Tinyman Liquid Staking Audit

By: Mariano Dominguez

Vestige Labs

Disclaimer

This audit was conducted with a high degree of professional diligence and care, based on the information and vast resources available to us at the time of the review. However, the findings, interpretations, and conclusions expressed in this audit are intended solely to provide insights based on the stated scope of it.

It is important to note that this audit is informative and does not serve as an endorsement of the protocol under review. Despite my thorough examination, inherent limitations exist in any auditing process. As such, the audit should not be considered a guarantee against the presence of bugs, vulnerabilities, or unexpected behaviors within the code. Future updates to the protocol and new interaction contexts may introduce new risks not identified during this audit.

Clients and stakeholders are encouraged to consider this audit as part of a broader risk management strategy. It is recommended that ongoing reviews and monitoring of the protocol be undertaken to manage potential risks effectively.

Executive Summary

Overview

With the upcoming introduction of the first Algorand consensus upgrade block rewards similar to those present on other blockchains will come to Algorand MainNet, this opens up the possibility for new DeFi products that harness these rewards. Of these, Tinyman, Algorand's largest Decentralised Exchange (DEX), is expected to offer their own solution which I had the privilege of auditing.

The Tinyman Liquid Staking protocol consists of two main smart contracts:

- Liquid Staking Token minting and management smart contract: This contract is in charge of minting the Liquid Staking Token (from now on "LST"), as well as creating and setting the accounts that will manage the deposited ALGO stake that will go online and accrue rewards. It also manages the rebalancing operation to ensure that stake in any given node doesn't go above or below the threshold to accrue rewards.

- Re-staking management smart contract: This contract allows tALGO holders to "re-stake" their tokens to earn the Tinyman Governance Token (TINY). It's worth mentioning that this "re-staking" has no impact on the Algorand consensus layer and is just a method by which tALGO holders can earn TINY and thus have a say and a vote in the future of the DAO.

The actual node operation, i.e. the node runners that will employ the deposited stake of ALGOs, is managed independently from any smart contract or on-chain interaction (other than the actual going online system) by the Tinyman team. The Tinyman team has informed me that it is their intention to eventually delegate this choice of node-runners to the Tinyman DAO, but that this will be done at a further stage. Considering their prior commitment to open source and decentralization of the Tinyman DAO as a whole, I have no reason to doubt their plans.

Team Background

Vestige Labs: We are a premier blockchain software development company with a focus on building applications for the Algorand Virtual Machine (AVM). Our team consists of seasoned professionals that have developed a myriad of applications, with a focus on DeFi. We have over three years of real world experience working with Algorand-based applications, everything from developing independent applications to interacting with deployed applications. This audit was conducted by Mariano Dominguez, who has written and audited AVM-specific applications on almost every available language (PyTEAL, TEAL, TealScript, Reach and recently Puya).

Tinyman: Tinyman is the largest DEX on Algorand by total value locked (TVL) and volume. Having been on MainNet for over three years the protocol has had multiple iterations. With this last one (v2) being online for over 22 months without any problem. It's a source-available protocol under a Business Source License (BSL) that is now looking to launch its liquid staking solution (LST). The team members we interacted with during this audit were Fergal Walsh, Tinyman's CTO, and Attila Bora Semerci, a core contributor.

Other teams involved in the audit: The audit was carried out in a peer-review style, with the involvement of other ecosystem builders with a background in writing, maintaining, auditing and working with Algorand-specific applications. These other team members were Kevin Wellenzohn and Hans Mitterer from Blockshake (a DeFi-centric Algorand software studio, developers of Defly) and Steve Ferrigno, an engineering team member from the Algorand Foundation.

Scope of the Audit

The audit was primarily concerned with just looking at the smart contract code bases. The audited smart contracts were:

Contract	Commit Hash
tAlgo Contract	**95d9dc26ee0c452849a2c8734527d3475df0dc53**
Staking Contract	**95d9dc26ee0c452849a2c8734527d3475df0dc53**

As the audit was only looking at the given smart contracts, it was assumed that the compiler was “trustworthy”, i.e. that it compiled to the given opcodes in accordance with the Tealish documentation. Therefore the compiler itself was considered out of scope. I did not look at the outputted TEAL files for correctness in this regard.

As no UI was provided in the code base, we also assumed it to be out of scope, in accordance we also considered UI-related exploits (i.e. malicious links, phishing and spoofing through impersonating UIs) as out of scope.

Lastly, we considered network-wide issues as out of scope. This means any issues on Algorand’s security, as well as any issues or unexpected behavior on the AVM. This is important because the contracts interact with upcoming features that are still **NOT** live on Algorand MainNet and that consist of full-on consensus upgrades. Given the Algorand’s team commitment and record of zero down-time I don’t have a reason to believe that this upgrade will have issues, but it is a massive network-wide change, so some unexpected could show up.

