Audit Report

Astroport Builder Unlock Contract

v1.0

December 7, 2022

Table of Contents

License

Disclaimer

This audit has been performed by

Oak Security

https://oaksecurity.io/
info@oaksecurity.io

Introduction

Purpose of This Report

Oak Security has been engaged by Delphi Labs Ltd. to perform a security audit of Astroport Builder Unlock Contract.

The objectives of the audit are as follows:

1. Determine the correct functioning of the protocol, in accordance with the project specification.
2. Determine possible vulnerabilities, which could be exploited by an attacker.
3. Determine smart contract bugs, which might lead to unexpected behavior.
4. Analyze whether best practices have been applied during development.
5. Make recommendations to improve code safety and readability.

This report represents a summary of the findings.

As with any code audit, there is a limit to which vulnerabilities can be found, and unexpected execution paths may still be possible. The author of this report does not guarantee complete coverage (see disclaimer).

Codebase Submitted for the Audit

The audit has been performed on the following GitHub repository:

https://github.com/astroport-fi/astroport-governance

Commit hash: `9f698cd1e0ad40f65e5a07584e6402d99158d4d1`

Methodology

The audit has been performed in the following steps:

1. Gaining an understanding of the code base’s intended purpose by reading the available documentation.
2. Automated source code and dependency analysis.
3. Manual line by line analysis of the source code for security vulnerabilities and use of best practice guidelines, including but not limited to:
    a. Race condition analysis
    b. Under-/overflow issues
    c. Key management vulnerabilities
4. Report preparation

Functionality Overview

The Astroport builder unlock contract implements unlock schedules of ASTRO for the original Astroport builders. It allows the contract owner to allocate tokens to the builder’s address given a start date, cliff, and duration and to adjust some of those values and it allows builders to withdraw tokens once unlocked and to transfer their allocation to a new recipient.

How to Read This Report

This report classifies the issues found into the following severity categories:

Severity	Description
Critical	A serious and exploitable vulnerability that can lead to loss of funds, unrecoverable locked funds, or catastrophic denial of service.
Major	A vulnerability or bug that can affect the correct functioning of the system, lead to incorrect states or denial of service.
Minor	A violation of common best practices or incorrect usage of primitives, which may not currently have a major impact on security, but may do so in the future or introduce inefficiencies.
Informational	Comments and recommendations of design decisions or potential optimizations, that are not relevant to security. Their application may improve aspects, such as user experience or readability, but is not strictly necessary. This category may also include opinionated recommendations that the project team might not share.

The status of an issue can be one of the following: **Pending, Acknowledged**, or **Resolved**.

Note that audits are an important step to improving the security of smart contracts and can find many issues. However, auditing complex codebases has its limits and a remaining risk is present (see disclaimer).

Users of the system should exercise caution. In order to help with the evaluation of the remaining risk, we provide a measure of the following key indicators: **code complexity**, **code readability**, **level of documentation**, and **test coverage**. We include a table with these criteria below.

Note that high complexity or low test coverage does not necessarily equate to a higher risk, although certain bugs are more easily detected in unit testing than in a security audit and vice versa.

Code Quality Criteria

The auditor team assesses the codebase’s code quality criteria as follows:

Criteria	Status	Comment
Code complexity	**Low-Medium**	-
Code readability and clarity	**Medium-High**	-
Level of documentation	**Medium-High**	-
Test coverage	**Medium**	cargo tarpaulin reports 45.06 % test coverage

Summary of Findings

No	Description	Severity	Status
1	Schedule cliff shifts the unlock start date rather than leads to unlocked tokens that cannot be claimed yet	Major	Resolved
2	Recipient transfer does not transfer unlocked amount checkpoint	Major	Resolved
3	Increasing the cliff may result in unlocked tokens becoming locked again	Major	Resolved
4	Inconsistency in validation during recipient transfer may lead to a loss of funds	Minor	Resolved
5	Maximum allocation amount can be set lower than total deposits	Minor	Resolved
6	Transfer of unallocated tokens leads to wrong token accounting	Minor	Resolved
7	Lack of validation when creating an allocation	Minor	Resolved
8	Unnecessary lowercasing of addresses is inefficient	Informational	Resolved
9	Unnecessary load of `PARAMS` during creation of an allocation is inefficient	Informational	Resolved
10	Unnecessary validation of `STATUS` during creation of an allocation is inefficient	Informational	Resolved
11	Unnecessary loading of `STATUS` during claim of new receiver is inefficient	Informational	Resolved
12	Misleading function name `compute_withdraw_amount` may negatively impact maintainability	Informational	Resolved
13	Inefficient validation of `prev_receiver`	Informational	Resolved
14	Lack of error message for query of non-existent allocation may confuse users	Informational	Acknowledged

Detailed Findings

1. Schedule cliff shifts the unlock start date rather than leads to unlocked tokens that cannot be claimed yet

Severity: Major

When defining an allocation of ASTRO for a user, the contract owner specifies a start date, a cliff, and a duration. However, when calculating the unlocked_amount in contracts/builder_unlock/src/contract.rs:939, the start time and cliff are added, effectively just shifting the start date. The consequence of this is that the unlock_amount does not increase linearly from the start time, but from the sum of start date and cliff instead.

