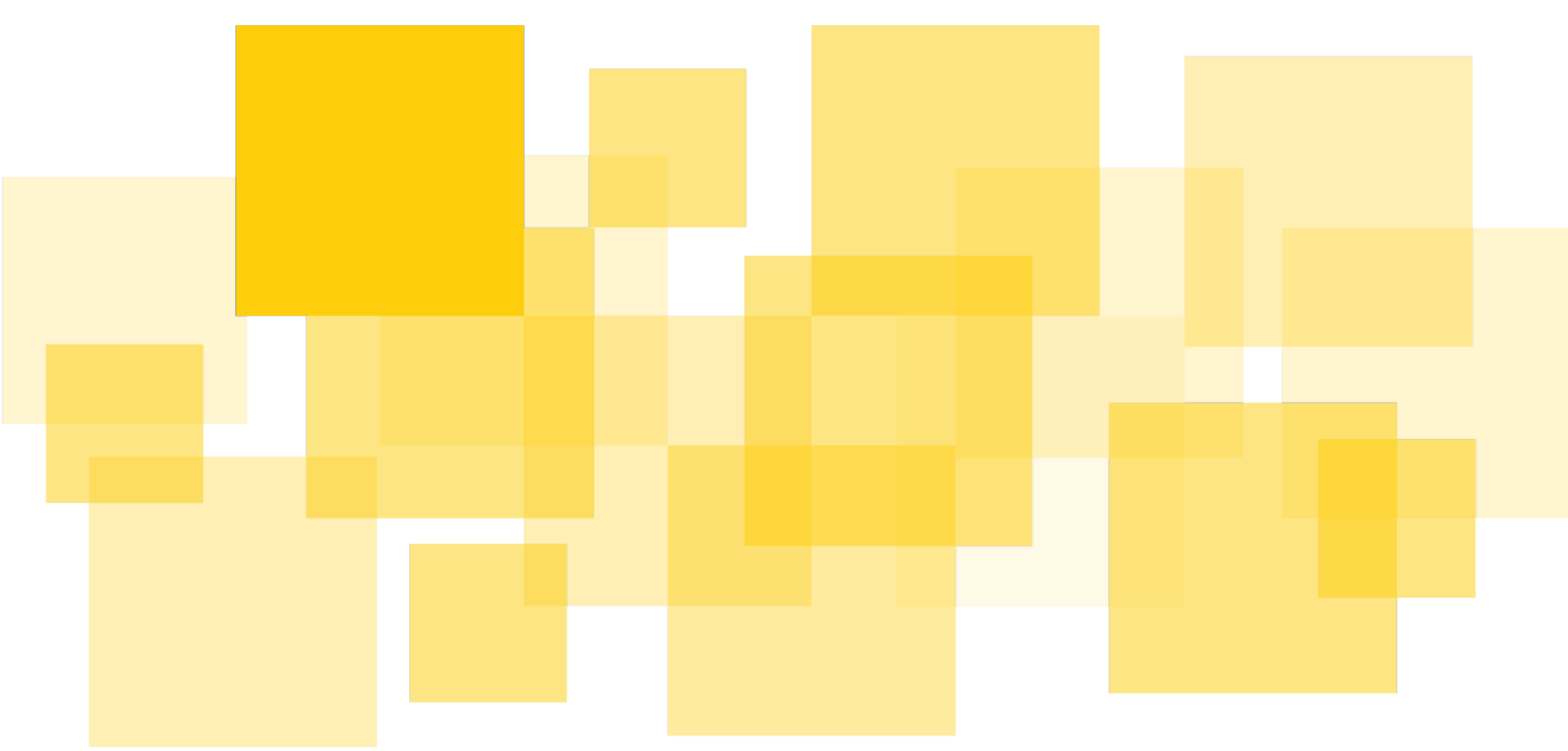


Security Audit Report

Tinyman Smart Contracts

Delivered: August 4th, 2021



Prepared for Tinyman by



Table of Contents

Executive Summary

Goal

Scope

Methodology

Disclaimer

Findings

A01: Bootstrapping Broken Pools through Ill-formed Transaction Groups

A02: Leaking Funds When Burning LP Tokens Due to Miscalculated Withdraw Amount

A03: Draining of Pool Funds Due to Missing GroupSize Checks

A04: Losing Unclaimed Excess LP Token Amounts When Minting (and Locking Them Up Indefinitely)