Process

This audit was tackled in a similar way to the previous Tinyman governance audit. However, unlike in that process a more individual audit was carried out by each team, which we then shared points individually with the Tinyman team or in a set of meetings.

My approach was simple, first off I validated the stated model and from there thought of potential vulnerabilities it may have. From there I reviewed the provided code, annotating functions I believed would either benefit from sanity checks, or would be problematic. After that I shared my findings with the team, who either introduced the extra changes or explained their reasoning as to why they didn't think they were needed.

The allotted time for the whole audit was 3 weeks, with one week to familiarize ourselves with the general concepts behind the LST and then one week per smart contract.

When reading the code my main concerns were:

- **Arithmetic operations:** Validate that the operations are in accordance with the given documentation.
- **Under/Overflow risks:** Validate that the contracts can't under or overflow.
- **Proper permission assertion:** Validate that only admins can call admin-specific interactions, and that these are properly handled by the contract.
- **Group txn assertions:** Validate that the grouping of transactions doesn't introduce a possible vulnerability or that checks are carried out properly.
- **Loss of funds:** Validate that the contract doesn't enter a state where users interacting with it as intended lose funds.
- **State spoofing:** Validate that users can't game the contract's mechanics to appear as if they held a higher stake than they have.
- **Logic coherence:** Validate that the contract's logic is consistent with its documentation and intended state.

As I tried to validate the contracts, whatever findings I found were classified based off of the following table:

Severity Classifications	
Severity	**Description**
Informational	The issue does not pose an immediate risk but is relevant to security best practices.
Undetermined	The extent of the risk was not determined during this engagement.
Low	The risk is small or is not one the client has indicated is important.
Medium	User information is at risk; exploitation could pose reputational, legal, or moderate financial risks.
High	The flaw could affect numerous users and have serious reputational, legal, or financial implications.

Difficulty Classifications	
Difficulty	**Description**
Undetermined	The difficulty of exploitation was not determined during this engagement.
Low	The flaw is well known; public tools for its exploitation exist or can be scripted.
Medium	An attacker must write an exploit or will need in-depth knowledge of the system.
High	An attacker must have privileged access to the system, may need to know complex technical details, or must discover other weaknesses to exploit this issue.

Lastly, it is worth mentioning that in an effort to ensure diligence when reviewing and reading the code the Tinyman team introduced one known vulnerability to it. This vulnerability is highlighted on my chart, and should be dismissed.

Findings

ID	Summary	Severity	Difficulty
01	**Buffer Attack**	Informational	Hard
02	**tALGO Token Amount**	Informational	Undetermined
03	**tALGO Airdrops**	Informational	Undetermined
04	**Claimed Protocol Rewards (THIS WAS AN INTENTIONAL BUG)**	High	Low
05	**Loss of Admin Keys**	Informational	Undetermined
06	**Sanity Assertions**	Informational	Undetermined
07	**Unrestricted Fees**	Medium	Hard
08	**Asserting Payment Possibilities**	Informational	Undetermined

Smart Contracts

Liquid Staking Manager

Overview

The liquid staking manager contract is a singleton (ie there's only ever one instance of it) in charge of a few things:

-Creating and minting the LST ASA (tALGO)
-Creating 5 addresses which it will delegate ALGO stake to
-Allowing users to mint tALGO by depositing ALGO
-Allowing users to burn tALGO by depositing ALGO, getting ALGO + Rewards back from this action
-Managing the stake between the 5 addresses, ensuring that they never drop below the rewards cap nor that they go above it.
-Sign the online participation registration keys to allow for consensus participation through a LogicSig (this design decision is to allow for any future changes on the key registration signature) *****

There are a few model limitations that the tALGO system has, this is a current limit on the amount of tALGO that is allowed to mint (345 million ALGOs) with each account being given a 1m ALGO buffer to ensure they don't go over the 70m rewards threshold.

***The LogicSig system was removed from the final codebase after review by one of the auditors (Steve Farrigno from the Algorand Foundation), since its removal didn't impact the contract flow, I simply reviewed the pull request and merge, but didn't re-do my report due to, in my opinion, it not being necessary**

Key Findings

01	**Buffer Attack**	Informational	High

Since tALGO has a limit on how much it can hold in each online account to accrue rewards, if close to the limit (70m) a potential malicious actor, or competing protocol could airdrop an amount to push it past that limit, thus kicking it out from rewards accruing. Whereas this is a **VERY** expensive exploit, since these are consensus parameters that might be subject to change, I recommend a “flush” function where funds past the buffer can be taken out of the contract. The buffer has since been increased to 5m, making the attack even more expensive, and the balance is updateable in case this ever changes at a protocol level.

