Audit Report

DAO DAO Veto

v1.0

January 10, 2024

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 In With The New to perform a security audit of the DAO DAO Veto feature.

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/DA0-DA0/dao-contracts
Commit	`4cc8cc4afdd74e12ba7e69daf5a17caa333f72be`
Scope	The scope was restricted to the following contract changes since our last audit, which was performed at commit `7f89ad1604e8022f202aef729853b0c8c7196988`. • `contracts/proposal/dao-proposal-single` • `contracts/proposal/dao-proposal-multiple` • `packages/dao-hooks` • `packages/dao-pre-propose-base` • `packages/dao-voting`

Fixes verified at commit	`a3a72db2faea4a891e5c72d46ffeb1ed6ac097f5` Note that changes to the codebase beyond fixes after the initial audit have not been in scope of our fixes review.

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 submitted code implements the smart contracts for DAO DAO, which creates a modular framework for creating DAOs, including staking and voting functionalities.

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**	Most functions are well-documented with concise and clear code comments.
Level of documentation	**Medium-High**	The client shared an architectural diagram demonstrating the proposal flow status in the `dao-proposal-single` and `dao-proposal-multiple` contracts' `README` files.
Test coverage	**Medium-High**	`cargo tarpaulin` reports an `81.49%` test coverage.

Summary of Findings

No	Description	Severity	Status
1	Unvalidated veto timelock duration	Minor	Acknowledged
2	State migration is not implemented in the `dao-proposal-multiple` contract	Minor	Acknowledged
3	Vetoer can execute `only_members_execute` proposals even if they are not part of the DAO	Informational	Resolved
4	Potential inability to veto proposals if chain halts	Informational	Acknowledged

Detailed Findings

1. Unvalidated veto time lock duration

Severity: Minor

In contracts/proposal/dao-proposal-multiple/src/contract.rs:67 and 634, the timelock_duration is not validated to be an actionable duration for humans, i.e., very short and even zero durations are allowed.

Consequently, this could lead to falsely executed proposals because the veto time lock duration is so short that it is de facto skipped even if it was configured.

We classify this issue as minor because only the DAO, which is a privileged address, can configure the veto timelock duration.

This issue also applies to contracts/proposal/dao-proposal-single/src/contract.rs:68, 653, and 967.

Recommendation

We recommend validating the timelock_duration to be larger than a minimum duration.

Status: Acknowledged

The client states that they decided that there is no meaningful validation to set on the veto time lock duration at this point. For example, there may be a configuration that optimizes for governance speed where the DAO enables veto_before_passed and sets timelock_duration to 0 so that once a proposal's voting period ends, it is not time-locked at all, but the vetoer still has the opportunity to veto ahead of time.

The client prefers allowing the configuration to be maximally customizable for this first implementation, and they can make more refined decisions later once use cases are well understood.

2. State migration is not implemented in the dao-proposal-multiple contract

Severity: Minor

In contracts/proposal/dao-proposal-multiple/src/contract.rs:1030, the migrate entry point does not implement any state migration to update existing proposals to support veto configuration. This is indicated in the comment in contracts/proposal/dao-proposal-multiple/src/msg.rs:235-239.

Consequently, migrating old `dao-proposal-multiple` contracts to use the latest version will cause proposals to fail to serialize and deserialize properly, causing the contract to fail to work as expected.

We classify this issue as minor because the contract migration admin can recover this by implementing the state migration and migrating the contract into a new code ID.

Recommendation

We recommend implementing migration similar to `contracts/proposal/dao-proposal-single/src/contract.rs:944-1044`.

Status: Acknowledged

The client states that state migration is not implemented for multiple-choice proposals because that contract has only ever existed on v2. Since there are no v1 DAOs with multiple choice, no v1 multiple choice contracts have been deployed, and thus there are none to migrate.

3. Vetoer can execute `only_members_execute` proposals even if they are not part of the DAO

Severity: Informational

In `contracts/proposal/dao-proposal-multiple/src/contract.rs:466-473`, the `execute_execute` function allows the vetoer to execute proposals when `config.only_members_execute` is enabled. This allows the vetoer to execute the proposal early if `veto_config.early_execute` is `true`, as seen in lines `492-504`.

However, if `veto_config.early_execute` is `false` and the proposal passes, the vetoer can execute it even if they are not one of the DAO members.

We classify this issue as informational because the vetoer can only be configured by the DAO, which is a privileged address.

Please see the `test_veto_execute_only_members_execute_proposal` test case in the appendix to reproduce this issue.

Recommendation

We recommend only allowing DAO members to execute the proposal if the proposal status is `Status::Passed`.

Status: Resolved

4. Potential inability to veto proposals if chain halts

Severity: Informational

In `contracts/proposal/dao-proposal-single/src/proposal.rs:82-88`, the `current_status` function determines the proposal status by checking whether the current timestamp exceeds the proposal expiration and veto timelock duration. If it exceeds, the proposal is evaluated as `Status::Passed`.

This is problematic because if a chain halt occurs and the chain is restarted, the current timestamp might already exceed the proposal expiration and veto timelock duration. In that case, the vetoer would not have time to react (e.g., to veto and cancel the proposal), allowing the proposal to be executed even if rejecting the proposal was the intended decision.

