Tinyman Governance Audit

By: Erik Hasselwander and 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 our 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

The Tinyman governance protocol seeks to be a fully decentralized autonomous organization (DAO) that governs the Tinyman decentralized exchange (DEX), this protocol is composed of a series of four smart contracts through which it can execute its decision making. These smart contracts are inspired by other popular governance protocols on other blockchains, specifically the CRV DAO protocol (link). Most interactions are based around Tinyman's incoming governance token $TINY. As this token is locked for periods of time, "Tiny Power" is accrued. Which in turn allow for further interactions with the DAO, such as giving users of the protocol rewards. The core contracts are:

- Vaults: Responsible for staking and tallying the amount of "Tiny Power" governors have. Essentially it is the main escrow where $TINY is locked into.

- Rewards: Responsible for distributing rewards to users based on their "Tiny Power".

- Staking Voting: Governors can vote to allocate liquidity mining rewards to different liquidity pools.

- Proposal Voting: Governors can create arbitrary proposals on-chain to then vote on.

These contracts have different degrees of decentralization, it's worth mentioning considering that some core functionality of the contracts require the direct involvement of the Tinyman team, specifically Rewards (since the team has to fund the rewards of TINY to be claimed), Staking (since the team has to create the chosen liquidity farming programs) and Proposals (since the team still has to execute the voted-on proposals). This places a high degree of trust in the team, and we're in an industry that shouldn't trust, but instead verify, in this case it is impossible for us to do so. Users should know that they're placing trust in the Tinyman team during this first iteration and that the Tinyman team is a single point of failure, something that is not the case with the Tinyman v2 DEX. Rekeying the admin accounts to smart contracts will solve this, and it's something the team has said they are committed to. But right now we can consider the protocol to be permissionless, but not trustless.

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 two of our team members, Erik Hasselwander and Mariano Dominguez, both of whom have written and audited AVM-specific applications on almost every available language (PyTEAL, TEAL, TealScript, Reach and recently Puya). Prior to this, neither team member had done formal work with Tealish, but were familiar with it due to having worked with the Tinyman v2 protocol (which was written in Tealish). Because of this, the team felt more than comfortable carrying out the audit.

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 18 months without any problem. It's a source-available protocol under a Business Source License (BSL) that is now looking to launch its governance program that will transition it into a Decentralised Autonomous Organization (DAO). The team members we interacted with during this audit were Fergal Walsh, Tinyman's CTO, and Attila Bora Semerci, a core contributor. It's worth mentioning that most of the contracts were written by a former team member that wasn't available for this audit. However, both Fergal and Attila were available for the process and were extremely familiar with the codebase, and thus could answer and clarify any questions we had during the process.

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, audinting and working with Algorand-specific applications. These other team members were Gidon Katen, the CTO of Folks Finance (a lending platform on Algorand), Kevin Wellenzohn, a co-Founder of Blockshake (a DeFi-centric Algorand software studio, developers of Defly), Steve Ferrigno, a DevRel team member from the Algorand Foundation and D13, a well known Algorand community member that has worked with a myriad of DAOs

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
Vault Approval	**c2d0e4741761719daa238bed4190c24981d991b3**
Rewards Approval	**c2d0e4741761719daa238bed4190c24981d991b3**
Staking Voting Approval	**f908cd543f72bf6882491d4f52d5159c62622121**
Proposal Voting Approval	**f9bf5ac6c8100cac4525df57306350a94f137a35**

Each one carried a corresponding clear state program that returned error, therefore for sake of brevity we didn't include them on the table above.

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 for the contracts' version. Therefore the compiler itself was considered out of scope.

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. Since this is a timestamp sensitive smart contract, some considerations went into a hypothetical halt followed by a restart. Since in this case Algorand has a slow ramp up to catch up with the current UNIX timestamp we couldn't think of an unexpected interaction that would show up or a potential exploit that this would introduce.

Process

This audit was tackled in an innovative way for the Algorand ecosystem, and is as far as we're aware the first endeavor of this kind. A work group of ecosystem experts was formed and in this work group a collaborative communication channel was opened where we shared findings as each member found them. We decided to simply put all the findings shared in this document, regardless of whether we, or other members of the group found them. Because of this it is likely that our report will have repetition when compared to others', and given the collaborative nature of the process it's hard to give individual credit to any one member.