02	**tALGO token amount**	Informational	Undertermined

tALGO is minted with a total supply of 10b. This amount is chosen semi-arbitrarily to correspond to the amount of ALGOs at a protocol level. However, not only should this amount never be needed (since the protocol itself has a cap on how much it can mint), but it’s highly unlikely that whole network token supply will be in the protocol. To add to it, tAlgo, as a token that represents the deposited ALGO + the rewards, tends towards zero as rewards are accrued, so it reinforces the idea that the amount of needed tokens by the protocol is arbitrary. Because of this I recommend to either cap the minting to the amount allowed by the protocol (i.e. 345m), or going with the max amount allowed (UInt64 Max) using the creation address as the reserve address (as it currently stands) so that its circulating supply can be properly tracked. The 10Bn tALGO amount was chosen as the final amount by the Tinyman team.

03	**tALGO Airdrops**	Informational	Undertermined

Since tALGO is created by the smart contract upon initialization, should a user forcibly opt-out of the ASA, or alternatively deposit the token directly into the smart contract, the underlying ALGOs are effectively lost forever. I believe an admin function that checks for current held balance, comparing it to minting amount and returns to the admin any discrepancies, should there be, is a safe method that introduces no further risk and that can allow for the recovery of the underlying ALGOs, especially since tALGO can only be minted or burned through the contract. A function to ensure minted tALGO becomes donated ALGO and not locked ALGO has since been implemented.

04	**Claimed Protocol Rewards (THIS WAS AN INTENTIONAL BUG)**	High	Low

THIS WAS AN INTENTIONAL BUG INTRODUCED BY THE TINYMAN TEAM THAT HAS SINCE BEEN REMOVED.

The claiming protocol rewards system, which pays the Tinyman treasury the rewards set by the protocol doesn't actually set them to zero after claiming, this means that the contract can be drained by anyone by simply spamming this function. The ALGOs go to a "trusted" address, but it's still a way to drain the contract through the over-minting of tALGO which can then be burnt until reaching zero.

```
# permission: anyone
@public()
func claim_protocol_rewards():
    transfer_talgo(UncheckedCast(app_global_get("fee_collector"), bytes[32]),
app_global_get("protocol_talgo"))
    log(ARC28Event("claim_protocol_rewards(uint64)",
itob(app_global_get("protocol_talgo"))))
    return
end
```

The fix is to reset the "protocol_talgo" amount to zero after the transfer.

05	**Loss of Admin keys**	Informational	Undetermined

The LST contract uses a manager address that can determine the roles of the staking operator. This manager can be updated through the following function:

```
@public()
func set_manager(new_manager: bytes[32]):
    assert(Txn.Sender == app_global_get("manager"))

    app_global_put("manager", new_manager)
    log(ARC28Event("set_manager(address)", new_manager))
    return
end
```

As we can see from it, there are no safe guards, this means that the manager is immediately upgraded upon a successful call. Because of this should the admin address ever be compromised the manager can be “locked out” immediately, or should there be a typo when calling this address the changes would be final and irreversible. I recommend using either a time lock, or a two-step verification process where the incoming address has to accept the new managerial role, at least as a way to verify that the changes are intended.

However, it is also important to mention that no method is infallible and that the design of the protocol is such that the actual underlying ALGOs are mitigated from any malicious actor that does indeed manage to get access to the managerial role. At most the protocol rewards are at risk. This two step proposal system has since been implemented.

06	**Sanity Assertions**	Informational	Undetermined

Throughout the contract there are a few assertions that considering the model can be explicitly stated in the code. I didn’t find any that were code breaking, but these can be considered best practices since they ensure that the behavior is as intended. For example:

```
@public()
func set_node_manager(node_index: int, new_node_manager: bytes[32]):
    bytes[32] user_address = Txn.Sender
    assert(user_address == app_global_get("manager"))

    bytes key = concat("node_manager_", ascii_digit(node_index))
    app_global_put(key, new_node_manager)
    log(ARC28Event("set_node_manager(uint64,address)", itob(node_index),
new_node_manager))
    return
end
```

Here the *node_index* can be asserted to *<5* set since there can only ever be 5 node operators, as can it be in the function:

```
@public()
func change_online_status(node_index: int):
...
end
```

Or in the *ascii_digit* one where, we *i* has to be *<10*:

```
func ascii_digit(i: int) bytes:
    return extract(7, 1, itob(i + 48))
end
```

None of these functions have a problem *per se*, but asserting the expected parameters is an easy way to avoid unexpected behavior. These assertions have been added.