A05: Vulnerability to LP Token Deflation Attack

A06: Miscalculated Protocol Fee Due to a Rounding Error

Informative Findings

B01: Draining/Locking Up of Funds in Pools Involving Untrusted Assets

B02: Alternative Protocol Fee Calculation Method

B03: Including Arbitrary Character Sequences in Liquidity Pool Asset Name

B04: Using Tinyman as a Price Oracle

B05: Code Readability/Maintainability Improvement Opportunities

B06: Code Size Optimization Opportunities

Appendix

Executive Summary

Tinyman engaged [Runtime Verification Inc](#) to conduct a security audit of their smart contracts. The objective was to review the contracts' business logic and implementations in TEAL and identify any issues that could potentially cause the system to malfunction or be exploited .

The audit was conducted by Musab Alturki in two separate rounds over the course of 8 calendar weeks:


1. Round 1 (6 weeks, over a period spanning April 23, 2021 through June 25, 2021) considered an early version of the contracts (commit ID `cdfcc1617f05c904a998ef9add9fda60f78bc5db` -- private repo), which was the latest version available at the time of starting this audit engagement.
2. Round 2 (2 weeks, July 21, 2021 through August 4th, 2021) focused on analyzing the security implications of the changes/additions the development team made to the contracts (commit ID [3ff4162a38856dae70f8e6ce0b05386a403a1fc3](#)) based on the outcome of the first round of the audit.

The audit led to identifying six potentially critical issues, in addition to several informative findings and general recommendations. The critical issues identified included: Bootstrapping ill-formed pools (A01), leaking of funds when withdrawing liquidity (A02), draining of pool funds (A03), locking up of liquidity tokens (A04), liquidity token deflation attack (A05) and miscalculated protocol fees (A06). Informative findings (B01 - B06) highlighted various aspects of the contracts that can be made more robust and more maintainable and readable. They also included general recommendations to improve code size and optimize performance.

All the critical issues have been addressed, and most of the informative findings and general recommendations have been incorporated as well. The code is amply documented and thoughtfully designed, following best practices.

Finally, a few highlights of the Tinyman contracts:

1. The validator application (whose logic is given in `validator_approval.teal` and `validator_clear_state.teal`) is immutable once deployed, as the application's "update" and "delete" operations are disallowed by the contract's logic (regardless of the sender).

- 
2. The pool's logic (given in `pool_logicsig.tmpl.teal`) ensures that the liquidity asset of any pool created through the protocol's bootstrapping process is not revocable or freezable, and is not centrally managed by any account.
 3. There is no mechanism for any account (including the validator application Creator account) to close out a pool or remove funds from a pool.

Goal

The goal of the audit is twofold:

1. Review the high-level business logic (protocol design) of the Tinyman system based on the provided documentation
2. Review the low-level implementation of the system given as smart contracts in TEAL

The audit focuses 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 highlights informative findings that could be used to improve performance and efficiency of the implementation and optimize its code size.

Scope

The scope of the audit is limited to the following artifacts:

1. The system's high-level documentation given by:
 - a. The Tinyman AMM Design document and the FAQs document (available [online](#))
 - b. The [readme.md](#) file (and linked documents) located in the [docs/ folder](#) of the system's github repository documenting protocol messages and their formats
2. The system's implementation as smart contracts in TEAL (located in the [contracts folder](#)) comprising the following contracts:
 - a. `pool_logicsig.teal.tmpl`
 - b. `validator_approval.teal` (in addition to `validator_clear_state.teal`)

The audit is limited in scope to the artifacts listed above. Off-chain and client-side portions of the codebase as well as deployment and upgrade scripts are not in the scope of this engagement.

