap") )
```

B2. Fail earlier on invalid or unexpected inputs

It is generally advised to explicitly reject invalid or unexpected inputs early on during execution, even when such inputs do not immediately result in undesirable execution behaviors or enable any attack vectors. This has several advantages, including:

1. making the implementation more readable by explicitly documenting the reasons of failure,
2. preventing unknown unwanted behviors, and
3. making contract execution slightly more efficient at nodes by failing earlier.

There were a few cases in the approval contract that could be improved in this sense. Below we list those cases with brief explanations. All these suggestions were promptly addressed.

Reject flash swap requests with output amounts greater than the reserve: Such requests should not be possible, and currently fail later in execution (in the `verify_flash_swap` transaction) due to underflow when attempting to calculate the difference `asset_i_reserves - asset_i_output_amount` . The recommendation is to fail in this case much earlier (in the `flash_swap` transaction) by adding the following assertions:

```
assert(asset_1_output_amount <= asset_1_reserves)
```

and

```
assert(asset_2_output_amount <= asset_2_reserves)
```

to the `flash_swap` logic just before transferring the assets to the user (similar to the existing checks in the `flash_loan` logic).

The issue has been addressed in 2d01fcedb959124c63b92def2644022d67effb07.

Reject swaps with zero input or output amounts: Flash swaps where the input or output amounts are zero are meaningless and should be rejected. Currently, such swaps fail indirectly at later points in the swap logic of the contract. We recommend explicitly rejecting

them earlier in the logic using assertion statemetns of the form: `assert(input_amount)` and `assert(output_amount)`.

The issue has been addressed in 41f85674b4e91b23add30e6bf84a63530ac4223a.

Reject requests to remove zero liquidity: It makes no sense to attempt to burn zero pool tokens, and thus this should be rejected. Such an attempt currently fails later in the logic as the calculated asset amounts are zero in this case. We recommend adding an explicit check earlier that the `removed_pool_token_amount` is non-zero.

The issue has been addressed in 41f85674b4e91b23add30e6bf84a63530ac4223a.

Fail earlier if the `index_diff` is too small: `index_diff` is the index difference between the symmetric pair of opening and closing flash loan transactions or the symmetric pair of flash swap transactions. In a flash loan, this difference cannot be less than less than two if one asset was requested, or less than three if two assets were requested, since the repayment transactions must appear in between the loan and verify transactions. Similarly, for a flash swap, the index difference needs to be at least two to allow for at least one transaction for the repayment. An index difference that does not satisfy these requirements currently causes the logic to fail due to unexpected or non-existent transactions. We recommend adding explicit checks that the these requirements are satisfied earlier in the flash loan and flash swap logic.

The issue has been addressed in 41f85674b4e91b23add30e6bf84a63530ac4223a.

B3. Typos and other suggestions

- In the main Tinyman protocol description document, Example 1 of swap fees: "If the input amount is 20000A, fee is 20000A*fee rate AssetA. If the fee rate is 0.3%, total fee is 60 AssetA and protocol fee is 60 AssetA". The latter 60 should be 10.
- the comment in line 1061 (in check_invariant) uses strict inequality `Initial K < Final K without fees` while the assertion in the following line uses `b<=`, which is correct, and thus the comment needs to be fixed.
- Refactor the fee calculation logic in add_liquidity so that the function

