BAIL security

ALTO
Staking Rewads

FINAL REPORT

October ‘2025

Disclaimer:

Security assessment projects are time-boxed and often reliant on information that may be provided by a client, its affiliates, or its partners. As a result, the findings documented in this report should not be considered a comprehensive list of security issues, flaws, or defects in the target system or codebase.

The content of this assessment is not an investment. The information provided in this report is for general informational purposes only and is not intended as investment, legal, financial, regulatory, or tax advice. The report is based on a limited review of the materials and documentation provided at the time of the audit, and the audit results may not be complete or identify all possible vulnerabilities or issues. The audit is provided on an "as-is," "where-is," and "as-available" basis, and the use of blockchain technology is subject to unknown risks and flaws.

The audit does not constitute an endorsement of any particular project or team, and we make no warranties, expressed or implied, regarding the accuracy, reliability, completeness, or availability of the report, its content, or any associated services or products. We disclaim all warranties, including the implied warranties of merchantability, fitness for a particular purpose, and non-infringement.

We assume no responsibility for any product or service advertised or offered by a third party through the report, any open-source or third-party software, code, libraries, materials, or information linked to, called by, referenced by, or accessible through the report, its content, and the related services and products. We will not be liable for any loss or damages incurred as a result of the use or reliance on the audit report or the smart contract.

The contract owner is responsible for making their own decisions based on the audit report and should seek additional professional advice if needed. The audit firm or individual assumes no liability for any loss or damages incurred as a result of the use or reliance on the audit report or the smart contract. The contract owner agrees to indemnify and hold harmless the audit firm or individual from any and all claims, damages, expenses, or liabilities arising from the use or reliance on the audit report or the smart contract.

By engaging in a smart contract audit, the contract owner acknowledges and agrees to the terms of this disclaimer.

1. Project Details

Important:
Please ensure that the deployed contract matches the source-code of the last commit hash.

Project	ALTO – Staking Rewards - Audit Report
Website	altofoundation.org
Language	Solidity
Methods	Manual Analysis
Github repository	https://github.com/altomoney/v1/tree/282aee08b4a53f753fa11e05e5139b27dcefe272
Resolution 1	https://github.com/altomoney/v1/tree/3d820be8ccb76d59fe8d015ca7ea2608a2f4af7f

2. Detection Overview

Severity	Found	Resolved	Partially Resolved	Acknowledged (no changes made)	Failed resolution	Open
High						
Medium	1		1			
Low	1			1		
Informational	17	2		14		1
Governance	2			2		
Total	21	2	1	17		1

2.1 Detection Definitions

Severity	Description
High	The problem poses a significant threat to the confidentiality of a considerable number of users' sensitive data. It also has the potential to cause severe damage to the client's reputation or result in substantial financial losses for both the client and the affected users.
Medium	While medium level vulnerabilities may not be easy to exploit, they can still have a major impact on the execution of a smart contract. For instance, they may allow public access to critical functions, which could lead to serious consequences.
Low	Poses a very low-level risk to the project or users. Nevertheless the issue should be fixed immediately
Informational	Effects are small and do not post an immediate danger to the project or users
Governance	Governance privileges which can directly result in a loss of funds or other potential undesired behavior

3. Detection

Staking

The scope of the Staking architecture audit includes an audit of the following functions from the AltoAdapter:

- **stakingLock**: This function allows locking funds on behalf of the initiator. It is expected that the initiator will always be the msg.sender and no initiator spoofing is ever possible. If there is ever any spoofing possibility, an attacker can continuously increase the lock duration for a user by relocking. The LOCK_TOKEN must sit in the balance of the AltoAdapter contract and since the lock function transfers funds solely from msg.sender (which is AltoAdapter in that context), there is no approval misuse risk for this function.

- **stakingUnlock**: This function allows for unlocking funds on behalf of the initiator. It is expected that the initiator will always be the msg.sender and no initiator spoofing is ever possible. If there is ever any spoofing possibility, that means funds from any address which granted authorization to the router can be stolen.