Methodology

Although the manual code review cannot guarantee to find all possible security vulnerabilities as mentioned in our [Disclaimer](#) below, we have followed the following approaches to make our audit as thorough as possible.

First, we rigorously reasoned about the business logic of the contract, validating security-critical properties to ensure the absence of loopholes in the business logic. As the system is based on Uniswap's AMM design, we carefully examined previous (publicly available) audit reports of Uniswap v1 and v2 to review the list of known issues and check whether they apply to Tinyman, and if they apply, we check if the code is vulnerable to them.

Second, we thoroughly reviewed the contract source code to detect any unexpected (and possibly exploitable) behaviors. As TEAL is a relatively low-level language, we constructed different higher-level representations of the TEAL codebase, including:

1. automatically generated control-flow graphs using an extended version of the [Tealer Static Analyzer](#) tool,
2. manually writing corresponding python-like pseudo code (based on the generated CFGs above), and then
3. cross-checking it with the high-level description of the system.

This approach enabled us to systematically check consistency between the logic and the low-level TEAL implementation.

Thirdly, we reviewed the TEAL guidelines published by Algorand to check for known issues.

Additionally, given the nascent TEAL development and auditing community, we reviewed [this list](#) of known Ethereum security vulnerabilities and attack vectors and checked whether they apply to TEAL smart contracts; if they apply, we checked whether the code is vulnerable to them.

Finally, we dedicated the last two weeks of the audit to review the security impact of the changes made to the contracts' code based on the issues raised in the first phase to ensure they were properly addressed and investigate effects on other parts of the contracts.



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 only, 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 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.

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: Bootstrapping Broken Pools through Ill-formed Transaction Groups

The `pool_logicsig` contract computes a value and pushes it to the top of the stack (lines 280 -- 286) but that value is never used. The value corresponds to the result of this Boolean expression:

```
global GroupSize == 5 && gtxn 4 Sender == txn Sender
```

which checks that:

1. the transaction group contains exactly 5 transactions, and that
2. the sender of the fifth transaction in the group is the Pool logicsig account.

Not using this result means that the logic effectively skips making these checks, which would allow ill-formed transaction groups when bootstrapping a pool (of two non-Algo ASAs). The transaction group may now be such that:

1. the fifth transaction is no longer a proper asset opt-in transaction, and/or
2. the group includes additional (malicious) transactions that are not checked by the logic.

An attacker may use this vulnerability to cause damage by breaking the protocol or at least creating a non-functioning pool.

Scenarios

1. The attacker sends a bootstrap transaction group with the fifth (opt-in) transaction having a sender different from the Pool account, rendering the pool unusable (as it's unable to receive Asset2 tokens)
2. The attacker sends a bootstrap transaction group that includes the required five transactions in addition to a sixth transaction, which can be:

- a. a pay transaction that causes the pool to pay back to the attacker most of the amount of Algo that the attacker just paid to the pool in transaction 0 of the group, or
- b. a pay transaction that drains the pool account of all Algos the account may have had just before bootstrapping
- c. an app call transaction that could possibly cause the validator application to get into an inconsistent state

Recommendation

Make sure the values computed by lines (280 -- 286) is used by inserting either an assert instruction right after line 286, or an && instruction right after line 289.

Status

This issue has been fully addressed.

A02: Leaking Funds When Burning LP Tokens Due to Miscalculated Withdraw Amount

On a burn operation, the wrong amount of Asset2 may be sent out to a pooler due to a calculation error. More specifically, line 394 loads the content of scratch memory slot 61:

```
Load 61 // outstanding_asset2_amount
```

which stores the outstanding amount of asset 1 as maintained in the state of the application, while what should be loaded is the outstanding amount of asset 2 (as the comment in the same line above indicates), which is stored in slot 62. This bug may result in inflating the computed value of the supply amount of Asset2, which inflates the amount of Asset 2 that is allocated for withdrawal to the pooler. This in turn affects the value of the issued pool shares and results in incorrect price adjustments.