  ```
  calculate_fixed_output_fee_amounts(...)
  ```

  is reused. The logic computes the fee of a fixed-output swap, which is already implemented by this function. This would improve maintainability and readability of the code.

Appendices

Transaction Field Tables

To systematically investigate sufficiency of input validation in the Approval contrcact, we built for each transaction that may appear in a transaction group in the contract's interface a table that lists: - the field's name, - the restrictions that need to be checked by the contract, and - whether these checks are currently made directly or indirectly, along with some notes.

There were a total of around 45 tables. All the tables can be found here. Below is a sample table.

Gtxn[N+X]: Verify Flash Swap AppCall from User

Gtxn[N]: AppCallTx	Flash Swap AppCall from User		
Field	**Restrictions**	**Checked?**	**Notes**
Fee	None	No	This is up to the sender (which cannot be a bootstrapped pool)
Sender	User address, which must be the same as 1. the receiver of itxn[0] and itxn[1] (as applicable) 2. the sender of Gtxn[N+X]	Yes	Cannot be the pool's address by design
TxType	AppCall => must also be checked by Gtxn[N+X]	Yes	
RekeyTo	None	No	
Application ID	Approval App ID, which must be the same as the app ID of Gtxn[N+X]	Yes	
OnComplete	NoOp => must also be checked by Gtxn[N+X]	Yes	
Accounts	1: Pool address, which must be a. the address of a valid bootstrapped pool b. the same as the sender address of itxn[0] and itxn[1] (as applicable) c. the same as the Gtxn[N+X].Accounts[1]	(a) Yes (Implicitly) (b) Yes (d) ~~No~~ Yes	This issue was self-reported, and has been fixed in the latest commit. (c) is now checked.
App Arguments	0: "flash_swap" => must also be checked by Gtxn[N+X] 1: index_diff => must be: a. greater than 1 b. equal to index_diff of Gtxn[N+X] 2: asset_1_amount 3: asset_2_amount a. the sum of these two amounts must be non-zero b. can at most be equal to the reserves of the corresponding asset	0: Yes 1: (a) No (b) Yes 2 & 3: (a) Yes (b) Yes (implicitly)	- when index_diff == 0, txn fails since the first app arg is different in both txns - when index_diff == 1, txn fails since the pool invariant won't hold Recommendation: check that the index_diff > 1
Foreign Apps	None	No	
Foreign Assets	0: asset_1_id 1: asset_2_id - The asset IDs must match the ones given in the pool contract	Yes (implicitly)	Inner txns would fail if these ids don't match the pool ids

Alloy-based Modeling and Analysis

In this section, we give a general overview (with examples) of the formal specification and analysis of transaction fields we developed for this audit using Alloy.

The full specification and supporting documents can be found here.

Structural Alloy specification

Tinyman AMM v2 protocol is very flexible in terms of imposed restrictions on a transaction group structure. While this design simplifies integration with other protocols, it complicates

safety analysis because of the huge number of possible configurations of valid transaction groups.

We used Alloy to tackle this problem. The Alloy specification language was specifically designed to perform structural analysis in various domains. The Alloy analyzer enumerates and checks, via SAT solvers, all possible structures up to specified bounds. For instance, we may ask the Alloy analyzer to check some property for up to 16 transactions in a transaction group, up to 16 accounts, 16 assets, 16 pools, etc, and the analyzer will essentially exhaustively search the reachable configuration space for violations of the specificed transaction field restrictions. This analysis gives us more confidence that the property is true for all possible configurations.

The specification was designed to model most important abstract qualitative structural properties of the protocol. Using this specification we were able to do several things, including:

1. manually reviewing possible transaction groups under certain conditions, and
2. automatically checking that certain relevant properties hold in all possible configurations up to some specified bounds.

A Brief Overview of the Model

We highlight below some important modeled items:

Assets:

Assets are modeled via ordered set `Asset` with one selected element `Algo`. Algo is always constrained to be the first element.

```
sig Asset{}
one sig Algo in Asset {} {Algo = oa/first}
```

Accounts:

Base signature for accounts:

```
sig Account{}
```

`Application` is some selected atom in `Account`:

```
one sig Application in Account {
```

with next relations (fields):

- `fee_setter` - points to one account that corresponds to a fee setter account:

```
fee_setter: one Account,
```

- `fee_collector` - self-describing:

```
fee_collector: one Account,
```

- `fee-manager` - self-describing:

```
  fee_manager: one Account
}
```

Pools:

Pools are selected set from `Account` set:

```
sig Pool in Account {
```

Assets of a pool:

```
  asset1 : one Asset,
  asset2 : one Asset,
  pool_token: one Asset,
```

Pool state regarding issued tokens:

```
  tokens_issued: Transaction -> one TokensIssuedState,
```

Lock state for each transaction:

```
  lock: Transaction -> one LockState
} {
```

The state of the lock and tokens may change in some transaction in transaction group, so locks and property of issued tokens are modeled via relations between `Transaction` and corresponding enumeration.

Transactions:

A transaction may be one of two types: transfer and application call:

```
enum TransactionType {
  Transfer,
  AppCall
}

sig Transaction {
```

Main fields of a `Transaction`:

```
  type: one TransactionType,
  sender: one Account,
  receiver: lone Account,
  asset: lone Asset,
  op: lone Operation,
```

Operations:

Operations correspond 1-to-1 to application calls.

Next list is the list of modeled operations:

```
enum Operation {
  ...
-- amm
  OpAddInitialLiquidity,
  OpAddLiquidity,
  OpRemoveLiquidity,
  OpSwap,
  OpFlashLoan,
  OpVerifyFlashLoan,
  OpFlashSwap,
  OpVerifyFlashSwap
}
```

Sample Properties

We highlight some important properties that were checked using the specification.

- `Verify flash swap` is performed always under the lock:

```
verify_flash_swap_always_under_lock:
check {
  all vfs: all_transactions_for[OpVerifyFlashSwap]
  | AmmParams.pool_address[vfs].lock[vfs] = Locked
} for 16 but 6 int

--   No counterexample found. Assertion may be valid. 481047ms.
```

- `Verify flash swap` is the only possible operation for a locked pool:

```
only_verify_flash_swap_is_possible_for_locked_pool:
check {
  all t: amm_transactions
  | AmmParams.pool_address[t].lock[t] = Locked implies t.op = OpVerifyFlashSwap
} for 16 but 6 int

-- No counterexample found. Assertion may be valid. 177ms.
```

- `Flash loan` and `Verify flash loan` are always symmetrical and the both refer to the same pool and user address:

```
-- check flash loan configurations

flash_loan1:
check {
```

```
  OpFlashLoan not in Transaction.op
  implies OpVerifyFlashLoan not in Transaction.op
}
      for 16 but 6 int
-- No counterexample found. Assertion may be valid. 686ms.

flash_loan2:
check {
  OpVerifyFlashLoan not in Transaction.op
  implies OpFlashLoan not in Transaction.op
}
      for 16 but 6 int
-- No counterexample found. Assertion may be valid. 605ms.

flash_loan3:
check {
  all fl: all_transactions_for[OpFlashLoan] {
    let vfl = FlashLoanParams.verify_txn_index[fl] {
      MainParams.user_address[fl] = MainParams.user_address[vfl]
    }
  }
} for 16 but 6 int
-- No counterexample found. Assertion may be valid. 2665ms.

flash_loan4:
check {
  all fl: all_transactions_for[OpFlashLoan] {
    let vfl = FlashLoanParams.verify_txn_index[fl] {
      AmmParams.pool_address[fl] = AmmParams.pool_address[vfl]
    }
  }
} for 16 but 6 int
-- No counterexample found. Assertion may be valid. 2232ms.
```

- `Flash swap` and `Verify flash swap` always referto the same pool and user address:

```
-- check flash loan configurations
flash_swap1:
check {
  all fs: all_transactions_for[OpFlashSwap] {
    let vfs = FlashSwapParams.verify_txn_index[fs] {
      MainParams.user_address[fs] = MainParams.user_address[vfs]
    }
  }
} for 16 but 6 int
-- No counterexample found. Assertion may be valid. 4546ms.
```

```
flash_swap2:
check {
  all fs: all_transactions_for[OpFlashSwap] {
    let vfs = FlashSwapParams.verify_txn_index[fs] {
      AmmParams.pool_address[fs] = AmmParams.pool_address[vfs]
    }
  }
} for 16 but 6 int
-- No counterexample found. Assertion may be valid. 3849ms.
```

Analysis of Invariants

We identified variaous properties that characterize the correctnss of the protocol, including protocol invariants and function pre- and post-conditions. Through manual analysis of the code, we developed informal arguments detailing how these properties may be formally (and possibly mechanically) verified, although this formal verification task was not attempted due to time constraints. The informal discussion of why these properties hold in the contracts can be found here.