The approach that was taken was having two team members (Erik and Mariano) look at the code from different perspectives. Erik went straight into the code, dismissing the provided documentation, this was so he wouldn't be swayed by the documentation's explanations, instead reading the code "as-is" to have as little preconceived ideas as possible. Whereas Mariano did the inverse, first reading the provided documentation, as well as doing in-depth research of the codebases the protocol was inspired by (the CRV DAO smart contract), without looking at the code until he was familiar with the expected functionality. All of this was done, because the more ways we tackled this audit, the more likely we were to find any potential issues and thus the more helpful this exercise would be. At the end of the day, the objective is to provide a comprehensive revision of the code that minimizes the risk of loss of funds, unexpected behavior and potential hacks.

The allotted time for the whole audit was 5 weeks, with one week to familiarize ourselves with the general concepts behind the DAO and then one week per smart contract. Although each group member went through the code at their own pace. We had general weekly meetings to go over any potential issues and findings, as well as discuss the intention of the code with Tinyman members as well as all other group members. During these meetings the Tinyman team was more than helpful sharing any concerns and questions we had, overall their professionalism and general interest during this whole process has been commendable.

When reading the code our 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 we tried to validate the contracts, whatever findings we 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.

Findings

ID	Summary	Severity	Difficulty
01	**User can delete his/her voting power**	Informational	Medium
02	**Risk of running out of storage when upgrading**	Informational	High
03	**Loss of admin keys**	Informational	Undetermined
04	**Centralization concerns**	Informational	High
05	**Inconsistent use of data types**	Informational	Undetermined
06	**Unnecessary payment fork**	Informational	Undetermined
07	**Box MBR claiming**	Low	Medium
08	**Funding rewards**	Low	Low
09	**Un-deployability**	Low	Low
10	**Rounding loss of precision**	Low	Low

Smart Contracts

Common Features

Overview

Across the smart contracts there were common design choices and snippets that were used multiple times, or alternatively there were some findings that would've impacted all the smart contracts even if they were present in only one of them. Because of this, we separated these findings into their own category.

Key Findings

01	**User can delete his/her voting power**	Informational	Medium

A user is able to delete his/her voting power by deleting his/her box. Whereas this is more on the side of an informational finding, since it's in a sense expected behavior (i.e. the account that funded the box can delete said box), a lot of care and attention should be placed so as to have the user aware of his/her decision. We have seen previously, when local state as a design choice was more common for state management, that users inadvertently deleted said state and caused loss of funds. In this case the damage is much more mitigated, but mentioning it is important, if only to avoid poor outcomes. We don't believe that there should be any code changes, since this is an issue that should be resolved at the UI level, considering that power-users should be more familiar with the risks they're taking when deleting their voting power.

02	**Risk of running out of storage when upgrading**	Informational	High

Some contracts (Staking Voting and Rewards) are upgradeable, this means that should they ever be upgraded, they'd be limited to the size they were given at creation. This applies to their local and global state schema, and number of pages allocated. Because of this we recommend the maximum possible amount of size is allocated, just in case it's ever needed.

03	**Loss of admin keys**	Informational	Undetermined

At the moment the Rewards, Staking Voting and Proposal Voting contracts intend to use a Tinyman admin multisig as its admin key that can call all admin functions. This admin multisig can be upgraded using the *set_manager* function:

```
func set_manager(new_manager: bytes[32]):
    bytes user_address = Txn.Sender
    assert(user_address == app_global_get(MANAGER_KEY))

    app_global_put(MANAGER_KEY, new_manager)
    log(Concat(method("set_manager(address)"), new_manager))
    return
end
```

As we can see, there's no safeguard other than the assertion of the MANGER_KEY. Because of the sensitive nature of the role, we recommend a time delay verification when calling this function. This way, an update of this kind can't lock out the team from it immediately, and should there be a need to migrate to a new set of admin keys there's some time to do so. However, we understand that this is a team choice, and there is no infallible way of managing this.

04	**Centralization concerns**	Informational	High

As we have mentioned before in the audit, the contracts can be considered permissionless, but not trustless, and it's imperative that users understand this since it's a core difference from previous products offered by the Tinyman team. The team is a bastion of decentralization and has assured us that their intention is to decentralize in the near future. However, that is not the reality right now, and whereas there's no real risk on the contract that holds users' funds (the Vault contract), the DAO can only operate through the good-will of the Tinyman team, which shouldn't be the case. We have no reason to believe that the Tinyman team won't follow through on their intentions, and are very optimistic that they will, but it's important that users are aware of this, and make their own decisions based on this knowledge.