Scenario

The attacker can exploit this miscalculation to steal Asset2 tokens by observing the current state of the application and repeatedly minting and burning (perhaps the same amount of) LP tokens. The attack will have a higher chance of going through undetected by performing deposits and withdrawals of small amounts at a relatively low frequency

but over an extended period of time, allowing swaps in between to “correct” the price before timing the next burn operation.

Recommendation

Fix the statement in line 394 so that the correct slot is used:

```
Load 62 // outstanding_asset2_amount
```

Status

This issue has been fully addressed.

A03: Draining of Pool Funds Due to Missing GroupSize Checks

The pool_logicsig and validator_approval contracts do not check the global GroupSize property of the transaction group being executed along every possible execution path. In particular, the mint, burn, swap, redeem and redeem fees operations do not include necessary logic to ensure that the size of the transaction group being processed is equal to the expected size. For example, the mint operation expects a transaction group consisting of exactly 5 transactions, but this is not enforced. This can potentially be exploited by an attacker to include an additional pool-signed transaction to an otherwise valid transaction group that would drain the pool account of its Algos and/or assets.

Scenario

An attacker could target a pool with some sizable holding amount of Algos (or assets) by constructing a proper transaction group of any of the affected operations listed above (say a swap) and including additional one or more transactions signed by the pool’s logic signature. These transactions could be pay transactions (for transferring Algos) and/or asset transfer transactions (for transferring assets), effectively draining the funds of the pool account.

Recommendation

Add a GroupSize check that asserts the exact expected size of the transaction group to every execution path corresponding to the affected operations: minting, burning,

swapping, redeeming and redeeming of fees. A natural location for these checks is the pool_logicsig contract.

Status

This issue has been fully addressed.

A04: Losing Unclaimed Excess LP Token Amounts When Minting (and Locking Them Up Indefinitely)

While discussing the logic that updates the application state in the contract (the branch labeled “update_local_state”), Tinyman developer Fergal pointed out that the application state update fragment in lines 844-850 that is supposed to accumulate excess liquidity amounts belonging to the pooler incorrectly overrides the current value, instead of adding to it. This causes the pooler to lose any unclaimed excess amount when performing a mint operation before claiming the current excess amount. Furthermore, as this amount is elsewhere recorded as outstanding liquidity, there would be no way for this amount to be burned, resulting in having it locked up forever.

Scenario

This bug causes the system to lock up pooler’s LP tokens every time a pooler mints while its excess amount of the LP token is non-zero. As more poolers mint before claiming excess amounts, more LP tokens get forever locked up, eventually leading to the pool being non-functional. Note that an attacker may be able to expedite this process, but the achievable speed up is bounded by how much the attacker is able to spend on locked up tokens.

Recommendation

Fix the logic in lines 844-850 so that the state variable maintaining the excess LP token amount in the state of a pooler account properly accumulates excess LP token amounts.

Status

This issue has been fully addressed.

A05: Vulnerability to LP Token Deflation Attack

In the first mint operation, the contract needs to check that the geometric mean of Asset1 and Asset2 amounts is bounded below by the minted LP token amount, which can be expressed as follows:

```
minted_squared <= amounts_product_squared
```

Since the amounts involved require wide (128-bit) multiplication, the contract was supposed to implement this assertion by checking that:

```
if (minted_squared_hi >= amounts_product_squared_hi) then
    minted_squared_hi == amounts_product_squared_hi &&
    minted_squared_lo <= amounts_product_squared_lo
```

The implementation in the contract (lines 519 -- 538) was found to incorrectly enforce the inverse of this check when the products “minted_squared” and “amounts_product_squared” differ only in the low 64 bits, that is, when

```
minted_squared_hi == amounts_product_squared_hi
```

Specifically, line 536 in the contract used the wrong comparison operator to assert this property.

This bug enables an attacker to severely deflate the value of an LP token to a point that would render the pool quickly unusable.

Scenario

The attacker may start with a first mint in which $2^{32} - 1$ LP tokens are minted to represent 1 token of Asset1 and 1 token of Asset2. Only a few subsequent minting operations (which would necessarily be using very large amounts of LP tokens) would then be possible before the minting logic would begin to panic, denying any further mint operations very early in the lifetime of the pool.

