Audit Report

Comdex Locking and Vesting Contracts

v1.0

October 28, 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 Comdex to perform a security audit of Comdex vesting and locking contracts

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 repositories:

https://github.com/comdex-official/locking-contract

Commit hash: `4cff15537a905d946786dc09744d2c5a4e35ec09`

https://github.com/comdex-official/vesting-contract

Commit hash: `fb6a04151506b8eac60ed45ea0f2fa14481ffb84`

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 Comdex Locking contract provides the core functionality for the Comdex chain's ve governance model in which tokens can be locked for various locking periods which are represented in an NFT. The locking contract also holds the key logic to handle token emission and its distribution. The Comdex Vesting contract provides base vesting functionality for periodic and linear token vesting.

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.

Summary of Findings

No	Description	Severity	Status
1	`PeriodicVesting` is unintendedly releasing tokens to users	Critical	Resolved
2	`LOCKINGADDRESS` is never populated which will block rebasing functionality	Critical	Resolved
3	Multiple rounding issues may cause zero rewards being distributed	Critical	Resolved
4	Unbounded iterations may cause `calculate_rebase_reward` to run out of gas	Major	Resolved
5	Incorrect key results in incorrect calculations in `calculate_bribe_reward` function	Major	Resolved
6	Sudo message `UpdateEmissionRate` overwrites an app's emission struct	Major	Resolved
7	Vesting contract allows any user to register a vesting account for any address and block users	Major	Resolved
8	Unbounded data structures processed in loops make several features prohibitively expensive or even unusable	Major	Resolved
9	Empty master_address restricts user deregistration	Major	Resolved
10	Rebase can be triggered before emission is completed	Minor	Resolved
11	`InstantiateMsg` parameters lack validation	Minor	Resolved
12	`LOCKINGADDRESS` entries are never removed which may introduce state bloat	Minor	Resolved
13	Overall lack of address validation	Minor	Resolved
14	`surplus_share` will not be paid unless there is a bribe in place	Minor	Resolved
15	Broken invariants should be handled as an error	Minor	Resolved
16	Surplus accumulated under a single denom	Minor	Acknowledged
17	NFT tokens hold an unrelated copy of vTokens	Informational	Resolved

	struct		
18	Admin can successfully perform flash staking in voting	Informational	Resolved
19	Custom access controls implementation	Informational	Resolved
20	Funds check can be simplified with `must_pay`	Informational	Acknowledged
21	`start_time`, `end_time`, and `vesting_interval` vesting parameters should be unsigned integers	Informational	Resolved
22	Inconsistent map and item namespaces	Informational	Resolved
23	CW20 vesting not fully implemented	Informational	Resolved
24	Code formatting	Informational	Resolved
25	Spelling errors	Informational	Resolved
26	Unused code	Informational	Resolved
27	Duplicate code	Informational	Acknowledged

Code Quality Criteria

Criteria	Status	Comment
Code complexity	Medium	-
Code readability and clarity	Medium	-
Level of documentation	Medium-High	-
Test coverage	Medium	48.19% test coverage for locking contract 79.12% test coverage for vesting contract

Detailed Findings

1. `PeriodicVesting` is unintendedly releasing tokens to users

Severity: Critical

In `vesting:src/msg.rs:117-138`, the vested amount of tokens that is ready to be redeemed by the user is calculated using a `PeriodicVesting` schedule.

It is supposed to calculate the current number of elapsed intervals since `start_time` in order to compute the total amount of vested tokens.

As demonstrated in the test case in Appendix 1, the code is not calculating the correct vested tokens though, which leads to the following problems:

- At `start_time`, the user can already redeem the first tranche of tokens, even if no interval has elapsed.
- The user gets a token tranche more than expected at the end of the vesting period.

Recommendation

We recommend reworking the code in order to make `PeriodicVesting` compute the correct vesting schedule.

Status: Resolved

2. `LOCKINGADDRESS` is never populated which will block rebasing functionality

Severity: Critical

The `handle_lock_nft` function in `locking:src/contract.rs:256-266` does not store the locking address to `LOCKINGADDRESS`. The function creates a mutable vector of addresses from `LOCKINGADDRESS` where it pushes the sender's address but the address is never saved to `LOCKINGADDRESS`. This results in the map never being populated, which will highly impact other functionality in the contract such as the `calculate_rebase_reward` function that performs a rebase.

Recommendation

We recommend storing the `addresses` vector in `LOCKINGADDRESS` before returning from the `handle_lock_nft` function.

