Audit Report

Volta CosmWasm

v1.0

September 18, 2024

Table of Contents

License

Disclaimer

This audit has been performed by

Oak Security GmbH

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

Introduction

Purpose of This Report

Oak Security GmbH has been engaged by NuKey Inc to perform a security audit of Volta CosmWasm 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 target:

Repository	https://github.com/VoltaHQ/smart-contract-cosmwasm
Commit	`a5c6f5d916bfad53d977ba679a0c1f4e45c086fe`
Scope	All contracts were in scope.
Fixes verified at commit	`88af26a937ecf07ab53bd54cb5c5d6b2e5817546` Note that only fixes to the issues described in this report have been reviewed at this commit. Any further changes such as additional features have not been reviewed.

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

Volta Circuit is a non-custodial, programmable multisig wallet solution designed for digital asset management.

Volta Circuit integrates multi-signature technology with programmable smart contracts, providing customizable governance structures. This combination enables organizations to maintain control over their digital assets while implementing sophisticated management frameworks.

The scope of this audit is restricted to the CosmWasm smart contracts.

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	Medium	-
Code readability and clarity	Medium-High	-
Level of documentation	Medium-High	The client provided very detailed documentation including diagrams of the architecture.
Test coverage	Medium	`cargo tarpaulin` reports a `55.45%` test coverage.

Summary of Findings

No	Description	Severity	Status
1	Incorrect vote logic prevents correct proposal processing	Major	Resolved
2	Malicious owners or users spamming messages could prevent the admin from executing `ProposalType::Configuration` proposals	Major	Resolved
3	Griefing risk due to unlimited fee grant allowance	Major	Resolved
4	Partial validation for `ProposalType::Configuration`	Minor	Resolved
5	Partial validation for `ProposalType::Rules`	Minor	Acknowledged
6	Inconsistent handling of duplicate keys during flattening	Minor	Acknowledged
7	`ProposalType::CosmosMsgs` proposals cannot be completed if contain failing messages	Informational	Acknowledged
8	The `RevokeAllRules` proposal cannot be queried	Informational	Resolved
9	Config proposal is missing from query options	Informational	Resolved
10	Superseded proposal votes are not reset in case of owner update	Informational	Acknowledged
11	Non-valid owner addresses are ignored without returning an error	Informational	Resolved
12	Contracts should implement a two step ownership transfer	Informational	Acknowledged
13	Miscellaneous comments	Informational	Partially Resolved

Detailed Findings

1. Incorrect vote logic prevents correct proposal processing

Severity: Major

In `src/msg.rs:224-231`, the `vote` function determines the state of a proposal based on the number of `yes` and `no` votes against a `threshold`. The proposal is rejected if the remaining votes are unable to achieve the required number of approvals.

However, the logic in line `227` for calculating the remaining votes is incorrect, leading to an underflow if the total votes exceed the threshold.

For instance, in a `(3,5)` threshold scheme with `2 yes` votes and `2 no` votes, the condition `no > (threshold - totalVotes)` causes a panic.

Consequently, this results in the specific proposal not being processed.

Recommendation

We recommend redefining the vote logic to account for whether the proposal can still achieve the required number of `yes` votes based on the remaining votes.

Status: Resolved

2. Malicious owners or users spamming messages could prevent the admin from executing `ProposalType::Configuration` proposals

Severity: Major

In `src/contract.rs:451-470`, during the execution of the `vote` function, if the `ProposalType::Configuration` is passed, all active proposals are iterated to reset the votes since the owners might have changed.

Since messages that do not match any rule are not rejected but are instead added to a new proposal in `ACTIVE_COSMOS_MSG_PROPOSALS`, this behavior can be exploited by malicious users or owners by spamming `CosmosMsg` with inner cosmos messages that always fail or with messages that do not match any rule.

Consequently, attackers can grow this vector arbitrarily and without paying fees since they have the grant to let the execution of `ProposalType::Configuration` fail, not allowing admins to update wallet configurations.

Recommendation

We recommend rejecting messages that do not match the defined rules.

Status: Resolved

3. Griefing risk due to unlimited fee grant allowance

Severity: Major

In `src/state.rs:168-184`, the `generate_fee_grant` function constructs `MsgGrantAllowance` messages with an `AllowedMsgAllowance` that utilizes an unlimited `BasicAllowance`.

This setup poses a griefing risk by permitting unrestricted fee spending for any CosmWasm message.

If an account is compromised, attackers could exploit this vulnerability by executing numerous `MsgExecuteContract` transactions with arbitrary contracts, draining the contract's funds.