This contradicts the comment in packages/astroport-governance/src/builder_unlock.rs:35 which states Time after the cliff during which the remaining tokens linearly unlock.

Recommendation

We recommend calculating the unlocked_amount using only the ratio of (timestamp - schedule.start_time) / duration, but returning zero until timestamp > schedule.start_time + schedule.cliff.

Status: Resolved

2. Recipient transfer does not transfer unlocked amount checkpoint

Severity: Major

During the execute_claim_receiver function in contracts/builder_unlock/src/contract.rs:650, only the astro_withdrawn field of the allocation status STATUS is transferred to the new receiver, but not any amount that was previously unlocked in unlocked_amount_checkpoint.

This implies that the new receiver may have to wait longer for funds to unlock in the case where the previous recipient had an allocation decrease and the unlocked funds were stored in unlocked_amount_checkpoint.

Recommendation

We recommend transferring the `unlocked_amount_checkpoint` from the previous to the new recipient.

Status: Resolved

3. Increasing the cliff may result in unlocked tokens becoming locked again

Severity: Major

In the `increase_cliffs` function, the contract owner can set a new cliff for an allocation in `contracts/builder_unlock/src/contract.rs:721`. This may result in already unlocked tokens no longer being unlocked.

Apart from this, an increasing cliff in the current implementation may lead to an underflow causing a panic in line `970` if a user already withdrew a bigger amount than the new unlocked amount. Note that this has no security implication, but a panic leads to a bad user experience.

Recommendation

We recommend either only allowing an extension of the cliff on allocations that have not yet started unlocking, or implementing a mechanism similar to the one used when decreasing an allocation, which sets `unlocked_amount_checkpoint`. If a mechanism similar to `unlocked_amount_checkpoint` is implemented, the `compute_unlocked_amount` function should be adjusted to always allow withdrawal of `compute_unlocked_amount`, even if the cliff has not been passed yet.

Status: Resolved

4. Inconsistency in validation during recipient transfer may lead to a loss of funds

Severity: Minor

In the `execute_propose_new_receiver` function, an error is returned in `contracts/builder_unlock/src/contract.rs:389` if the receiver has allocation parameters `PARAMS` with a non-zero `amount`. In the `execute_claim_receiver` function, however, an error is returned in `contracts/builder_unlock/src/contract.rs:630` if a `PARAMS` entry exists for the receiver, even if the amount is zero.

This inconsistency allows a recipient with `PARAMS` but a zero `amount` to be proposed as the receiver, but that new recipient can never claim that allocation. This might lead to a loss of funds for the new receiver if the previous receiver of the allocation does not cooperate to resolve the issue.

We classify this issue as minor since firstly, the creation of `PARAMS` allocations with a zero `amount` does not make much sense and may never exist, and secondly, the previous and new recipients are likely controlled by the same entity, in which case they can resolve the issue.

Recommendation

We recommend changing the condition in the `execute_propose_new_receiver` function to return an error if the receiver has a `PARAMS` entry, even if it is zero.

Status: Resolved

5. Maximum allocation amount can be set lower than total deposits

Severity: Minor

In the `update_config` function the contract owner is able to set a `new_max_allocations_amount` in `contract/builder_unlock/src/contract.rs:689`. There is currently no validation, however, that the new value is bigger than `total_astro_deposited`.

This breaks the assumption that `total_astro_deposited <= max_allocations_amount` and implies that no new allocations can be created.

We classify this issue as minor since only the owner can cause it, and since it can be resolved by the owner.

Recommendation

We recommend ensuring that `max_allocations_amount` cannot be less than `total_astro_deposited` during instantiation and in the `update_config` function.

Status: Resolved

6. Transfer of unallocated tokens leads to wrong token accounting

Severity: Minor

During the `execute_transfer_unallocated` function, the `total_astro_deposited` state variable is not decreased. This is inconsistent with the `execute_increase_allocation` function, which increases the `total_astro_deposited` amount in contracts/builder_unlock/src/contract.rs:519. The total_astro_deposited amount hence includes unallocated tokens.