Note: The initiator() logic and all other core components of the AltoAdapter are not part of this scope.

AltoStaking

The AltoStaking contract allows users to lock their LOCK_TOKEN for a specific epoch duration into different containers. Each container has its own multiplier and decay/growth rates which determine currentDurationEpoch whenever an epoch is processed. It is also possible to reuse an already existing lock and simply extend it - without transferring additional LOCK_TOKEN to the contract.

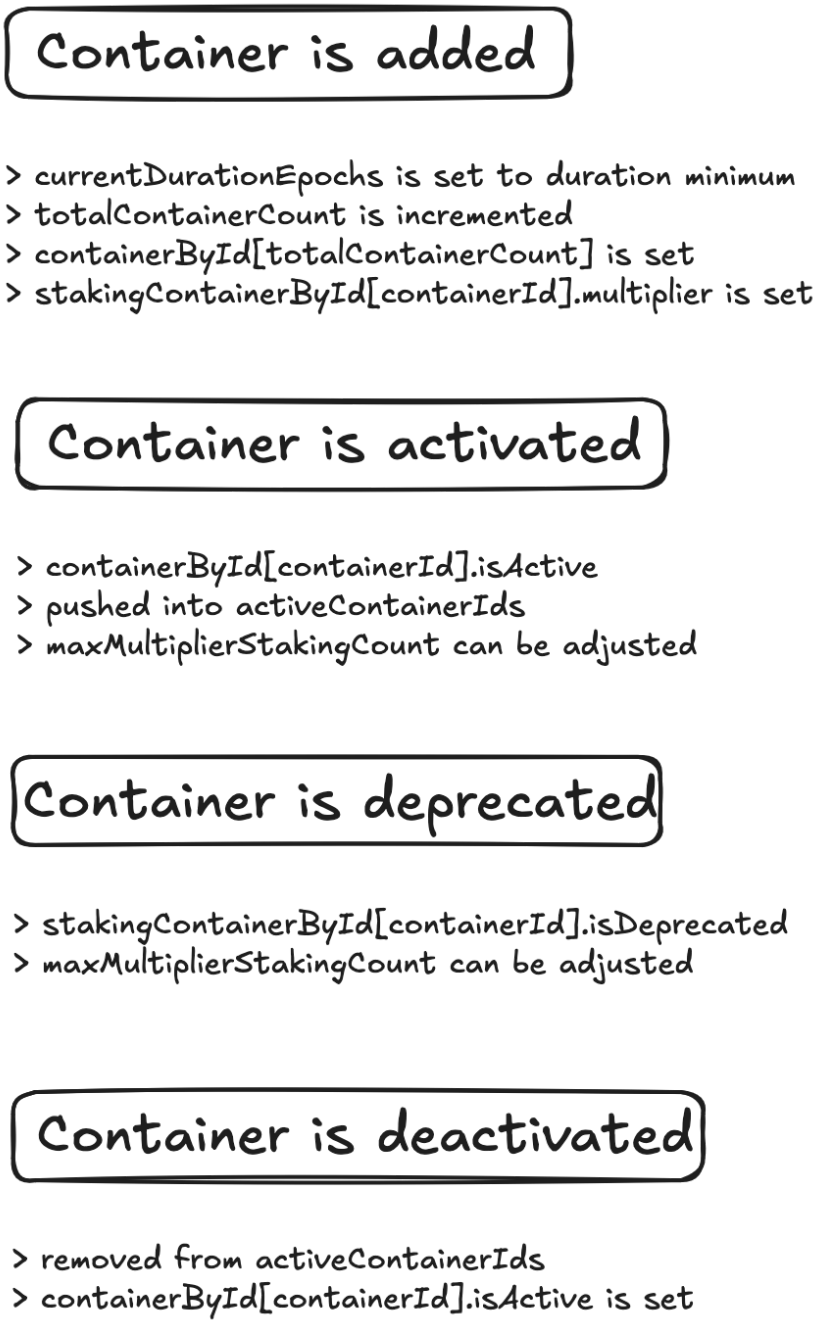
Users can unlock their LOCK_TOKEN once the position is considered as expired.