Recommendation

Either replace the comparison operator in line 536 with \geq , or interchange lines 534 and 535 to achieve the same effect.

Status

This issue has been fully addressed.

A06: Miscalculated Protocol Fee Due to a Rounding Error

The formula for calculating the protocol fee as specified when performing a swap operation might potentially produce the value 0 more often than necessary, due to (integer) division being performed before multiplication (at the same precedence level):

```
((asset_in_amount * 5) / 20000) * issued_liquidity_tokens) / input_supply
```

Any `asset_in_amount < 4,000` would immediately result in a calculated fee of 0 (regardless of the other parameters). This is in addition to the (inevitable) case when:

```
asset_in_amount < 4000 * input_supply / issued_liquidity_tokens
```

Therefore, this formulation could potentially lead to significant losses in protocol fees over a large number of small-to-medium-sized swaps.

Scenario

No interesting attacker-specific scenarios other than intentionally using swaps with small amounts to deprive the protocol of its fees.


Recommendation

Re-order the operations so that division is delayed as much as possible in the computation, that is:

```
(asset_in_amount * 5 * issued_liquidity_tokens) / (20000 * input_supply)
```

The fee would be 0 iff

```
asset_in_amount < 4000 * input_supply / issued_liquidity_tokens
```



Therefore, when $\text{input_supply} < \text{issued_liquidity_tokens}$, the recommended reordering above would yield a positive fee value, while the original formulation would yield zero.

Status

This issue has been fully 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. Due to discussion about previous issues hitting code size limits, we have included information on potential code size reductions.

B01: Draining/Locking Up of Funds in Pools Involving Untrusted Assets

A pool may be created for an asset whose Manager, Clawback and/or Freeze parameters are set to non-ZeroAddress values. These values specify addresses of accounts that are authorized to perform certain authoritative functions over holdings of the asset. For example, the Clawback address if set means that the specified address is authorized to revoke holdings of this asset.

For Tinyman pools, this introduces a potentially serious vulnerability through which a malicious or a compromised account that has an authority over the asset of an Asset-Algo pool using one of these attributes may be able to either:

1. drain all assets **and Algos** from the pool, or
2. lock up all assets of the pool, preventing users from swapping assets or burning LP tokens.

Scenario

1. The attacker creates a revocable asset A (setting its clawback or manager address field to a compromised address).
2. The attacker bootstraps a pool for the pair A-Algo
3. Poolers (and maybe the attacker) fund the pool minting LP tokens
4. The attacker revokes all the asset amount back, bringing the pool's asset A balance to 0
5. The attacker then executes a 0-N swap (N is the amount of Algo in the pool minus the swap fee) to claim all the pool's Algo amount.

Note how the attacker managed to claim almost all Algos the poolers contributed at minimal cost. Similarly, an asset with a non-zero freeze address can potentially be used to lock up user funds, denying them from swapping or burning.

Recommendation

Despite being potentially catastrophic in theory, there are a few considerations that make this vulnerability less severe in practice:

1. Well-known assets are trusted and typically backed by entities that employ measures and mitigations to protect accounts with authority over their assets (e.g. using multi-sig accounts)
2. For less well-known assets, there is always this risk of the asset being part of a larger scheme to walk away with users' funds, so poolers in particular would need to be aware of these risks and ready to accept them for such assets. One way to ensure users' awareness of these risks is to direct users to descriptions of potential risks in the system's documentation and make sure that the frontend displays proper warning messages whenever dealing with pools involving untrusted assets.

Although creating pools involving assets with some of these configurable parameters set can be prevented by the smart contract, it would be too restrictive and would render Tinyman of little usability since many top and well-known assets in the Algorand ecosystem currently set at least one of these parameters. Furthermore, this authority over tokens is not new in Algorand as many ERC tokens in Ethereum, for example, provide similar authoritative functionality over token holdings. Algorand makes these properties more explicit and user-facing.

So the recommended mitigations for this issue generally lie outside of the scope of the smart contracts.

Status

As the recommended mitigations lie outside of the scope of the contracts, there is no immediate need to address them here. The development team is fully aware of the potential of this issue and will address the issue in the user-facing application's front-end.