This issue also exists in `contracts/proposal/dao-proposal-multiple/src/proposal.rs:82`.

Recommendation

We recommend modifying the implementation so the expiration time is computed based on the current timestamp and the veto timelock duration. Alternatively, we recommend using block height-based instead of timestamp-based proposal expiration.

Status: Acknowledged

The client states that the chain halt risk is an inherent tradeoff in deciding to use time-based expirations over block-based expirations, and the DAO can weigh all the tradeoffs when choosing its configuration. They reason that it's mostly participation and awareness issues that would contribute to missing an expiration, which needs to be solved on the human level via the front end. They intend to show proposals that a DAO can veto in the UI and send notifications to the vetoer DAO's members to address the awareness issue.

Appendix

1. Test case for "Vetoer can execute `only_members_execute` proposals even if they are not part of the DAO"

```
#[test]
pub fn test_veto_execute_only_members_execute_proposal() {

    // modification of
test_allow_voting_after_proposal_execution_pre_expiration_cw20
    // reproduced in
contracts/proposal/dao-proposal-multiple/src/testing/adversarial_tests.rs

    let mut app = App::default();
    let vetoer = "vetoer";

    let instantiate = InstantiateMsg {
        voting_strategy: VotingStrategy::SingleChoice {
            quorum: PercentageThreshold::Percent(Decimal::percent(66)),
        },
        max_voting_period: Duration::Time(604800),
        min_voting_period: None,
        only_members_execute: true,
        allow_revoting: false,
        pre_propose_info: get_pre_propose_info(
            &mut app,
            Some(UncheckedDepositInfo {
                denom: dao_voting::deposit::DepositToken::VotingModuleToken {},
                amount: Uint128::new(10_000_000),
                refund_policy: DepositRefundPolicy::OnlyPassed,
            }),
            false,
        ),
        close_proposal_on_execution_failure: true,
        veto: Some(dao_voting::veto::VetoConfig {
            timelock_duration: Duration::Time(604800),
            vetoer: vetoer.to_string(),
            early_execute: false,
            veto_before_passed: false
        }),
    };

    let core_addr = instantiate_with_multiple_staked_balances_governance(
        &mut app,
        instantiate,
        Some(vec![
            Cw20Coin {
                address: CREATOR_ADDR.to_string(),
                amount: Uint128::new(100_000_000),
            },
            Cw20Coin {
                address: ALTERNATIVE_ADDR.to_string(),
                amount: Uint128::new(50_000_000),
            },
        ]),
    );
    let proposal_module = query_multiple_proposal_module(&app, &core_addr);
    let gov_token = query_dao_token(&app, &core_addr);

    // Mint some tokens to pay the proposal deposit.
    mint_cw20s(&mut app, &gov_token, &core_addr, CREATOR_ADDR, 10_000_000);

    // Option 0 would mint 100_000_000 tokens for CREATOR_ADDR
    let msg = cw20::Cw20ExecuteMsg::Mint {
        recipient: CREATOR_ADDR.to_string(),
        amount: Uint128::new(100_000_000),
    };
    let binary_msg = to_json_binary(&msg).unwrap();

    let options = vec![
        MultipleChoiceOption {
            title: "title 1".to_string(),
            description: "multiple choice option 1".to_string(),
            msgs: vec![WasmMsg::Execute {
                contract_addr: gov_token.to_string(),
                msg: binary_msg,
                funds: vec![],
            }
            .into()],
        },
        MultipleChoiceOption {
            title: "title 2".to_string(),
            description: "multiple choice option 2".to_string(),
            msgs: vec![],
        },
    ];

    let mc_options = MultipleChoiceOptions { options };

    // get current timestamp
    let proposal_time = app.block_info().time;

    let proposal_id = make_proposal(&mut app, &proposal_module, CREATOR_ADDR,
mc_options);

    app.update_block(next_block);

    // someone votes enough to pass the proposal
    app.execute_contract(
        Addr::unchecked(CREATOR_ADDR),
        proposal_module.clone(),
        &ExecuteMsg::Vote {
            proposal_id,
            vote: MultipleChoiceVote { option_id: 0 },
            rationale: None,
        },
        &[],
    )
    .unwrap();

    app.update_block(next_block);

    // proposal enters veto timelock
    let expected_end_time =
proposal_time.plus_seconds(604800).plus_seconds(604800);

    let prop = query_proposal(&app, &proposal_module, proposal_id);
    assert_eq!(
        prop.proposal.status,
        Status::VetoTimelock { expiration: cw_utils::Expiration::AtTime(
expected_end_time)  }
    );

    // finish veto timelock
    app.update_block(|block| block.time = expected_end_time);

    // random user cannot execute due to only_members_execute
    app.execute_contract(
        Addr::unchecked("random"),
        proposal_module.clone(),
        &ExecuteMsg::Execute { proposal_id: 1 },
        &[],
    )
    .unwrap_err();

    // vetoer not part of DAO, but can execute proposals with
only_members_execute
    app.execute_contract(
        Addr::unchecked(vetoer),
        proposal_module.clone(),
        &ExecuteMsg::Execute { proposal_id: 1 },
        &[],
    )
    .unwrap_err();

}
```