The purpose of locking tokens is an increase of voting power which is reflected via the getVotes functions.

Containers can have different states, these will be elaborated in the corresponding appendix section.

Appendix: Container States

A container can be in multiple different states, which has an impact on various functionalities such as the voting power determination or lock abilities. Usually, the lifecycle of a container looks as follows:



Appendix: Epoch Mechanics

The contract works on an epoch-basis which is similar to many other contracts, most prominently the VE implementation. Basically, each epoch is EPOCH_DURATION long and once the EPOCH_DURATION has surpassed, a new epoch starts. This is achieved via the standard truncation mechanism:

> (timestamp - initializedAtTimestamp) / EPOCH_DURATION

Notably, the initializedAtTimestamp variable accurately reflects the start of a new epoch, which allows governance to define the exact epoch start.

Appendix: Pausing Mechanics

The contract inherits the Pausable contract (**out-of-scope**), which allows determined addresses to move the contract into the RESCUE_TYPE or PAUSE_TYPE mode. If the contract has been moved once into the RESCUE_TYPE, users have the ability to withdraw their locks without waiting for expiry. However, the contract is basically permanently marked as non-functional.

Appendix: Voting Power Determination

The voting power determination is a very straightforward mechanism as it simply multiplies the locked amount with a container's multiplier and returns this as voting power. Only active containers can count into this.

Privileged Functions

- setPauser
- emergencySweep
- addContainer
- updateContainer
- activateContainer
- deactivateContainer
- deprecateContainer
- startEpoch

Core Invariants:

INV 1: Users cannot lock for more than currentEpochDuration

INV 2: During _updateMaxMultiplierStakingContainerID, it must always set maxMultiplierContainerStakindID to containerID with highest multiplier

INV 3: Multiplier of a containerId must never be larger than 1e18

INV 4: During lock, containerById[containerId].currentDurationEpochs must never be larger than maxEpochsToLockFor

INV 5: During lock, expiryTimestamp must always be set to corresponding containerById[containerId].currentDurationEpochs

INV 6: unlock can be executed even if position is not expired - in rescue mode

INV 7: lock must automatically advance the epoch up to nextEpochToProcess

INV 8: lock is not allowed to a deprecated containerID

INV 9: Whenever a lock happens, currentDurationEpochs must always be up-to-date

INV 10: initializedAtTimestamp is used as a source of truth for when an epoch is switched

INV 11: Whenever startEpoch is called, the epoch starts with EPOCH_DURATION

INV 12: Once RESCUE_TYPE is set, can never unpause the contract

Issue_01	Governance Issue: General Privileges
Severity	**Governance**
Description	Currently, governance of this contract has several privileges for invoking certain functions that can drastically alter the contracts behavior. - Sweeping of funds - Pausing unlocks - Changing of multipliers Furthermore, governance can: - deprecate a non-active container - deactivate a deprecated container that still contributes to future TVL
Recommendations	Consider incorporating a Gnosis Multisignature contract as owner and ensuring that the Gnosis participants are trusted entities.
Comments / Resolution	Acknowledged.

Issue_02	Hindsight impact of updateContainer
Severity	**Medium**
Description	The updateContainer function can be called while not all epochs are advanced. This can have a hindsight impact on the new duration determination for a container.
Recommendations	Consider implementing a boolean flag which advances epochs before any configuration change if set to true.
Comments / Resolution	Partially resolved, it is now enforced that such an update is executed. However, contrary to our recommendation it is not optional - this can result in a DoS of updateContainer in unexpected scenarios where such an advancement would revert.

Issue_03	isReuse feature or top-up deposit can be used to decrease duration
Severity	**Low**
Description	It is possible that the duration for a container is decreased, even so far that the decrease might result in an actually lower period than a user has as leftover period. This is specifically true if decayRate.max > 1. In such a situation, users can simply use the isReuse feature or top up their deposit with 1 wei, resetting their expiryTimestamp to the potentially smaller one.
Recommendations	Consider either acknowledging this issue or ensuring that decayRate.max stays at 1.
Comments / Resolution	Acknowledged. The client confirmed this is a design choice.