05	**Inconsistent use of data types**	Informational	Undetermined

There are inconsistencies in the code when it comes to data types. As we were reading it, we were surprised to find that some functions that were functionally the same, had assertions placed in different spots for no real reason. For example:

On Proposal Voting we have:

```
@public()
func set_manager(new_manager: bytes):
    bytes user_address = Txn.Sender
    assert(user_address == app_global_get(MANAGER_KEY))

    assert(len(new_manager) == 32)
    app_global_put(MANAGER_KEY, new_manager)
    log(Concat(method("set_manager(address)"), new_manager))
    return
End
```

And on Staking Voting:

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

    app_global_put(MANAGER_KEY, new_manager)
    log(Concat(method("set_manager(address)"), new_manager))
    return
end
```

Whereas these two methods are functionally equivalent, the first one doesn't do the parameter type check at the function declaration, instead opting for an assertion at a lower level of the byte length, whereas the second one does it at the function declaration. We believe the later one is better in style and readability. Therefore we would prefer to solve these slight consistency problems in the few functions where they were present.

06	**Unnecessary payment fork**	Informational	Undetermined

In the Vault and Rewards contract there was a function for carrying out transfers that was copied from Tinyman v2 AMM:

```
func transfer(asset_id: int, amount: int, sender: bytes, receiver: bytes):
    # This function is copied from Tinyman AMM Contracts V2.
    # "asset_id == 0" is updated as "!asset_id" for budget optimization.
    #
https://github.com/tinymanorg/tinyman-amm-contracts-v2/blob/main/contracts/
amm_approval.tl#L1146

    if !asset_id:
        inner_txn:
            TypeEnum: Pay
            Sender: sender
            Receiver: receiver
            Amount: amount
            Fee: 0
        end
    else:
        inner_txn:
            TypeEnum: Axfer
            Sender: sender
            AssetReceiver: receiver
            AssetAmount: amount
            XferAsset: asset_id
            Fee: 0
        end
    end
    return
end
```

This function has the possibility of either paying an Algorand Standard Asset or Algos. Considering the contract is intended to work solely with TINY and we couldn't think of a situation that would require this fork, like it would on an AMM we found the inclusion of this possibility as unnecessary, therefore recommend removing it altogether.

Vault

Overview

The Vault contract is the main “escrow” contract in the whole DAO protocol. It acts as the place where users “lock up” their $TINY for chosen periods of time, after which they get “TinyPower” based off the amount they lock and for how long they lock for. The function to determine the amount of TinyPower is based on the CRV-DAO protocol, with some slight differences, such as the introduction of a minimum amount of locked amount and time period, as well as a concept of Cumulative Power, which increases over time. Other than that, some AVM-specific changes are introduced due to the way it differs from the EVM, more specifically the need to build txn references off-chain, due to the lack of a native array support.

Key Findings

07	**Box MBR claiming**	Low	Medium

Boxes on the Algorand can act as a type of global state extension, because of this Algorand limits their creation by requiring the depositing of a minimum balance, to avoid the spamming of them. This minimum balance requirement (MBR) is deposited on the smart contract’s address. Because the Vault contract allows anyone to open a position, it doesn’t do a check to see if the user paid its MBR, this is to be as flexible and composable as possible. Because of this, should the contract ever find itself with more ALGOs than it needs to keep users minimum balances these ALGOs are “claimable”, through a mechanism of creating the lowest possible position, and then closing it (since at close the MBR is returned to the person that deleted the box). This quirk means that any ALGO in the can be “maliciously” claimed, which sounds drastic, but considering the contract isn’t meant to deal with ALGOs, barring MBR-payments it’s safe to assume any and all ALGOs that do make it into the contract were either air-dropped or donated to the Vault, and can thus be considered “available to the public”. Because of this, we didn’t recommend any code changes based off of this finding, it is worth mentioning since this is likely going to be one of the most referenced code bases on Algorand and thus other Algorand builders should have this quirk in mind when thinking about building applications that use boxes.

Conclusion

As we hope can be appreciated from the lack of critical findings, the general structure of the code, its intention and the execution of it is of top quality. We found a very clear description of what the team sought out to build and a codebase that matches those intentions pretty clearly with no real eyebrow raising issues.

Rewards

Overview

The Rewards contract is the one that's in charge of distributing $TINY to governors based on their Cumulative Power, it calls the Vault contract to fetch governors' Cumulative Power and from there do a weekly reward distribution, with each governors' claim being proportional to their Cumulative Power.

Key Findings

08	**Funding rewards**	Low	Low

It's worth mentioning that the rewards paid out to governors are funded into the contract by the Tinyman team, should this funding event not take place the contract will fail in its execution since it won't have the funds to pay-out. Even though this is functionally catastrophic, since it breaks the core purpose of the contract, the reason why it's a Low finding is because this should already be expected when interacting with the contract (as we have mentioned multiple times in this document), and no user funds are at risk, just promise of future rewards. Lastly, rewards for the period can be consulted on-chain as can the balance of the contract, therefore any well versed user can audit balances and from this conclude if rewards for the period will be paid out, since once funded it is effectively under the control and management of the contract.

Conclusion

The rewards contract, whereas part of the not-trustless triad of contracts, is in our opinion the one that's the least affected by it. This is because unlike the following two, it has a set of "fail-safes' " where the user can audit balances and rewards periods and from there conclude that payouts will occur, therefore other than funding the involvement from the Tinyman team is minimal.

Staking Voting

Overview

The Staking Voting contract allows governors to vote on liquidity mining incentives for different pools in the Tinyman DEX. Votes are recorded, and based on off-chain analytics the corresponding farms are created using the already developed Tinyman farm smart contracts. It's worth mentioning that the contract has no real "binding", that is the contract itself only creates a tally, and doesn't create or manage the farms that are voted on.

Key Findings

09	**Un-deployability**	Low	Low

The committed wasn't deployable, this is because of the following lines of code:

```
@public(OnCompletion=CreateApplication)
func create_application(vault_app_id: int):
    app_global_put(VAULT_APP_ID_KEY, btoi(Txn.ApplicationArgs[0]))
```

This call wouldn't pass a vault_app_id and thus would fail on launch. We did a hotfix on our build to run the contracts through the provided tests, but it's worth mentioning that should a user run the commits on this document they will experience this error. Obviously, this error would be caught on MainNet since it wouldn't allow for the application to exist.

10	**Rounding loss of precision**	Low	Low

In the function *cast_vote* a division is performed first when updating the vote amount. This division causes a slight loss in precision:

```
        # update vote amounts
        option_vote_amount = tmp_vote_percentage * (account_voting_power /
100)
```

Considering that the risk of overflow isn't there due to tmp_vote_percentage being at most 100, it is safe to first multiply by it and then divide, without running the risk of overflowing, and without the loss of precision.

Conclusion

In general the contract was straightforward, since it’s relatively simple in nature, since it acts more as a tally for other events that occur off-chain more-so than handling complex logic itself.

Proposal Voting

Overview

The proposal app allows governors to propose and vote on arbitrary motions that have to do with the Tinyman DAO. To avoid spam, the proposals are limited to governors that have above a threshold of Tiny Power and in this current version a manager has to also approve of the proposal (this is a power that can be revoked). Once the proposal is approved, a minimum level of quorum is needed for the proposal to be either accepted or rejected.

Key Findings

There were no findings to report on the Proposal Voting App that weren't detailed before.

Conclusion

In general, much like the Staking Voting app, since the Proposal Voting app acts mainly as a tallying contract, with some slight logic to allow for voting at different periods of time, but not the actual execution of the proposals, it is rather simple and therefore straightforward in its functions. This is a good thing when looking at the code, since the potential pitfalls are low, especially since any distribution of funds or employment of them isn't done directly through this contract.

Final thoughts

Overall, as we worked with the code we were surprised with how straightforward in its intentions and execution of them, with very little surprises in its design choices. We were unable to find any serious critical findings, a testament to the expertise of the Tinyman team as seasoned builders in the space. Not only this, but what little we did find they resolved in no time and pushed changes for when needed, or where open ended in nature, with their design choices being perfectly valid solutions.

However, we can't stress enough that these protocols are inherently centralized. Yes, the vault smart contract is fully permissionless and probably one of the most sophisticated token-based DAO management systems we've seen on Algorand that can operate fully autonomously. But the two contracts through which this voting power is used are at the moment dependent on the Tinyman team, and the app through which the token is distributed also requires their direct funding (granted this can be done for multiple periods, and is therefore less of a concern).

From our conversations with the Tinyman team and considering their previous track record it is likely that this will change in the near future and that the Tinyman protocol will transition into one of the first operational DAOs on Algorand, something we're personally excited for.

Lastly, we'd like to thank everyone involved in this process, as the Algorand ecosystem matures, it is great to see that we can now have seasoned builders in the space audit and comment sophisticated code in a peer-reviewed fashion. We are believers in decentralization and community-led security efforts over third-party audits, if only because in our experience the community is filled with members that interact with these type of protocols day to day and are very aware of the potential security pitfalls Algorand–specific applications can have and how to best avoid them. We're grateful for being included in this experiment, and hope our insights were helpful to the Tinyman team, who we want to commend for spearheading this initiative.