Status: Resolved

3. Multiple rounding issues may cause zero rewards being distributed

Severity: Critical

Several functions of the locking contract are affected by rounding issues since their mathematical operations use `U128` integers for divisions where the numerator is smaller than the denominator. This causes the result to be truncated to zero instead of the desired ratio before it is multiplied, causing the whole operation to be zero. Therefore, reward distribution will result in a zero tokens distribution.

The affected operations can be found on rewards related features. In particular in the `calculate_bribe_reward`, `calculate_rebase_reward` and `calculate_surplus_reward` functions. The affected instances can be found at `locking:src/contract.rs:770`, `870`, `888`, `904`, `919`, `984` and `locking:src/query.rs:322`.

Recommendation

We recommend using the `Decimal` type to calculate ratios before multiplying with other unsigned integers.

Status: Resolved

4. Unbounded iterations may cause `calculate_rebase_reward` to run out of gas

Severity: Major

The `calculate_rebase_reward` function in `locking:src/contract.rs:798` performs an unbounded iteration over all entries in `LOCKINGADDRESS`, and then for each address in `LOCKINGADDRESS` it will iterate through all vtokens for the `gov_token_denom`. Both `VTOKENS` and `LOCKINGADDRESS` are unbounded and have the potential to grow large with time and normal use. In addition, as mentioned in the finding *LOCKINGADDRESS entries are never removed which may introduce state bloat*, `LOCKINGADDRESS` can never be reduced, exacerbating this issue.

The impact of this issue is that the rebasing functionality of the contract could become blocked for a specific `app_id`.

Recommendation

We recommend implementing a pull-over-push based approach to the rebase functionality. Rather than relying on one invocation to rebase all addresses for a specific app, we recommend allowing addresses to invoke this function individually or potentially building in a rebase call to other messages when a user invokes the contract.

Status: Resolved

5. Incorrect key results in incorrect calculations in `calculate_bribe_reward` function

Severity: Major

The `calculate_bribe_reward` function uses an incorrect key to load the contents of both `BRIBES_BY_PROPOSAL` and `PROPOSALVOTE`. This results in the bribe rewards feature not being usable or not behaving as designed.

In both cases, one in `locking:src/contract.rs:762` and the other in `locking:src/contract.rs:765`, the keys used to load the desired data are `(proposal1.app_id, vote.extended_pair)` that are the application ID and the extended pair selected by the voter. However, the actual keys used for both storage elements are `(proposal_id, extended_pair)`. This implies that instead of retrieving the information of the desired proposal identified by `proposal_id`, the data related to the proposal with the ID equal to the Application ID will be returned.

Recommendation

We recommend using the target proposal ID as the first key to load both pieces of storage instead.

Status: Resolved

6. Sudo message `UpdateEmissionRate` overwrites an app's emission struct

Severity: Major

The `SudoMsg::UpdateEmissionRate` entry point in `src/contract.rs:1416` takes an entire emission struct and overwrites the existing entry for that specific app in `EMISSION` rather than simply updating `emmission_rate`. This overwrites the existing `total_rewards`, `rewards_pending`, and `distributed_rewards`.

Recommendation

We recommend supplying only the `emmission_rate` to `SudoMsg::UpdateEmissionRate` and using that value to update the `EMISSION` entry. This value should also be validated to ensure that the emission rate is not greater than `1.0`.

Status: Resolved

7. Vesting contract allows any user to register a vesting account for any address and block users

Severity: Major

The `register_vesting_account` function in `vesting:src/contract.rs:78` allows any user to register a vesting account for any address. This is problematic because `VESTING_ACCOUNTS` only allows one vesting account per user/denom combo.

In fact, this allows a malicious actor to spam the network with `RegisterVestingAccount` messages with a small amount of `denom` tokens to a large number of users' addresses, preventing these users from registering vesting accounts for that `denom`.

Recommendation

This issue arises because an address can only have one vesting account per denom. If the protocol requires that `VESTING_ACCOUNTS` may only contain one entry per denom, we recommend restricting the vesting account registration to governance, otherwise we recommend implementing the ability for an address to have multiple vesting accounts per denom. Another solution can be allowing the user to deregister their own vesting account.

Status: Resolved

8. Unbounded data structures processed in loops make several features prohibitively expensive or even unusable

Severity: Major

In the handling of the following messages:

- `Withdraw` in `locking:src/contract:90`,
- `Transfer` in `locking:src/contract:91`, and
- `ClaimReward` in `locking:src/contract:84`,

the use of multiple and nested iterations through unbounded vectors could lead the execution to run out of gas.