Issue_04	Inconsistency between TVL and aggregated deposits due to expiryTimestamp not being truncated to epoch boundaries
Severity	**Informational**
Description	The staking system schedules TVL per epoch to drive container duration adjustments. Lock schedules are recorded per epoch using a half-open range [fromEpoch, expiryEpoch). Unlockability is time-based: a user may withdraw once block.timestamp ≥ expiryTimestamp, where expiryTimestamp is computed from the lock timestamp plus an integer number of epochs. Note that expiryTimestamp is not clamped to epoch boundaries, while TVL accounting is epoch-granular. A locked position remains valid (and contributes votes) into its expiry epoch until its exact expiryTimestamp, but its amount is not counted in that epoch's TVL because the schedule excludes the expiry epoch by design. This creates a deterministic vote/TVL divergence window (up to almost one full epoch). Root-cause: A mismatch between: • Time-continuous lock expiry (expiryTimestamp not snapped to epoch boundaries), and • Epoch-discrete TVL accounting (half-open [fromEpoch, expiryEpoch) that omits the expiry epoch entirely). As a result, any per-epoch TVL snapshot under-represents "currently locked" amounts during the portions of the expiry epoch prior to each position's expiryTimestamp.
Recommendations	As discussed with the team during the audit, this is not considered as an issue. Therefore, we recommend acknowledging it.

Comments / Resolution	Acknowledged, as discussed this is not an issue. This only serves as an informational note.

Issue_05	Order dependency violation of maxMultiplierStakingContainerId due to swap and pop implementation
Severity	**Informational**
Description	The determination process of the maxMultiplierStakingContainerId is order relevant, as it loops over all active containers. If swap and pop is executed during an active container removal, it can happen that now an ID is pushed to an earlier element in the loop while at the same time the maxMultiplierStakingContainrId still points to the later element (which was previously before).
Recommendations	Consider acknowledging this issue
Comments / Resolution	Acknowledged.

Issue_06	Truncation of tvlSharePercentage is in favor of the user
Severity	**Informational**
Description	The tvlSharePercentage is used to calculate the new duration. The higher this value, the higher the duration (simplified). Therefore, the truncation of this value results in a potentially shorter duration.
Recommendations	Consider acknowledging this issue as the impact is minimal. We do not see it necessary to round up.
Comments / Resolution	Acknowledged.

Issue_07	Locking possibility to non-activated containers
Severity	**Informational**
Description	Currently, it is possible to lock tokens to containers that are not deprecated. There is however no such check that a container is indeed active. This means that users may lock tokens without receiving actual voting power.
Recommendations	Consider either adding such a check or acknowledging this issue.
Comments / Resolution	

Issue_08	Inconsistent behavior for maxMultiplierStakingContainerId and voting power
Severity	**Informational**
Description	Whenever a container is deprecated, its impact is removed from the maxMultiplierStakingContainerId. However, the multiplier is indeed still used for the getVotes function since it is still marked as active. This behavior can be considered as a minor inconsistency.
Recommendations	Consider elaborating whether this inconsistency exposes any issues.
Comments / Resolution	Acknowledged, the team added the following comment: *"This is intended behavior because:* *maxMultiplierStakingContainerId determines where to auto-lock rewards (e.g., in AltoRewardsDistributor). New deposits should not go into a deprecated container.* *getVotes returns voting power for existing positions. Users who locked before deprecation should retain their voting power until their lock expires.* *This is not a valid issue."* ########## Since it is explicitly desired that voting power should remain until the lock expires, this means voting power will remain even if a container is deprecated. We need to note that this means that users who have locked in a now deprecated container can have an advantage versus other users who would like to lock in such a container but now are prohibited from doing so because it is deprecated: *if (stakingContainerById[containerId].isDeprecated) {* *revert AltoStakingContainerDeprecated();*

	} This behavior should be elaborated, whether it is acceptable or not. It is very likely how the design should be but we consider it as important to point out, in the unlikely scenario where this would be contrary to the design idea. If this is desired, no further actions/comments are required.