B02: Alternative Protocol Fee Calculation Method

The protocol calculates the protocol fee and allocates the fee in LP tokens every time a swap operation is performed. Although this calculation method is probably the most natural and intuitive (since protocol fees are meant to represent $\frac{1}{5}$ of the swap fees), it may generally be likely to yield less protocol fees being allocated to the protocol than what should actually be allocated to more accurately represent this percentage due to truncation in integer division. To alleviate this effect, Uniswap v2 does not compute fees on the spot (when performing a swap). Instead, it computes the **accumulated** fees between mints/burns. Since swaps are likely to occur more frequently than mints or burns in the normal operation of a pool, the accumulation becomes less likely to yield zero fees. The way this is achieved is by computing the fees based on the difference in the square root of the pool's constant, which is the product of the `asset1_supply` and `asset2_supply` (See [Uniswap v2 white paper](#) for more details).

Recommendation

Implement Uniswap v2's method of computing the protocol fees, and evaluate the improvement that this approach provides. More specifically, the pool state now maintains a new variable, `kRootLast`, which is the square root of the pool's constant from the last mint or burn operation (initially is zero). Then, on every mint or burn,

1. compute the square root of the current pool constant:

```
kRoot = isqrt(asset1_supply * asset2_supply)
```

2. Use this and the last constant to compute the protocol fee:

```
if kRootLast == 0 or kRootLast >= kRoot:
    fee = 0
else:
    num = issued_liquidity_tokens * (kRoot - kRootLast)
    den = 5 * kRoot + kRootLast
    fee = num // den
```

3. Update the constant in preparation for the next mint/burn

kRootLast = kRoot

Initial simulation suggests that in practice this method yields a fee amount that is closer to the target 0.05% of trades on the long run. We recommend that further simulations are done to confirm this finding though,

Status

The development team conducted tests and simulations, which indicated that the suggested alternative methods for calculating the fee performs only marginally better while being a little less intuitive than the original method. Therefore, it was decided that there is no need to change the calculation method.

B03: Including Arbitrary Character Sequences in Liquidity Pool Asset Name

When bootstrapping a pool, the contracts verify that the name of the liquidity pool asset that is being created (the transaction indexed 2 in the bootstrap operation, T2) has a specific format, namely “Tinyman Pool X-Y”, where X is the asset unit name of the first asset in the pool’s asset pair and Y is the asset unit name of the second asset. To perform this check, the validator_approval contract expects that the IDs of the asset pair of the pool are passed as members of the foreign assets array to the application call (the transaction indexed 1 in the group, T1). The contract uses these parameters to read the asset unit names of the assets and compares them with the name supplied by the liquidity asset creation transaction T2. However, the contract does not currently check that the foreign asset IDs in T1 actually correspond to the asset pair of the pool, which means that the liquidity asset name can potentially include arbitrary asset unit names (provided that the unit names correspond to some assets in the blockchain).

Scenario

A malicious user may:

1. Create dummy assets with arbitrary unit names X and Y
2. Bootstrap a pool for a proper (actual) pair of assets, say USDC-GOLD\$, while supplying:
 - a. In T1, the asset IDs of the dummy assets whose unit names are X and Y in the transaction’s foreign assets array

- b. In T2, the arbitrary asset name “Tinyman Pool X-Y” for the liquidity pool asset

Recommendation

Check that the asset IDs given in the foreign assets array actually match the asset IDs of the pool stored in the pool account’s state. One simple way to do this in TEAL 4 (which is the version used by the `validator_approval` contract) is to use the actual asset IDs (rather than array indices) to index into the foreign assets array.

Status

This has been properly addressed.

B04: Using Tinyman as a Price Oracle

Tinyman may generally be used by other systems as a price oracle by querying the pool’s current supply of assets. However, it’s important to note that this price can potentially be manipulated by sandwiching a price query between two swaps in the opposite direction (at the cost of two swap fees). The attacker would need to make sure that these two swap operations are included in the same block at the right order, which is feasible only if the attacker happens to be in control of (or is able to influence) the validator node creating block.

