BLOCK SHAKE

[DRAFT] Tinyman Consensus Protocol

Security Audit

November 15, 2024

Prepared for:
Tinyman

Prepared by:
Hannes Mitterer, Kevin Wellenzohn
Blockshake

Blockshake GmbH
Dr.-Helmut-Marsoner-Weg 3C
6175 Kematen in Tirol, Austria

Hypo Bank Tirol
IBAN: AT60 5700 0300 5569 0178
BIC: HYPTAT22XXX

Register Court: Innsbruck, Austria
VAT-ID: ATU77728808

BLOCK SHAKE

BLOCK SHAKE

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 the 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

BLOCK SHAKE

Introduction

The Tinyman team engaged Blockshake to audit their upcoming consensus system in a peer reviewing process alongside other well-known companies & individuals in the Algorand ecosystem. The audit was conducted between November 1, 2024 and November 15, 2024 and we focused on the following objectives:

- Provide Tinyman with an assessment of the security of their smart contracts
- Provide recommendations to improve the security where possible

Overview

The audited code consists of two components, the tALGO contract and the tALGO restaking contract, that target Algorand's upcoming Réti upgrade. With this upgrade node runners are incentivized through ALGO rewards to participate in consensus and secure the network. Since the network topology plays an important role for the health of the chain, it was decided that a node runner requires between 30k and 70M ALGO to earn rewards. Too few nodes with very large accounts (> 70M ALGO) pose a centralization risk while too many nodes with small ALGO stakes (< 30k ALGO) risk attracting poor-quality nodes that prove too unreliable. Because of these limitations, Tinyman proposes a pooling service for ALGO holders.

The tALGO contract allows ALGO holders to pool their ALGO for consensus and earn rewards. In exchange for their ALGO, they receive a newly minted asset, called tALGO. This asset allows users to stay liquid and participate, e.g., in DeFi while they provide their ALGOs for consensus. Users can burn their tALGO at any point in time and get back their ALGO with any potential rewards that have accrued in the meantime.

The tALGO restaking contract allows tALGO holders to stake their tALGO and earn TINY tokens as a reward. When users stake their tALGO, they receive stALGO in exchange and they can return their stALGO at any time to regain their original tALGO.

Scope

In-Scope

This audit's findings are based on commit [3e0b40ca9090a5c9d2203732c32d1699bb043794]() and we considered the following source code files

- `contracts/talgo/auxapp.tl`

- `contracts/talgo/clear_state.tl`
- `contracts/talgo/keyreg_logic_sig.tl`
- `contracts/talgo/talgo.tl`
- `contracts/talgo_staking/talgo_staking_approval.tl`
- `contracts/talgo_staking/talgo_staking_clear_state.tl`

It was stated by the Tinyman team that this code will be open sourced upon launch of the contracts.

We were provided with supplementary documentation:

- A system description, presented as Google Docs document, that briefly describes the overall architecture and each individual contract.
- A link to a publicly-available Github repository that explains the math used in the staking contract.

Out-of-Scope

The above source code files are written in Tealish, a programming language developed by the Tinyman team. We did not inspect the TEAL files that Tealish generates, nor did we attempt to verify that the Tealish compiler outputs valid TEAL code. This aspect was covered by a different auditor.

We did not audit the web application / user interface that Tinyman users will use to interact with the audited contracts.

Methodology

Exploration

We started the audit with a thorough review of the provided documentation to understand the system's overall expected functionality and design. Since the system consists of several smart contracts, we looked at how the contracts interact with each other and what dependencies exist between them.

Manual code review

We analyzed each smart contract and focused on the following areas that are a common source of bugs and vulnerabilities:

- **Data storage (global state / box storage)**: the contracts use global state and box storage to persist all relevant state. We ensured that only the correct users are able to manipulate the state at the appropriate times.
- **Arithmetic overflow / underflow**: Algorand smart contracts fail upon integer overflows or underflows in arithmetic operations. This can block contracts and lock up funds if exploited by attackers. We checked all arithmetic operations and the quantities they operate on to detect any potential overflows or underflows. We computed the ranges of all important quantities to see if they can be represented within their allotted integer type (e.g., `uint64`).
- **Permissioned methods**: some of the contracts have permissioned methods that can only be called by specific users / Algorand accounts. We checked that these methods can indeed only be called by the appropriate accounts.