Issue_09	Incorrect view-only behavior of getContainerAfterEpochProcessing
Severity	**Informational**
Description	The getContainerAfterEpochProcessing function allows to determine nextDurationEpochs for future epochs. This behavior is incorrect as it does not "compound" each epoch up to the desired epoch, instead, it simply jumps to the desired future epoch. Furthermore, even if it would be compounding, it is still only a snapshot value which can deviate from the real value once that epoch reaches, due to the fact that there may be further locks.
Recommendations	This is a simple limitation of the view-only function and should be acknowledged.
Comments / Resolution	Acknowledged.

Issue_10	New containers will be immediately initialized with minimum duration
Severity	**Informational**
Description	Whenever a new container is being added, its duration will be set to the minimum duration. While this is in favor of the user and not the protocol, we consider this as a design decision.
Recommendations	Consider acknowledging this.
Comments / Resolution	Acknowledged. Expected behavior, as mentioned earlierthat we consider this as a design decision.

TVLWeightedContainers

The TVLWeightedContainers contract is inherited by the AltoStaking contract and handles all internal functions for container TVL handling, epoch processing, container state transitions.

Appendix: Next Duration Epoch Determination

Whenever an epoch is processed, it will process all active containers for this epoch. Processing a container means the re-determination of currentDurationEpochs which basically determines how many epochs a new stake is locked.

A specific mathematical algorithm was implemented for that purpose.

Each epoch, it nudges a container's lock duration toward a target duration derived from its relative TVL share that epoch. The nudge size (how many epochs to move) is rate-limited by growthRate/decayRate, and those rates are themselves scaled by how far the current value is from the target, normalized by maxRateDifferenceEpochs.

Below we will explain the algorithm step by step:

Base:

a) Calculate containerTVLSharePercentage
   i) containerTVLSharePercentage = epochTVL[epoch][container] * 1e18 / totalTVL[epoch]
b) Calculate average duration based on share
   i) newRequestedDurationEpochs = ([maxDuration - minDuration] * containerShareTVLPercentage / 1e18) + minDuration

Scenario 1: newRequestedDurationEpochs is larger than currentDurationEpochs

a) Calculate the difference b/w [newRequestedDurationEpochs - currentDurationEpochs]
   i) difference = newRequestedDurationEpochs - currentDurationEpochs
b) Calculate difference in % to maxRateDifferenceEpochs
   i) differenceFromMaxRatePercentage = difference * 1e18 / maxRateDifferenceEpochs
c) Difference below 1e18: Calculate epochsToGrow as relation from difference to (growthRateMax - growthRateMin)
   i) epochsToGrow = ([growthRateMax - growthRateMin] * differenceFromMaxRatePercentage / 1e18) + growthRateMin
d) Difference above/equal 1e18: use growthRateMax
   i) epochsToGrow = growthRateMax

e) Set currentDurationEpochsNext
    i) min(currentDurationEpochsNext = currentDurationEpochs + epochToGrow, durationMax)

Scenario 2: newRequestedDurationEpochs is smaller than currentDurationEpochs

a) Calculate the difference b/w [currentDurationEpochs - newRequestedDurationEpochs]
    i) difference = currentDurationEpochs - newRequestedDurationEpochs
b) Calculate difference in % to maxRateDifferenceEpochs
    i) differenceFromMaxRatePercentage = difference * 1e18 / maxRateDifferenceEpochs
c) Difference below 1e18: Calculate epochsToDecay as relation from difference to (decayRateMax - decayRateMin)
    i) epochsToGrow = ((growthRateMax - growthRateMin) * differenceFromMaxRatePercentage / 1e18) + growthRateMin
d) Difference above/equal 1e18: use decayRateMax
    i) epochsToGrow = growthRateMax
e) min(currentDurationEpochsNext = currentDurationEpochs - epochsToDecay, durationMin)

Appendix: Container TVL Update

The container TVL is a fundamental instrument for the duration determination. Whenever a lock is executed, it will increase the TVL for the current epoch up to the expiration epoch.