Recommendation

The recommendation is either to:

1. include documentation of this limitation and advise developers to familiarize themselves with this possibility if they intend to use Tinyman as a price oracle, or
2. add additional logic that would facilitate the use of Tinyman as a reliable price oracle, while alleviating this issue of price manipulation.

Status

Tinyman developers added logic that included computing current prices and cumulative prices of assets over a period of time and maintaining them in the state of the pool

account. An additional layer of logic (in a separate contract) can now be implemented on top of Tinyman that can be used as a reliable price oracle.

B05: Code Readability/Maintainability Improvement Opportunities

I. pool_logicsic

1. Some labels use names of the form `[digit]+`. Replacing them with more descriptive names would be helpful.
2. It is generally better to use `TypeEnum` to check the type of a transaction instead of relying on whether `XferAsset` is zero. This is also to stay consistent with how similar checks are made in the contract.

II. valicator_approval

1. Some PyTeal-generated labels (e.g. `l47`, `l48`, ...) remain in the contract. It would greatly help readability/maintainability if more meaningful names for these labels are used.
2. Using location 113 instead of 104 for `outstanding_liquidity_token_key` in line 147 (and refactoring the code) is suggested to remain consistent with the convention used for assets (ids are in locations `10[1|2|3]` and keys are in locations `11[1|2|3]`).
3. Comments:
 - a. Typo: “suppply” (lines 553, 568, 651, 654, 660, 727, 729, 733, 756, 767) => “supply”
 - b. Typo in line 5: “3:” => “2:” (for `asset2_balance`)
 - c. The comment in line 9 does not seem to apply, and should be removed.
 - d. It would be useful to make the documentation of scratch memory slots used more complete. The suggested additions are:

```
// 52: Txn 2 asset or algo amount
```

```
// 53: Txn 3 asset or algo amount
```

```
// 61: outstanding_asset1_amount
```

```
// 62: outstanding_asset2_amount
```

```
// 103: liquidity_token_id
// 113: outstanding_liquidity_token_key (currently 104)
// 123: excess_liquidity_token_key
// 201: excess_asset1_amount
// 202: excess_asset2_amount
// 203: excess_liquidity_token
```

4. After bootstrapping a pool but before performing the first mint, attempting to perform any non-mint operation would (correctly) fail. However, this failure is manifested as either a division by zero, integer underflow or as an invalid access to a non-existent transaction. The contract would be more readable and robust (to changes in TEAL or changes in the txn group structure) if the contract rejected upfront any non-mint op when the `liquidity_token_id` is zero early in the logic, perhaps with an assert of the form:

```
assert(liquidity_token_id == 0 => AppArgs[0] == "mint")
```

This can also potentially enable simplifying the logic for the `swap_mint_burn` branch.

5. Check the “exists” flag of `asset_holding_get` with `assert` instead of discarding it with `pop`.

B06: Code Size Optimization Opportunities

Given the imposed size limits on TEAL smart contracts, it's very important to optimize code as much as possible (while maintaining code readability and logical structure as much as possible). Below we highlight a list of code optimization recommendations.


I. `pool_logicsic`

1. Use `int` instead of `byte` to represent Asset/App IDs. This would save four lines of code (Out of 8 locations where Asset/App IDs are referenced, 6 ``btoi`` ops can be removed while two ``itob`` will need to be added).
2. Lines 269-274 and Lines 311-318 share common computations that can be factored out to increase modularity and reuse, and decrease the size of the program. One way to accomplish this is as follows:

- a. In the previous basic block (just above line 267), store the value 0 at location 1 of the scratch memory
 - b. Remove lines 311-315 and also 317 (so this part will now simply store ``gtxn[4].Fee`` at location 1). Then, change the branch label in Line 319 to something like `“calculate_boot_fee”`, the label of the new basic block we are creating next.
 - c. Create the new basic block labeled `“calculate_boot_fee”`, which will have the instructions of lines 269-275 with a `“+”` opcode added right after line 269.
3. Lines 64-70, 79-85, 93-99, 107-113 and 121-127 check the same condition (that there is one application argument and the ``OnCompletion`` field is ``NoOp``). Since failing any of these checks results in panic, and since they are all done at the top of the non-bootstrap branches, they can all be replaced by an assert at the beginning of the non-bootstrap branch at line 64. This can be accomplished by just adding an assert right after line 70, and then removing blocks 79-85, 93-99, 107-113 and 121-127 so that branching within the non-bootstrap branch will be based only on the value of the application argument -- `“mint”`, `“burn”`, ... etc).
 4. The same ``select`` pattern used in lines 577-586 can be used in other locations where similar checks are asserted: checks that branch out based on ``XferAsset`` being 0 and ``TypeEnum`` being equal to ``pay``. These are: lines 378-386, 448-459, 478-489, 537-547, and 550-560. Although gains in terms of number of lines removed may not be significant when using ``select`` here, improvements in structure resulting from `select``'s implicit branching can be quite useful to readability and maintainability of the code.

II. valicator_approval

1. lines 758-760 compute the `asset_2_amount`, which is already computed earlier in a parent block in lines 66-68 and stored in location 52 (which is untouched in the entire contract). So lines 758-760 can be replaced by a `load 52`
2. Similarly, lines 687-689 compute `asset_3_amount`, which is already computed earlier in a parent block in lines 295-297 and stored in location 53. So lines 687-689 can be replaced by a `load 53`.
3. `burn`: label is not used, so it may safely be removed without affecting the logic. A commented-out label can be used instead for readability (like already done for other parts of the contract).
4. Line sequences 609-611 and 633-635 can each be replaced by a ``load 62``, since the ``outstanding_asset2_amount`` was already read from the pool's state and



stored in slot 62 in a parent block (and is untouched in between). Similarly, 616-618 and 627-629 can each be replaced by a `load 61` for the same reasons.

5. Another code optimization suggestion: The check `calculated_amount_in > 0` (Lines 666-668) seems redundant since the formula for `calculated_amount_in` guarantees that the amount will always be at least 1, assuming no arithmetic overflows or errors which would have caused TEAL to panic before reaching this point. (Note that if lines 666-668 are removed, then line 674 will need to be removed as well).

Appendix

We include here samples of artifacts created during the process of conducting this audit. One type of artifact is an automatically generated set of diagrams visualizing the control-flow graphs (CFG) of the programs' logic. The CFGs were generated using an extended version of the Tealer static analyzer by Trail of Bits available [here](#). Below (next page) is a sample showing an excerpt of the generated CFG of the validator contract.

Another type of artifact is a manually developed Python-like pseudo code that extracts the high-level logic implemented by the TEAL smart contracts. This pseudo code makes it easier and more natural to reason about the business logic of the application and cross-check it against the intended behaviors of the different components of the system. This pseudo-code later evolved into a Python-based simulator for Tinyman to investigate some economic aspects of the protocol (swap fees, protocol fees, LP token deflation rate, ... etc). Below is an excerpt of the specification developed for the validator_approval contract showing a specification of the burn logic:

```
update_oracle_if_needed()

assert(sender != pooler)
if (_liquidity_token_id != 0) #after first mint
    assert(burn_asset_id == _liquidity_token_id)
issued_liquidity_tokens =
    (MAX_UINT64 - _liquidity_token_balance) + _outstanding_liquidity_token

calculated_asset1_out =
    Lo64( (burn_amount *W _asset1_supply) /W (issued_liquidity_tokens *W 1) )
excess_asset_1 = calculated_asset1_out - asset1_out

calculated_asset2_out =
    Lo64( (burn_amount *W _asset2_supply) /W (issued_liquidity_tokens *W 1) )
excess_asset_2 = calculated_asset2_out - asset2_out

assert(calculated_asset1_out and calculated_asset2_out)

final_asset1_supply      = asset1_supply - calculated_asset1_out
final_asset2_supply      = asset2_supply - calculated_asset2_out
Final_issued_liquidity_tokens = issued_liquidity_tokens - burn_amount

update_local_state()
```