07	**Unrestricted Fees**	Medium	High

In the given code there are no limits to the fees the manager can set:

```
# permission: manager
@public()
func set_protocol_fee(fee_amount: int):
    bytes[32] user_address = Txn.Sender
    assert(user_address == app_global_get("manager"))

    app_global_put("protocol_fee", fee_amount)
    log(ARC28Event("set_protocol_fee(uint64)", fee_amount))
    return
end
```

Because of this, should the manager ever be compromise, or should a rouge team member ever choose to do so, he/she could set the fees to whatever they so choose, in so doing they could have the protocol “steal” from the user by setting fees to a max so as to ensure that all rewards are accrued by them. It should not affect the principal, since no fees are charged on mint, just on the update of rewards, in accordance to Tinyman team’s design principles of ensuring that the initial funds are never at risk. It has since been updated.

Re-Staking Manager

Overview

The “re-staking” contract is a contract that allows users to “re-stake” their tALGO to earn TINY, which can later be used on the Tinyman DAO. It is worth mentioning that this “re-staking” doesn’t impact consensus at all, but the name is used since it’s the common nomenclature used by the industry whenever a contract emits tokens as rewards, whether these come from consensus participation or not. The contract also emits a token (stALGO), however this token has no monetary value and is frozen, it simply acts as an ASA that is always 1:1 pegged to tALGO and is an easy way for portfolio management tools to track the value of a given account without looking at that account’s smart contract state. Lastly, I should mention that the contract’s TINY allocations come from an exogenous source, that is to say that the TINY is deposited by an agent (be it a DAO and/or admin) outside of the scope of the contract. This means that there are underlying trust assumptions that this agent fulfills their duty and is outside of the scope of this audit.

Key Findings

08	**Asserting Payment**	Informational	Undetermined

In the reward rate function setting:

```
# Permission: manager
@public()
func set_reward_rate(total_reward_amount: int, end_timestamp: int):
    assert(Txn.Sender == app_global_get(MANAGER_KEY))

    assert(total_reward_amount)
    assert(end_timestamp > Global.LatestTimestamp)

    # Wrap up, accumulate for the last rate.
    update_state(Global.LatestTimestamp)

    int duration = end_timestamp - Global.LatestTimestamp
    int reward_rate_per_time = total_reward_amount / duration

    int current_reward_rate_per_time_end_timestamp =
app_global_get(CURRENT_REWARD_RATE_PER_TIME_END_TIMESTAMP_KEY)
    int last_current_reward_rate_per_time =
app_global_get(LAST_REWARD_RATE_PER_TIME_KEY)
```

```
    int current_reward_rate_per_time =
app_global_get(CURRENT_REWARD_RATE_PER_TIME_KEY)

    int max_rate_per_time
    if last_current_reward_rate_per_time < current_reward_rate_per_time:
        max_rate_per_time = current_reward_rate_per_time
    else:
        max_rate_per_time = last_current_reward_rate_per_time
    end
    max_rate_per_time = max_rate_per_time + ((max_rate_per_time *
MAX_RATE_INCREMENT_PERCENTAGE) / 100)

    # Check is disabled for initialization.
    if max_rate_per_time:
        assert(reward_rate_per_time < max_rate_per_time)
    end

    app_global_put(CURRENT_REWARD_RATE_PER_TIME_KEY, reward_rate_per_time)
    app_global_put(CURRENT_REWARD_RATE_PER_TIME_END_TIMESTAMP_KEY, end_timestamp)

    log(ARC28Event("set_reward_rate(uint64,uint64,uint64,uint64)",
itob(total_reward_amount), itob(Global.LatestTimestamp), itob(end_timestamp),
itob(reward_rate_per_time)))
    return
end
```

Rewards are deterministic, because of this it can be known ahead of time whether the new set of rewards can indeed be paid through a few asserts. This is a great sanity check since it can avoid any fat fingering or unexpected behavior from a reward amount that's set too high. The Tinyman team has since employed these assertions.

Final thoughts

Once again, I am impressed by the quality of the code delivered by the Tinyman team. Their LST is both simple and intuitive, reflecting thoughtful design. While some may argue that its federated system challenges the principles of decentralization—given that trust is placed in a pre-selected set of node operators—this concern can be mitigated through their existing governance framework. Considering the team's strong commitment to decentralization, I am confident they will address this in the future, including refining the node operator selection process and enhancing the TINY rewards funding system for their re-staking smart contract.

Moreover, their dedication to ensuring user custody of funds is a standout feature. This design principle not only strengthens the system's integrity but also provides users with a vital guarantee of autonomy—users can always reclaim their principal. This level of commitment is both commendable and critical to fostering trust in the protocol.

Additionally, the team's advocacy for source-available smart contracts deserves recognition. By championing transparency, they add an extra layer of security, enabling stakeholders—whether power-users or other ecosystem developers—to continuously audit and monitor the protocol. This openness enhances the protocol's reliability and fosters a collaborative environment for ongoing innovation and improvement.