A notable edge-case is whenever a lock is extended/increased, as it will then first remove the TVL impact from the corresponding lock and then re-assigns it. This is to make sure no double addition occurs.

Privileged Functions

- none

Core Invariants:

INV 1: _startEpoch cannot be called twice

INV 2: nextEpochToProcess must be increased during _advanceEpoch

INV 1: epochsToGrow must be rounded up

INV 3: epochsToDecay must be rounded down

INV 4: currentEpochDurationsNext must never become larger than maxDuration or smaller than minDuration

INV 5: Whenever newRequestedDurationEpochs > currentDurationEpochs, the minimum increase must be growthRate.min

INV 6: Whenever newRequestedDurationEpochs < currentDurationEpochs, the minimum decrease must be decayRate.min

INV 7: Whenever differenceFromMaxRatePercentage > MATH_PRECISION, set epochsToGrow/epochsToDecay to growthRate.max / decayRate.min

INV 8: Only deprecated containers can be deactivated

Since codebases are heavily interconnected, we decided to add all issues within the AltoStaking section.

Rewards

Security Advise:

- oracle correctness

The scope of the Rewards architecture audit includes an audit of the following functions from the AltoAdapter:

- **rewardsDistributorClaimRewardTokens**: This function allows a user to interact with the AltoRewardsDistributor.claimRewardToken function, by specifying a payAmount, triggering the reward purchase and refunding and unconsumed payAmount. It is expected that the initiator will always be the msg.sender and no initiator spoofing is ever possible.

AltoRewardsDistributor

The AltoRewardsDistributor contract is a sale contract which allows users to purchase the rewardToken for a corresponding paymentToken amount. Users with a valid merkle proof and corresponding leaf can call the distributeRewardsAmount function which calculates the required paymentToken amount and transfers it in will then distribute rewards to the caller either via direct transfer (liquid solution) or via a lock into the AltoStaking contract (lock solution). In the case of the lock solution, a lockDiscount is applied which simply reduces the required amount of paymentToken.

This contract also interacts with the AltoReferralWhitelistAdapter and even allows the incorporation for up to 10 adapters. Users with valid referrers will further benefit from referral discounts while referrers will receive a share of the corresponding paymentToken amount.

Appendix: Discount Mechanism

The discount consists of the potential discount for the lock solution (lockDiscount) and the aggregated discount for up to 10 referrer discounts. It is simply applied on the final paymentToken amount as follows:

> finalAmount = paymentTokenRequired * (1e18 - discount) / 1e18

These two discounts are on top of the base discount which is defined by the leaf data.

Appendix: Lock Mechanism

Users can decide whether to directly claim their tokens or to lock them into the container with the highest multiplier on the AltoStaking contract. In the lock scenario, an additional lockDiscount will be applied which decreases the required paymentToken amount, incentivizing users to rather lock their tokens than claiming them as liquid tokens.

Appendix: Payment Amount Calculation

The paymentAmount calculation is a simple conversion from the desired rewardToken amount to the corresponding paymentToken amount, using a two-step conversion with additional discount application:

a) convert rewardAmount to USD (8 decimals)

i) rewardInUSD = rewardAmount * rewardPrice / rewardDecimals

b) convert rewardInUSD to paymentAmount
   i) paymentAmount = rewardInUSD * paymentDecimals / paymentPrice

c) Apply discount
   i) paymentAmount * (1e18 - discount) / 1e18

Privileged Functions

- setPaymentToken
- setRewardTokenUsdOracle
- setLockBonusDiscount
- setLockContract
- withdrawContractTokens
- whitelistAdapter
- revokeAdapter

Core Invariants:

INV 1: Rewards are not claimable if expirationTimestamp has passed

INV 2: Users cannot claim more than totalRewardsForUser

INV 3: In case of claimData.expirationTimestamp = 0, there is no expiration for a claim