Besides the above general security checks, we studied each individual contract in depth, looking for the presence or absence of code that could lead to a vulnerability. Depending on a contract's functionality, we focused on different areas.

Meetings

Throughout the audit, we kept a tight feedback loop with the Tinyman team to clarify open questions and discuss any findings.

Findings

We first start with general findings that apply to multiple or all contracts and later we discuss findings that are specific to an individual contract.

Note that we do not include findings in this report that other parties in this collaborative peer reviewing process have previously uncovered. We refer the reader to the respective audits to get a comprehensive picture of the auditing process.

We use the severity classification as defined by Trail of Bits in this report, which is given in the Appendix.

BLOCK SHAKE

TALGO Contract

F1. Extracting User Funds through Protocol Fee Rewards

Severity: High	Difficulty: High

The tALGO contract takes a configurable fee from the rewards and donations that the pool collects. Because of a missing input validation for the fee percentage, the fee can be set to more than 100% by the manager account.

Impact

Setting the protocol fee to more than 100% can lead to two unwanted results:

- The fee collector can claim more than the available rewards, effectively extracting users' funds.
- A high fee percentage can cause an integer overflow or underflow in an internal state that can temporarily block users from minting / burning tALGO until the fee percentage is reduced again to a valid value.

The fee is defined in the global state `protocol_fee` as a percentage, e.g., setting `protocol_fee = 20` means that 20 percent of the collected rewards are reserved for the protocol itself. The protocol fee can be set in the function `set_protocol_fee` and a change to the fee takes effect for newly accrued rewards. The `fee_amount` parameter is not validated and can therefore be set to any value between `0` and `UINT64_MAX`.

```
@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
```

The `protocol_fee` variable is used in function `update_rewards_and_rate`, where it is used to compute the protocol rewards. This function first computes the newly accrued rewards (variable `new_rewards`) through block proposals and ALGO donations since the last

time this function was called. In the following we highlight several cases how this function can be exploited:

- Assume `new_rewards = 200` and `protocol_fee = 150`, then the protocol claims a `200 * 150 / 100 = 300` ALGO reward, effectively taking 100 ALGO out of user funds.
- Assume `new_rewards > 1` and `protocol_fee = UINT64_MAX`, then the code will panic because of an integer overflow when computing `protocol_rewards`.
- Assume `algo_balance = 250`, `new_rewards = 200`, `protocol_fee = 150`, then we have that `protocol_rewards = 300`. In this case the rate computation underflows because of `algo_balance - protocol_rewards = 250 - 300`.

```
func update_rewards_and_rate(delta: int):
    ...

    int protocol_rewards = (new_rewards * app_global_get("protocol_fee")) / 100

    int rate = btoi((itob(algo_balance - protocol_rewards) b* itob(RATE_SCALER)) b/
itob(minted_talgo))

    ...

    int protocol_talgo = app_global_get("protocol_talgo")
    int new_protocol_talgo = calc_talgo(protocol_rewards)
    protocol_talgo = protocol_talgo + new_protocol_talgo
    app_global_put("protocol_talgo", protocol_talgo)

    ...
end
```

Recommendation

This finding can be easily addressed by validating the `protocol_fee` in `set_protocol_fee`.

We recommend that:

- The code checks that `protocol_fee <= 100` to ensure the security of the protocol and prevent this finding
- Ensure that `protocol_fee` has a reasonable upper-bound below 100 to give users confidence that rewards will always be shared in a fair manner.

Response

This issue was addressed in commit e47c5a28261ef2bdcf16e5f18bb0757a9b5961c4 by adding an assertion to `set_protocol_fee` that the protocol fee is at most 100.

F2. TALGO Donations

Severity: Informational

While the tALGO contract handles ALGO donations correctly, tALGO donations are ignored. ALGO donations are considered rewards much like rewards earned for block proposals and they are distributed to tALGO holders. On the other hand, tALGO donations are not accounted for by the smart contract.

Impact

Donated tALGO (and their equivalent ALGO) are locked up forever because the contract considers those tALGO to be still in circulation.