Recommendation

We recommend using `PeriodicAllowance` instead of `BasicAllowance` to impose spending limits and enhance security.

Status: Resolved

4. Partial validation for `ProposalType::Configuration`

Severity: Minor

In `src/msg.rs:102-111`, during the handling of `ProposalType::Configuration` creation, the execution performs some validation to ensure that the proposal is correctly constructed.

However, it is not ensured that the new list of `owners` does not include duplicates and that all addresses differ from the admin address. Additionally, the code does not verify that the number of new owners is greater than two and less than `m`.

Consequently, invalid proposals could be created and stored in the contract.

Recommendation

We recommend validating `ProposalType::Configuration` proposals before storing them in the contract.

Status: Resolved

5. Partial validation for `ProposalType::Rules`

Severity: Minor

In `src/msg.rs:157-174`, during the handling of `ProposalType::Rules` creation, the execution performs some validation to ensure that the proposal is correctly constructed.

However, rules are accepted without any checks, which can lead to the inclusion of incompatible ones. For instance, a rule set containing both greater than one and less than one for the same value is accepted without validation.

Consequently, invalid proposals could be created and stored in the contract.

Additionally, there is no limit on the number of `RuleSet` items that can be provided, potentially leading to out-of-gas errors when executing messages.

Recommendation

We recommend validating `ProposalType::Rules` proposals before storing them in the contract.

Status: Acknowledged

6. Inconsistent handling of duplicate keys during flattening

Severity: Minor

In `src/flatten.rs:66-77`, the `insert_value` function handles duplicate keys by creating an array of values.

However, the current implementation may lead to confusion and potential errors when applying rulesets to transaction fields that expect unique keys.

Specifically, when flattening nested maps with duplicates, it becomes unclear which value should be associated with the key, potentially causing incorrect interpretation and application of transaction rulesets.

A test case showcasing the issue is provided in the Appendix.

Recommendation

We recommend returning an error in case of duplicated keys.

Status: Acknowledged

7. `ProposalType::CosmosMsgs` proposals cannot be completed if contain failing messages

Severity: Informational

In `src/contract.rs:471-485`, during the execution of the `vote` function, if the `ProposalType::CosmosMsgs` passes, all the messages are added as sub-messages to be executed.

However, if one of these messages fails, the entire transaction reverts.

Consequently, it becomes impossible for the last `owner` to send the vote and for the proposal to be completed.

Recommendation

We recommend implementing a mechanism to complete proposals while ensuring that proposal execution remains atomic. This could be achieved by allowing any user to complete pending proposals after a certain time has passed. The current implementation ensures atomicity, which is important to prevent partial execution of proposals but does not allow for completion.

Status: Acknowledged

8. The `RevokeAllRules` proposal cannot be queried

Severity: Informational

In `src/msg.rs:128-131`, during the handling of `ProposalType::RevokeAllRules` creation, the newly computed proposal ID is not stored in `REVOKE_ALL_PROPOSAL`, and the default value is always returned for the superseding mechanism.

Consequently, this prevents the proper execution of this type of proposal since the first ID is set to one, but the function will always return zero, which is the default value.

As a result, any subsequent attempts to query a `RevokeAllRules` proposal will fail because the necessary ID is not being updated and stored correctly.

Recommendation

We recommend storing the proposal ID for `ProposalType::RevokeAllRules`.

Status: Resolved

9. Config proposal is missing from query options

Severity: Informational

In `src/contract.rs:607`, the `proposals` function in the `query` module allows a user to query for proposals based on the `ProposalFilter` passed into the query message.

However, there is no `ProposalFilter` for returning the current config proposal from `CONFIG_PROPOSAL` storage.

Recommendation

We recommend adding a `ProposalFilter` to allow for returning the `CONFIG_PROPOSAL`. Moreover, the `ProposalFilter::All` should also include the `CONFIG_PROPOSAL` if there is one.

Status: Resolved

10. Superseded proposal votes are not reset in case of owner update

Severity: Informational

In `src/contract.rs:451-470`, during the execution of the `vote` function, if the `ProposalType::Configuration` is passed, all active proposals are iterated to reset the votes since the owners might have changed.

However, the `reset` method of the proposal only resets the votes for open proposals and does not reset votes for superseded proposals.

Consequently, in a scenario where a proposal with votes is superseded by another proposal and moves to the superseded status, the votes remain. If a `ProposalType::Configuration` is executed and owners are updated, the old votes will still be present, leading to an incorrect calculation of the result.

Recommendation

We recommend resetting votes for superseded proposals.