INV 4: For locking tokens, onBehalf must have approved the AltoRewardsDistributor as operator

INV 5: For locking tokens, an additional lockBonusDiscount is applied

Issue_11	Oracle change can result in more consumed tokens than expected
Severity	**Governance**
Description	Currently, governance of this contract has several privileges for invoking certain functions that can drastically alter the contracts behavior. The oracle can be changed which results in more consumed paymentToken than expected. Similarly, it is possible to deactivate token purchases by adding a non-compliant adapter.
Recommendations	Consider incorporating a Gnosis Multisignature contract as owner and ensuring that the Gnosis participants are trusted entities.
Comments / Resolution	Acknowledged.

Issue_12	Race condition due to different settings
Severity	**Informational**
Description	Functions such as removeAdapter/setLockBonusDiscount/setPaymentToken/setRewardTokenUSDOracle are not only problematic in case of compromised governance but also in the scenario where a user has just submitted a transaction and now any value is changed before execution, essentially resulting in receiving less rewardToken than expected. This issue has only been rated as informational because it is expected that the AltoAdapter serves as router which implements such a slippage idea by allowing the caller to specify a paymentAmount.
Recommendations	Consider communicating a change in these variables with the community.

Comments / Resolution	Acknowledged.

Issue_13	Lack of slippage check can result in higher paymentToken amount
Severity	**Informational**
Description	Currently it is possible that the price for rewardToken increases before the actual execution of a claimRewardTokens call which can result in more provided paymentToken than expected. This issue has only been rated as informational because it is expected that the AltoAdapter serves as router which implements such a slippage idea by allowing the caller to specify a paymentAmount. A similar issue can occur if an expected referrer is not whitelisted anymore, as this can result in a discount decrease and therefore a higher paymentToken amount.
Recommendations	Consider implementing the AltoAdapter for frontend interactions. Users that interact directly with the smart contract are doing this at their own risk.
Comments / Resolution	Resolved, the AltoAdapter will be solely used for that purpose and users that interact directly with this contract do this at their own risk.

Issue_14	Potentially unexpected lock outcome in case of two containers with same multiplier
Severity	**Informational**
Description	The maxMultiplierStakingContainerId determination process is order dependent which means that if there are two containers with the same multiplier, the ID of the first container will be assigned to maxMultiplierStakingContainerId. In a real-world scenario, it can happen that these containers have different durations which means that users might experience an unexpected outcome.
Recommendations	Consider documenting this behavior.
Comments / Resolution	Acknowledged, this will be documented.

Issue_15	Incorrect comment about 100% discount
Severity	**Informational**
Description	The following section highlights a potential 100% discount: *// We allow zero paymentAmount only in case discount is 100%* *return paymentAmount;* This is incorrect, as the maximum discount is clamped to 0.5 (50%)
Recommendations	Consider removing the inaccurate comment.
Comments / Resolution	Resolved.

Issue_16	Blacklist concerns for paymentToken
Severity	**Informational**
Description	If a referrer is blacklisted for receiving the paymentToken, it happens that the whole claimRewardTokens function reverts, essentially not allowing a user to proceed.
Recommendations	Consider keeping this in mind and if such a situation ever occurs in a very rare circumstance, governance can always intervene by disabling a blacklisted referrer. We don't recommend a pull over push pattern as this will certainly alter the contract behavior and increase complexity for no reason. The current mechanism is safe and this is only considered as an informational note.
Comments / Resolution	Acknowledged, this will be documented.

Issue_17	Lack of transfer-tax token support
Severity	**Informational**
Description	This contract is not compatible with transfer-tax tokens as paymentToken. If these token types are used for any purpose within the contract, this will result in down-stream issues and inherently break the accounting. The rewardToken can be a transfer-tax token only in case of liquid transfers but the recipient will receive less than expected.
Recommendations	Consider not adding these tokens as paymentToken.
Comments / Resolution	Acknowledged, such tokens will not be added.