Recommendation

The contract could account for donated tALGO in function `update_rewards_and_rate` much like it does for ALGO donations. In particular, the contract could subtract the donated tALGO from the `minted_talgo` global state, which would mean that the ALGO/tALGO rate increases. This would benefit all other tALGO holders, because they can redeem their tALGO for more ALGO.

Response

This issue was addressed in commit 23c972dacb84e02c55ad7abf784e5d8de96992e9.

F3. Rounding Inaccuracies

Severity: Informational

TALGO are minted at a 1:1 exchange rate to ALGO only when the contract is first deployed and no rewards have accrued yet. As soon as rewards or donations start to flow in, the exchange rate increases and when users burn their tALGO they get more ALGO out as they put in initially. This is how users earn rewards.

Impact

Small rounding inaccuracies can occur in the `update_rewards_and_rate` function when the ALGO:tALGO exchange rate is computed. For example, we simulated the following scenario:

- User A adds 100 ALGO to the contract
- The contract receives 4 ALGO as reward or donation
- User B adds 25'000'000 ALGO to the contract
- User B removes his full stake and receives 24'999'999.999999 ALGO

As is evident from the example, the user receives one base unit less than what she added to the contract in the previous step.

Upon analyzing the contract in more detail, we observe that the rounding inaccuracies are always in favor of the contract, which means that the contract does not hand out more ALGO than expected. On the other hand, a user may receive a negligible amount less than expected.

Recommendation

While rounding inaccuracies cannot be completely eliminated, increasing the `RATE_SCALER` constant would reduce the problem. Given that the inaccuracies are negligible, we think the current setting is acceptable.

F4. LogicSig Usage for Keyreg Transaction

Severity: Informational

To change the online status of one of the managed accounts, a KeyReg transaction must be issued from that account. The tALGO contract implements that by rekeying that particular account to a LogicSig that carries out the necessary KeyReg transaction before rekeying it back to the tALGO contract. Rekeying to a LogicSig must be done carefully since this increases the attack surface. We can confirm that the code as written is secure and handles the rekeying flow correctly, but we think it is important to reflect on why this approach was chosen.

The simplest alternative to using a LogicSig in this use case is issuing an inner transaction from the tALGO contract that executes the necessary KeyReg transaction. This approach was considered by the Tinyman developers and dropped in favor of the current LogicSig

approach. Their argument was that if the KeyReg transaction changes with a future Algorand protocol upgrade, the code may be irreversibly broken, which means the node cannot be taken online anymore. There is precedent where the KeyReg transaction was updated in a backwards-incompatible way to include StateProofs and state proof keys in 2022. The chosen approach is safe as long as no “dangerous” fields (akin to close-to, etc.) are added to the KeyReg transaction. The chances of this happening, while slim, are non-zero, and could introduce an attack vector in the future. On the other hand, the approach that uses inner transactions is not at risk of such issues, but could require re-deploying the contract if the KeyReg transaction ever changes in a backwards-incompatible way.

Response

The Tinyman team has chosen to drop the KeyReg LogicSig in favor of a simpler approach that rekeys the account temporarily to the node manager who is authorized to perform this single KeyReg transaction on behalf of the contract account. This can be found in this commit: [3ac56b1913b528feb9021ec076c5d8116ad1ee94]

Observations

In the following we list a few observations that do not have any security implications.

O1. Checking MAX Account Balance

The contract defines a `MAX_ACCOUNT_BALANCE` (currently, 69M ALGO) that defines the amount of ALGO that the contract holds before stopping to mint new tALGO. The contract currently checks if the account’s balance is smaller than this maximum account balance, whereas it would be cleaner to check that the account balance is smaller than or equal.

```
@public()
func mint(algo_amount: int):
    assert(Gtxn[-1].Amount == algo_amount)
    assert(Gtxn[-1].Receiver == Global.CurrentApplicationAddress)
    assert(balance(app_global_get("account_0")) < MAX_ACCOUNT_BALANCE)
    ...
end
```

O2. More Explicit Argument Validation

The contract does not always validate parameters explicitly or as soon as possible. For example, the following code block shows an `ascii_digit` function that takes a parameter