Status: Acknowledged

11. Non-valid owner addresses are ignored without returning an error

Severity: Informational

In the `validate_and_save_config` function, defined in `src/state.rs:18-65`, each owner address is validated using `addr_validate`.

However, in line `30` the owner is removed if not valid rather than returning a `ContractError`.

This can lead to UX issues and potential misconfiguration if the multisig is expected to be created with a specific number of owners.

Recommendation

We recommend returning a `ContractError` if any of the owner addresses are not valid.

Status: Resolved

12. Contracts should implement a two-step ownership transfer

Severity: Informational

The contracts within the scope of this audit allow the current owner to execute a one-step ownership transfer. While this is common practice, it presents a risk for the ownership of the contract to become lost if the owner transfers ownership to the incorrect address.

A two-step ownership transfer will allow the current owner to propose a new owner, and then the account that is proposed as the new owner may call a function that will allow them to claim ownership and actually execute the config update.

Recommendation

We recommend implementing a two-step ownership transfer. The flow can be as follows:

1. The current owner proposes a new owner address that is validated and lowercase.
2. The new owner account claims ownership, which applies the configuration changes.

Status: Acknowledged

13. Miscellaneous comments

Severity: Informational

Miscellaneous recommendations can be found below.

Recommendation

The following recommendations to improve the overall code quality and readability:

- The `cosmwasm-std` version in use has dependencies that are affected by vulnerabilities. For instance, `curve25519-dalek` is affected by `RUSTSEC-2024-0344`. We recommend updating `cosmwasm-std` to the latest version.
- In `src/state.rs:28-36`, the `validate_and_save_config` function iterates the `owners` vector two times to execute filtering operations. We recommend merging the filter predicates in the same iteration.
- In `src/helpers.rs` and `src/integration_tests.rs`, the whole code is commented out so the files can be removed.
- In `src/lib.rs`, the `helpers` and `integration_tests` modules are being imported. Since these are both to be removed, the module references can also be removed from `src/lib.rs`.
- In `src/state.rs`, the `next_id` function can be called to return the next proposal id. Therefore it is more semantically correct to call the function `next_proposal_id`.
- In `src/flatten.rs`, the associated tests are not performing any actual `assert` calls, which can prevent build systems from reporting failed tests during build and impact quality assurance feedback. Therefore, we recommend adding `assert` statements to all tests.
- In `src/state.rs:52` and `62`, the `validate_and_save_config` function saves the `CONFIG` storage two times. We recommend removing the `CONFIG.save` call in line `52`.
- In `src/state.rs`, the storage items are given single-character or double-character namespaces. For example, `CONFIG` is assigned to the namespace `"c"`. Using a single character for storage keys can makes the code harder to read and also increases the chance of a storage key collision. We recommend using full, descriptive namespaces such as `"config"`.
- In `src/contract.rs`, the `vote` function populates a `proposals_to_reset` vector. However, the `revoke_all_proposal_to_reset` is being chained to the vector twice in lines `440` and `444`. We recommend removing the `chain` call in line `444`.

Status: Partially Resolved

Appendix A: Test Cases

1. Test case for “Inconsistent handling of duplicate keys during flattening”

Initial Setup:

- A deeply nested map is created.
- Two identical maps are put into an array.
- This array is then inserted into the target map under the key "a".

```
 # [test]
    fn test_flatten_large_input_data() {
        let mut inner_map_1 = BTreeMap::new();
        inner_map_1.insert(Value::String("f".to_string()),
Value::String("g".to_string()));
        let mut inner_map_2 = BTreeMap::new();
        inner_map_2.insert(Value::String("e".to_string()),
Value::Map(inner_map_1));
        let mut inner_map_3 = BTreeMap::new();
        inner_map_3.insert(Value::String("d".to_string()),
Value::Map(inner_map_2));
        let mut inner_map_4 = BTreeMap::new();
        inner_map_4.insert(Value::String("c".to_string()),
Value::Map(inner_map_3));
        let mut inner_map_5 = BTreeMap::new();
        inner_map_5.insert(Value::String("b".to_string()),
Value::Map(inner_map_4));
        let mut array_1 = vec![
            Value::Map(inner_map_5.clone()),
            Value::Map(inner_map_5),
        ];
        let mut target = BTreeMap::new();
        target.insert(Value::String("a".to_string()),
Value::Seq(array_1));
        let flat = flatten(&target);
        assert_eq!(flat.get(&Value::String("a.b.c.d.e.f".to_string())),
Some(&Value::String("g".to_string())));
    }
```