This implies that users may not be able to perform these calls when the vectors contain too many entries.

Also, this issue drastically reduces the composability of the protocol with third party contracts, because these unbounded datastructures lead to high execution costs.

For example in the handling of the `Transfer` message, the array containing the `vToken` of the sender and the recipient accounts is iterated four times. That means that if one of the users has a lot of `vToken` instances stored on chain or if a third party contract manages `vTokens` for a lot of different users, the execution may get prohibitively expensive up to the point where it becomes unusable.

Recommendation

We recommend reducing the overall use of iterations when not strictly needed.

Status: Resolved

9. Empty `master_address` restricts user deregistration

Severity: Major

According to the documentation, the vesting contract's `deregister_vesting_account` function should allow for an empty `master_address` in which case the user address should act as master too. This is not reflected in the implementation, which allows for new accounts to register without providing a `master_address` but not having the actual ability to deregister themselves at a later stage.

In `vesting:src/contract.rs:237`, the contract checks if `master_address` is `None` and throws an `Unauthorized` error in that case. As `vesting:src/contract.rs:185` directly clones the provided `master_address` option into the `VestingAccount` struct, this causes that a valid empty option results in broken functionality for the deregistration feature.

Recommendation

In order to adhere to the documentation while maintaining functionality, we recommend substituting `vesting:src/contract.rs:185` with `master_address: master_address.unwrap_or_else(address)`. This will set the user's address when an empty master option is provided.

In addition, the `None` check in `vesting:src/contract.rs:237` should be removed as the value would never be `None` once the above fix is applied.

Finally, `vesting:src/contract.rs:205` should be modified to reflect the correct address on the response attributes.

Status: Resolved

10. Rebase can be triggered before emission is completed

Severity: Minor

The documentation of the locking contract states that the rebase feature should only be triggered when the emission has been complete, similar to the case of founding rewards. However, the `calculate_rebase_reward` function in `locking:src/contract.rs:798` does not adhere to these specifications and the condition is not enforced. This would cause the rebase to be marked as done with incomplete information being used during calculation, not distributing the expected rewards.

This issue has been raised as minor as the rebase feature is controlled by the admin user, therefore the impact would be limited.

Recommendation

We recommend requiring completion of the emission as a prerequisite to executing the rebase, as done in the `emission_foundation` function.

Status: Resolved

11. `InstantiateMsg` parameters lack validation

Severity: Minor

The `instantiate` function in `locking:src/contract.rs:31` does not properly validate the `InstantiateMsg` parameters.

a) It is best practice to validate addresses before saving them to the contract state. Currently `msg.vesting_contract`, `msg.admin` and the vector of addresses in `msg.foundation_addr` are not validated. In addition validation should be implemented to ensure that the `msg.foundation_addr` does not contain duplicate addresses.
b) `msg.foundation_percentage` should be validated to ensure that it is set to a value that is not greater than `1.0`.
c) `msg.emission` is also not properly validated. The emission struct should be validated to ensure that it refers to a valid `app_id`, is initialized with `total_rewards`, `rewards_pending`, and `distributed_rewards` all equal to 0 and with an `emmission_rate` less than 1.

Recommendation

We recommend adding additional validation to this function by performing `addr_validate` before saving addresses to state, deduplicating `msg.foundation_addr`, and ensuring that `msg.foundation_percentage` is not greater than `1.0`. In addition, we recommend that proper validation is performed on `msg.emission` as described above.

Status: Resolved

12. `LOCKINGADDRESS` entries are never removed which may introduce state bloat

Severity: Minor

The `handle_transfer` function in `locking:src/contract.rs:466` does not remove addresses from `LOCKINGADDRESS` after a transfer has been performed and the sender no longer has a lock for a specific `app_id`. This does not have a direct impact on user funds, but it does contribute to state bloat as the `LOCKINGADDRESS` map is never decreased in size.

Recommendation

We recommend removing the address from `LOCKINGADDRESS` after an address has transferred its tokens of a specific denom.

Status: Resolved

13. Lack of address validation

Severity: Minor

The contracts in scope lack address validation in several functions. While some cases do not represent an actual security risk but just an inconvenience to the user, such as in query messages, others do, such as in the `RegisterVestingAccount` function of the vesting contract where a direct loss of funds could happen.

The function `register_vesting_account` in `vesting:src/contract.rs:78` which is called when handling `RegisterVestingAccount` messages, is not validating `address` and `master_address` parameters. Invalid addresses could cause the loss of the vested funds.