Issue_18	Potential revert for tokens that do not support zero-value transfers
Severity	**Informational**
Description	There are certain tokens that revert on zero-value transfers. This can happen due to truncation in the following operation: > *paymentTokenShare.multiplyWithPrecision(paymentAmount)* This would result in a revert of the claimRewardTokens function.
Recommendations	We do not consider this as an issue because the likelihood that such a token is used is very low while at the same time the truncation remains just for a 1 wei amount of paymentTokenShare.
Comments / Resolution	Acknowledged, such tokens will not be used.

AltoReferralWhitelistAdapter

The AltoReferralWhitelistAdapter contract is a simple referral storage contract where users can add whitelisted addresses as their referral. This contract is used by the AltoRewardsDistributor contract to determine the potential discount which is applied towards a rewardToken purchase and the paymentTokenShare which reflects the part of paymentToken that is relayed to the referrer and thus effectively deducted from the contract's received amount of the paymentToken.

Appendix: Whitelist Mechanism

Whitelisting can either happen via the governance-callable setReferralWhitelist function or via the user-callable activateReferralWhitelist function which validates the msg.sender using the merkle proof mechanism.

Once an address is whitelisted, it can be used as a referrer.

Appendix: Referral Mechanism

Once an address is whitelisted, it can be used as a referrer for other addresses. This can either happen via the governance-callable setReferral function or the user-callable activateReferral function.

Privileged Functions

- setBonusDiscount
- setPaymentTokenShare
- setReferral
- setReferralWhitelist

Core Invariants:

INV 1: Only whitelisted addresses can be set as referrers

INV 2: Referring to the own address is not allowed

INV 3: Reset of a user's referral is only allowed if old referrer is not whitelisted

Issue_19	Referrals are vulnerable to sybil-attacks
Severity	**Informational**
Description	As with all referral-related systems, it is always possible to set a self owned (different) address as referrer. This essentially allows to relay a part of the referral fee back to the initial buyer.
Recommendations	Consider accepting this design limitation.
Comments / Resolution	Acknowledged, the team is aware of this design limitation.

Issue_20	Lack of replay protection in activateReferral function allows a removed WL address to re-add itself to whitelist
Severity	**Informational**
Description	The activateReferral function reverts if an address is already whitelisted. On the first view, this seems to be sufficient replay protection. However, a realistic scenario is that governance removes an address from the whitelist if that address should not engage anymore in referral interactions. Such an address can simply call the activateReferral function and set itself back to being a whitelisted address, if the same merkleRoot is still used.
Recommendations	Consider only allowing the activateReferral function to be called once.
Comments / Resolution	Acknowledged, a specific flow will be used to prevent this issue from ever happening.

MerkleRootManager

The MerkleRootManager contract is inherited by the AltoRewardsDistributor contract and by the AltoReferralWhitelistAdapter contract. For the RewardsDistributor, it manages the proof validation when users attempt to claim rewards and for the ReferralWhitelistAdapter it ensures that an address can join the whitelist only if a valid proof is provided.

A two-step root setting mechanism has been implemented which is described in the appendix below.

Appendix: Two-Step Root Setting

The contract implements a two-step root setting mechanism which requires the submission of a pendingMerkleRoot first, followed by either a governance call to acceptPendingRoot or a permissionless call to the same function after the merkleRootTimelock period has passed. It is furthermore also possible for governance to remove a pending root before it has been accepted.

Privileged Functions

- submitNewPendingRoot
- revokePendingRoot
- setMerkleRootTimelock
- acceptPendingRoot

Core Invariants:

INV 1: Governance can immediately accept a pending root

INV 2: Any non-privileged address can accept a pending root only if validFrom has passed

Issue_21	merkleRoot cannot be reset to address(0)
Severity	**Informational**
Description	Currently, the setting of pendingMerkleRoot to address(0) is prohibited. Such a setting might be valid in case the contract should be disabled for further verification processes.
Recommendations	While it is possible to set a dummy merkleRoot to prevent further verifications, the idea of setting it to address(0) might still be relevant.
Comments / Resolution	Acknowledged.