`i`, which according to a note by the developers must be smaller than 10. However, this assertion is never explicitly done in the code.

```
# NOTE: i MUST be < 10
func ascii_digit(i: int) bytes:
    return extract(7, 1, itob(i + 48))
end
```

Another example can be seen in the `set_node_manager` function, which receives a `node_index` parameter. According to the system specifications, the node index must be between zero and four, but this is never explicitly checked. As a result, the manager account can set node managers for invalid indexes (invalid according to the specification).

```
@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
```

TALGO ReStaking Contract

F5. Manager can block contract

Severity: High	Difficulty: High

An integer overflow can be provoked in function `update_state_internal` if the reward rate is set to a high value. The reward rate (variable `reward_rate_per_time`) denotes the amount of TINY tokens that are distributed per second and if this value is greater than 18446744073 base units (that is, 18'446.744073 TINY), the multiplication with constant `RPU_SCALER = 1000000000`) overflows.

```
func update_state_internal(timestamp: int):
    ...
    int total_staked_amount = app_global_get(TOTAL_STAKED_AMOUNT_KEY)
    int reward_rate_per_time = get_reward_rate_per_time(timestamp)
    ...
    if total_staked_amount:
        ...
        # This would overflow if reward_rate_per_time > 18446744073 microunit.
        int reward_rate_per_unit_per_time = (reward_rate_per_time * RPU_SCALER) /
total_staked_amount
        ...
    end
    return
end
```

The reward rate can be configured by the manager account (and only the manager account) with the function `set_reward_rate`, shown in the next code block. While omitted here in this code snippet, the function only allows reward rate increases of 10% each time it is called. However, a (rogue) manager account can overcome this by simply calling the function multiple times.

```
@public()
func set_reward_rate(total_reward_amount: int, end_timestamp: int):
    assert(Txn.Sender == app_global_get(MANAGER_KEY))
    ...
    # Wrap up, accumulate for the last rate.
    update_state_internal(Global.LatestTimestamp)
    ...
    int duration = end_timestamp - Global.LatestTimestamp
    int reward_rate_per_time = total_reward_amount / duration
    ...
    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)
    ...
end
```

Notice that `set_reward_rate` calls `update_state_internal` *before* it computes & sets the reward rate. This means an overflow does not happen at the time the reward rate is configured (which would resolve this issue since the code would panic), but rather the overflow happens in a delayed fashion the next time this is called when it is too late.

Impact

This issue can be exploited to block the contract such that users can no longer increase their stake, withdraw their tALGO, claim their rewards, etc. Moreover, once triggered, this issue is irreversible, which means that all funds are lost forever.

The root cause of this issue is that most functions in the contract (`increase_stake`, `decrease_stake`, `claim_rewards`, etc.) call `update_state_internal` before it carries on with its task (e.g., withdrawing staked tALGO). Therefore, if the integer overflow happens, the contract panics and is aborted. In addition, also the reward rate can no longer be decreased to avoid the overflow, because function `set_reward_rate` also calls `update_state_interal` before actually setting the reward rate.

It is important to note that this attack can only be carried out by a (rogue) manager account, and therefore is difficult to execute in practice.

Recommendation

We recommend explicitly checking if the reward rate is within a reasonable upper-bound in `set_reward_rate`. If this upper-bound is bigger than what is currently safe from an integer overflow (circa 18k TINY/second), we recommend using wide-integer math to avoid this overflow.

Response

This issue was addressed in commit [18322b0bd720ad23c464de22c2f92c2aadff94ce] by adding an assertion to `set_reward_rate`.

F6. Promised Reward Rate may not be backed by Available Funds

Severity: Medium	Difficulty: High

The manager account can set an arbitrary reward rate, which may not be backed by actually available TINY tokens. In function `set_reward_rate` the manager specifies the `total_reward_amount` that is distributed as rewards over a period of time (see the code block above). It is not checked that the contract actually has the specified amount of TINY tokens.

Impact

It can happen that a user accrues rewards over a period of time but then cannot claim them because the contract is not funded with sufficient TINY tokens.

By setting the reward rate the manager can control the annual percentage rate (APR). Therefore, a malicious manager can artificially increase the APR to attract users, but without providing sufficient funds to actually reward the users.