The following list details different instances of lack of address validation:

- `locking:src/contract.rs:1412 - address`
- `locking:src/contract.rs:1425 - address`
- `locking:src/contract.rs:1442 - admin`
- `locking:src/query.rs:104 - address`
- `locking:src/query.rs:156 - address`
- `locking:src/query.rs:225 - address`
- `vesting:src/contract.rs:335 - recipient`
- `vesting:src/contract.rs:253 - vested_token_recipient`
- `vesting:src/contract.rs:281 - left_vesting_token_recipient`
- `vesting:src/contract.rs:450 - address`

This issue is considered to be minor given that CW20 tokens are not yet supported and native token transfers would fail when moving funds to a badly formatted

address. Please note that if CW20 gets supported, some of the instances mentioned will cause a major impact as funds could be locked/lost.

Recommendation

We recommend performing validation of all addresses used in the contract through `deps.api.addr_validate()`.

Status: Resolved

14. `surplus_share` will not be paid unless there is a bribe in place

Severity: Minor

The `claim_rewards` function aggregates the funds to be paid as surplus to a previous calculation of the funds to be paid as part of the bribe in `locking:src/contract.rs:714-717`. However, as this is done by iterating the coins from the bribe, if the denom to be paid as surplus is not part of them it will not be paid at all. This causes the surplus to not be rewarded unless a matching bribe is in place in the proposal.

Recommendation

We recommend adding the funds of the surplus to the `bribe_coins` array, or updating the existing `Coin` object if already added.

Status: Resolved

15. Broken invariants should be handled as an error

Severity: Minor

If `total_vest` is `None` in `vesting:src/contract.rs:406-410` and `vesting:src/contract.rs:310-314`, it means that an invariant is broken and the execution should stop and return an error.

The current implementation assigns 0 to `total_vested` and this leads to underflows in `vesting:src/contract.rs:411` and `vesting:src/contract.rs:315`.

Recommendation

We recommend returning an error when an invariant is broken instead of continuing the execution flow.

Status: Resolved

16. Surplus accumulated under a single denom

Severity: Minor

The locking contract's `calculate_surplus_reward` function iterates over every completed and unclaimed proposal accumulating each individual surplus under a single `Coin` struct. In case the surplus denom changes at some point, the full surplus will get paid under the latest denom instead of both the new and the old.

During each iteration, the denom gets updated to the current proposal's surplus denom in `locking:src/contract.rs:964` and its amount gets accumulated. As it is overwritten in each iteration, the final value will be the one of the latest completed proposals processed, not taking into account the scenario where two different denoms are found in the proposals.

We classify this issue as minor since it can only be caused through governance.

Recommendation

We recommend using `Coins` for the surplus distribution, adding a new element for each different denom found during the affected loop.

Status: Acknowledged

17. NFT tokens hold an unrelated copy of vTokens struct

Severity: Informational

The `TokenInfo` struct defined in `locking:src/state.rs:60` which represents the protocol `NFT` implementation, defines `vtokens` as a `Vec<Vtoken>`.

This implies that any time that a `vtoken` is created and stored, a copy of it is created and added to the `vtokens` vector of `TokenInfo`.

This implementation has the consequence that every time a `vtoken` is updated, also the stored copy in `TokenInfo` needs to be updated. This is error-prone and inefficient.

Recommendation

We recommend referencing `vtokens` in `TokenInfo` instead of copying them.

Status: Resolved

18. Admin can successfully perform flash staking in voting

Severity: Informational

The locking contract implements a mechanism to avoid users taking advantage of flash staking to disrupt proposal voting by adding a block height timestamp restriction based on

the proposal's creation. However, the administrator could still create a transaction where a new proposal is created and their vote cast while taking a flash loan in the same block.

This issue has been raised as informational only as the administrator is considered to be a trusted party. However, in case the account is compromised or a malicious insider is in place they could subvert the voting of important proposals.

Recommendation

We recommend either restricting the admin user access to the voting feature or calculating the voting power using the block before the one in which the proposal is created.

Status: Resolved

19. Custom access controls implementation

Severity: Informational

The locking contract implements custom access controls. Although no instances of broken controls or bypasses have been found, using a battle-tested implementation reduces potential risks and the complexity of the codebase.

Also, the access control logic is duplicated across the handlers of each function, which negatively impacts the code's readability and maintainability.

Recommendation

We recommend making use of a well-known access controls implementation such as `cw_controllers::Admin` (https://docs.rs/cw-controllers/0.14.0/cw_controllers/struct.Admin.html).