Consequently, the total_astro_deposited state variable will be bigger than actual deposits after a transfer of unallocated funds. This implies that the condition checking total_astro_deposited against max_allocations_amount in line 523 will lead to an error even if the actual deposited amount does not surpass max_allocations_amount.

We classify this issue as minor since it can be worked around by the owner by increasing max_allocations_amount.

Recommendation

We recommend decreasing the total_astro_deposited state variable in the execute_transfer_unallocated function.

Status: Resolved

7. Lack of validation when creating an allocation

Severity: Minor

In the execute_create_allocations function, the passed AllocationParams are stored in contracts/builder_unlock/src/contract.rs:295. This implies that allocations can be created with a start_time in the past, a zero duration, zero amount or an invalid proposed_receiver address.

Recommendation

We recommend performing validation on the passed AllocationParams.

Status: Resolved

8. Unnecessary lowercasing of addresses is inefficient

Severity: Informational

Throughout the codebase, both the deps.api.validate and the custom addr_validate_to_lower function are used to convert String to Addr types and validate their correctness. As the version of cosmwasm-std defined in the Cargo.toml is 1.1, it is no longer necessary to convert strings to lowercase. Using a custom function is inefficient and unnecessarily increases the codebase.

Recommendation

We recommend removing the use of the custom function `addr_validate_to_lower` and consistently using the standard function `deps.api.validate` instead.

Status: Resolved

9. Unnecessary load of `PARAMS` during creation of an allocation is inefficient

Severity: Informational

During the creation of an allocation, the contract verifies that a user does not have an existing allocation by loading the value from `PARAMS` in `contracts/builder_unlock/src/contract.rs:287`. Loading the value is inefficient though, since it is not used.

Recommendation

We recommend using the `has` method of `PARAMS` as opposed to `load` since `has` does not have to load and parse the value from storage.

Status: Resolved

10. Unnecessary validation of `STATUS` during creation of an allocation is inefficient

Severity: Informational

During the creation of an allocation, the contract verifies that a user does not have an existing allocation by checking the `STATUS` in `contracts/builder_unlock/src/contract.rs:299`. However, there is already a check for `PARAMS` in line `287`. Since both `STATUS` and `PARAMS` are created for new allocations, non-existence of an entry in `PARAMS` implies non-existence of that entry in `STATUS`, which makes the `STATUS` check unnecessary.

Recommendation

We recommend removing the `STATUS` check in `299`.

Status: Resolved

11. Unnecessary loading of `STATUS` during claim of new receiver is inefficient

Severity: Informational

In the `execute_claim_receiver` function, the `STATUS` of the new receiver is loaded in `contracts/builder_unlock/src/contract.rs:649`. However, an error is returned before in line `630` if that new receiver has stored `PARAMS`. Since `PARAMS` and `STATUS` for a receiver always exist both or neither, the code block in lines `649-653` has no effect.

Recommendation

We recommend removing the block in lines `649-653`.

Status: Resolved

12. Misleading function name `compute_withdraw_amount` may negatively impact maintainability

Severity: Informational

The `compute_withdraw_amount` function calculates the amount of Astro a user may withdraw in `contracts/builder_unlock/src/contract.rs:956`. In addition, the function also updates the user's `STATUS`, which contradicts the function name, which only indicates a computation. This might mislead

Recommendation

Rather than updating the `STATUS` in the `compute_withdraw_amount` function, we recommend returning the value and updating the `STATUS` in the calling function.

Status: Resolved

13. Inefficient validation of `prev_receiver`

Severity: Informational

When a user accepts a claim of a token allocation, the function loads the `PARAMS` and `STATUS` of the previous receiver in `contracts/builder_unlock/src/contract.rs:625` and subsequently removes both the `PARAMS` and `STATUS` of said previous receiver.

During each of these four interactions, the function validates the `prev_receiver` address. This is unnecessary and wastes computational resources.

Recommendation

We recommend validating the previous receivers address a single time and storing the address in a variable that can be reused.

Status: Resolved

14. Lack of error message for query of non-existent allocation may confuse users

Severity: Informational

When an allocation does not exist for a queried user through the `QueryMsg::Allocation` query, the `query_allocation` function returns an allocation with empty values in `contracts/builder_unlock/src/contract.rs:760` and `763`, rather than an error message that the allocation does not exist. This may be confusing to users.

Recommendation

We recommend returning an error if an allocation does not exist..

Status: Acknowledged