Assume the contract is funded with plenty of TINY tokens that an attacker wants to steal. If the attacker manages to get hold of the manager account, then the attacker could (a) increase her stake, (b) set the reward rate temporarily to a very high amount, and (c) claim a high amount of rewards in a very short period of time. In this same period of time, normal users also accrue a large reward, but assuming that they want to claim those after the attacker has claimed her reward, the contract may have insufficient TINY tokens to spend.

Recommendation

We recommend that the contract checks in `set_reward_rate` if the specified `total_reward_amount` is less than or equal to the available TINY balance to spend. Here it is important to take into consideration also TINY tokens that, while not yet claimed, have already been earned by users.

Response

This was addressed in commit 5566419edb88a0bc5c0e2dfb4d690731e479cf7c. The contract now checks that it has sufficient TINY tokens to reward users.

F7. TINY Power

Severity: Low	Difficulty: Low

A user needs a certain amount of TINY power to be allowed to stake tALGO, where TINY power is the voting power that a user gets from staking TINY tokens in Tinyman's governance system. Currently, the contract only checks the user's TINY power if the user has not staked any tALGO at the present moment. If a user retains a minimal amount of stake in the restaking contract (one base unit is sufficient), the user can increase and decrease her stake at will without having to have sufficient TINY power.

```
@public()
func increase_stake(amount: int):
    ...
    box<UserState> user_state = OpenOrCreateBox(Txn.Sender)
    ...
    if !user_state.staked_amount:
        int current_tiny_power = get_account_voting_power(Txn.Sender)
        assert(current_tiny_power >= app_global_get(TINY_POWER_THRESHOLD_KEY))
        ...
    end
    user_state.staked_amount = user_state.staked_amount + amount
    ...
end
```

Impact

A user can game the system by entering the tALGO staking pool when she has sufficient TINY power and then freely enter/leave the pool without having to maintain any TINY power. Also, the user would be immune from future changes of the `TINY_POWER_THRESHOLD` by the manager.

Recommendation

Instead of checking the user's TINY power only when the user enters the staking pool for the first time (or rather when they have zero tALGO staked), the contract could check the TINY power every time the user wants to increase her stake.

Response

This was addressed in commit [ca4c6d7b1bd83abeff15237879ffb8f904446c4d] by checking that the user still has sufficient TINY power by the time she claims her rewards.

Observations

In the following we list a few observations that do not have any security implications.

O3. Unused Variables & Code

There are several instances in the code where variables are declared but never used. For example, in the `apply_rate_change` function the variable `last_update_timestamp` is read from global storage, but never used. Then, there is an `if/else` branch where the `if` branch writes to a variable whose value is never read again.

```
@public()
func apply_rate_change():
    int current_reward_rate_per_time = app_global_get(CURRENT_REWARD_RATE_PER_TIME_KEY)
    int current_reward_rate_per_time_end_timestamp =
app_global_get(CURRENT_REWARD_RATE_PER_TIME_END_TIMESTAMP_KEY)
    int last_update_timestamp = app_global_get(LAST_UPDATE_TIMESTAMP_KEY)

    if Global.LatestTimestamp <= current_reward_rate_per_time_end_timestamp:
        # Do nothing. CURRENT_REWARD_RATE_PER_TIME is valid.
        int reward_rate_per_time = current_reward_rate_per_time
    else:
        ...
        int reward_rate_per_time = 0
        ...
        log(ARC28Event("apply_rate_change(uint64)", itob(reward_rate_per_time)))
    end
end
```

O4. Unused Timestamp in UserState

The restaking contract stores the user's state (amount staked, etc.) in box storage. The data is modeled as a `UserSate` struct that also includes a `timestamp` variable, which is written to in the contract, but that is never read. This may be useful information for third-party applications (e.g., wallets) since box storage can be accessed off-chain. If this is the intended use case, it should be documented or else the variable should be dropped.

```
struct UserState:
    staked_amount: int
    accumulated_rewards_per_unit_at_last_update: int
    accumulated_rewards: int
    timestamp: int
end
```

Appendix

The findings reference a severity and difficulty classification which has been defined by Trail of Bits.

BLOCK SHAKE

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.