Status: Resolved

20. Funds check can be simplified with `must_pay`

Severity: Informational

The `handle_lock_nft` and `bribe_proposal` functions in `locking:src/contract.rs:244-254` and `620-632` both perform validations to ensure the proper funds are sent to the contract. It is common practice to use the `cw_utils must_pay` function to simplify this validation.

Recommendation

We recommend using the `must_pay` function to simplify the fund-checking logic, see https://docs.rs/cw-utils/latest/src/cw_utils/payment.rs.html#32-39.

Status: Acknowledged

21. `start_time`, `end_time`, and `vesting_interval` vesting parameters should be unsigned integers

Severity: Informational

In `vesting:src/msg.rs:75-76` and `vesting:src/msg.rs:85-86` the `start_time` and `end_time` parameters are defined of type `String`.

As they contain timestamp information and need to be parsed to integer every time they are used, it should be better to directly define them as `uint`.

In addition, `vesting_interval` should not be a String type. It is best practice to avoid these type conversions.

Recommendation

We recommend changing the type of `start_time`, `end_time`, and `vesting_interval` parameters to `uint`.

Status: Resolved

22. Inconsistent map and item namespaces

Severity: Informational

The items and maps in `locking:src/state.rs:155-175` have inconsistent namespaces that may impact the readability and upgradability of the code in the future. We recommend using a consistent naming convention. Currently, the namespaces use a mix of spaces, underscores, upper-/lowercase letters, and trailing spaces.

Recommendation

We recommend creating a standard namespace naming convention and modifying the namespaces mentioned in `locking:src/state.rs:155-175` to match this standard.

Status: Resolved

23. CW20 vesting not fully implemented

Severity: Informational

The vesting contract contains a partial and non-functional implementation of CW20 support in addition to native tokens. The Comdex team stated that this will be finalized in the future. Although not a security issue itself, at the moment it creates a clear instance of unused code that negatively impacts both the contract's readability and maintainability.

Recommendation

We recommend removing unused code or keeping it in a feature branch of the repository. We also recommend getting the CW20 vesting code audited once it has been finalized.

Status: Resolved

24. Code formatting

Severity: Informational

The codebase is not consistently formatted, which negatively impacts the readability and maintainability of the codebase.

Recommendation

We recommend running the `cargo fmt` command to consistently format the code.

Status: Resolved

25. Spelling errors

Severity: Informational

The following spelling errors were found in the codebase:

- `locking:src/contract.rs:85 - ExecuteMsg::Emmission`
- `locking:src/state.rs:138 - emmission_rate`
- `locking:src/contract.rs:92, 95 - recipent`
- `locking:src/contract.rs:127 - calucation`

Recommendation

We recommend correcting these spelling errors.

Status: Resolved

26. Unused code

Severity: Informational

The `Rewards` struct in `locking:src/state.rs:150` is currently unused. It is best practice to remove unused code before releasing the contracts into production.

Recommendation

We recommend removing the `Rewards` struct in `locking:src/state.rs:150`.

Status: Resolved

27. Duplicated code

Severity: Informational

Some instances of duplicated code have been found. Although not a security risk, it negatively impacts the maintainability of the codebase and could lead to bugs if implementations are changed in some (but not all) places.

The locking contract implements two identical functions in `locking:src/helpers.rs:34` and `49` named `get_token_supply` and `get_token_vote_weight`. In addition, `calculate_bribe_reward` in `locking:src/contract.rs:741` and `calculate_bribe_reward_query` in `locking:src/query.rs:286` share a lot of code, although they are not completely identical.

Recommendation

We recommend refactoring the affected functions to deduplicate the functionality.

Status: Acknowledged

Appendix

1. **Test case for “PeriodicVesting is unintendedly releasing tokens to users”**

```
fn periodic_vesting_vested_amount_hack() {
    let schedule = VestingSchedule::PeriodicVesting {
        start_time: "105".to_string(),
        end_time: "110".to_string(),
        vesting_interval: "5".to_string(),
        amount: Uint128::new(500000u128),
    };

    assert_eq!(schedule.vested_amount(100).unwrap(), Uint128::zero());

    //FAILS. Got the first tranche at the start_time
    assert_eq!(
        schedule.vested_amount(105).unwrap(),
        Uint128::zero()
    );

    //FAILS. Got the first tranche at the start_time
    assert_eq!(
        schedule.vested_amount(106).unwrap(),
        Uint128::zero()
    );

    //FAILS. Got double of the intended amount
    assert_eq!(
        schedule.vested_amount(110).unwrap(),
        Uint128::new(500000u128)
    );
}
```