certora

Reflector Security Assessment v1

October 2024

Prepared for
Reflector DAO

certora

Table of content

Project Summary

Project Scope

Project Name	Repository (link)	Latest Commit Hash	Platform	Comment
Reflector DAO Contract	https://github.com/reflector-network/reflector-dao-contract	a05dc7f	Stellar	Audit version
Reflector DAO Contract		d889dc1	Stellar	Fixed version
Reflector Subscription Contract	https://github.com/reflector-network/reflector-subscription-contract	773ea7b	Stellar	Audit version
Reflector Subscription Contract		3353668	Stellar	Fixed version

The scope includes all files under `src/` in both repositories.

Project Overview

This document describes the specification and verification of the Reflector DAO Contract and Reflector Subscription Contract using manual code review. The work was undertaken from September 25, 2024, to October 10, 2024.

The Certora team performed a manual audit of all contracts in the scope. During the audit, the team discovered bugs in the contract code, as listed on the following page. In addition to the manual audit, the Certora team wrote formal rules and verified them using the Certora Prover, as listed below.

We have verified the fixes that are present in this commit hashes in the above table.

Findings Summary

The table below summarizes the findings of the review, including type and severity details.

Severity	Discovered	Confirmed	Fixed
Critical	0	0	0
High	0	0	0
Medium	0	0	0
Low	5	5	3
Informational	2	2	2
Total	**7**	**7**	**5**

Severity Matrix

Impact	High	Medium	High	Critical
	Medium	Low	Medium	High
	Low	Low	Low	Medium
		Low	Medium	High
		Likelihood		

Detailed Findings

Low-Severity Issues

L-01 Security rules do not adhere to the 'least privilege' principle

Severity: **Low**	Impact: **Medium**	Likelihood: **Low**
Files: Multiple locations	Category: Design, Key Management, Access Control	Status: Acknowledged, will not fix

Description: If upgradability is included as a feature in a smart contract, it is imperative that proper key maintenance and role-based security policy safeguards will be implemented, i.e., the admin key should not be used for standard daily maintenance operations and should not have to be 'hot'. This is not the case in Reflector's Subscription and DAO contracts where the admin role has upgradability privileges but is the only privileged role which is needed for basic tasks such as publishing trigger events and setting fees (in Subscription) or approving ballots (in DAO).

Impact: If the admin key is ever compromised, both the subscription and DAO contracts are in peril.

Recommendation: We suggest that the current admin function would be split into several less powerful roles.

Customer's response: We discussed this internally with members of the teams that run the DAO. The consensus is that we don't want to add any additional complexity to the DAO contract, which will be non-upgradeable from day one, as this may backfire in the future.

As for the Subscriptions contract (which is already deployed), we are going to assess different approaches with the aim to adopt a more fine-grained security privileges approach within the next 6 months.

L-02 Missing Events

Severity: **Low**	Impact: **Low**	Likelihood: **Low**
Files: reflector-dao-contract/src/lib.rs, reflector-subscription-contract/src/lib.rs	Category: Events	Status: Fixed

Description: The following functions make substantial changes to the state but do not emit events:

- SubscriptionContract.config
- SubscriptionContract.set_fee
- SubscriptionContract.update_contract
- DAOContract.update_dao_balance
- DAOContract.set_dao_balance
- DAOContract.create_ballot

Impact: Off-chain monitoring of important events becomes more difficult.

Recommendation: add the missing events.

Customer's response: Fixed

L-03 Privileged Address and Token Address is set without confirmation

Severity: **Low**	Impact: **Low**	Likelihood: **Low**
Files: Multiple locations	Category: Input Validation	Status: Acknowledged, will not fix

Description: In both the Subscription and the DAO contract, setting the admin role is done without a challenge-response mechanism, and setting the token address does not include a check that the value of the Address is indeed valid:

```
pub fn config(e: Env, config: ContractConfig) {

    // check admin permissions

    config.admin.require_auth();

    // can be executed only once

    if e.is_initialized() {

        e.panic_with_error(Error::AlreadyInitialized);

    }

    // validate the funding amount

    if config.amount <= 0 {

        e.panic_with_error(Error::InvalidAmount);

    }

     // save the configuration
```

```
    e.set_admin(&config.admin); //@audit (LOW) admin should be two-step

    e.set_token(&config.token); //@audit (LOW) token should include on-chain verification.
```

Recommendation: In regard to the admin address (or any other privileged address that may exist in the contract), we recommend adding a 2-step verification process for setting the admin address (for a sample implementation in the context of Ethereum and Solidity, see OpenZeppelin's Ownable2Step.sol). In regard to the token, we suggest to add a call to some standard SEP-41 function signatures to ensure the address is correct.

Customer's response: See L-01.

L-04 Unchecked arithmetic

Severity: **Low**	Impact: **Low**	Likelihood: **Low**
Files: Multiple locations	Category: Arithmetic	Status: Fixed

Description: In many places in the contract, unchecked arithmetic operations are used. Examples are:

```
pub fn retract_ballot(e: Env, ballot_id: u64) {

    // …

        BallotStatus::Rejected => (ballot.deposit * 75) / 100,

        // …



            (ballot.deposit * 125) / 100

          //…

    update_dao_balance(&e, &(-refunded));

}
```

```
pub fn vote(e: Env, ballot_id: u64, accepted: bool) {

    //…
```

```
        // calculate the amount of DAO tokens to burn

        let burn_amount = match new_status {

            BallotStatus::Rejected => (ballot.deposit * 25) / 100,

            BallotStatus::Accepted => ballot.deposit,

            _ => e.panic_with_error(Error::BallotClosed),

        };

        // …

        update_dao_balance(&e, &(-burn_amount));

        // …
```

```
fn update_available_balance(e: &Env, address: &Address, amount: &i128) {

   let balance = e.get_available_balance(address);

   e.set_available_balance(address, balance + amount);

}
```

```
fn update_dao_balance(e: &Env, amount: &i128) {

    let dao_balance = e.get_dao_balance(); //@audit when first called during initialization is the
one place where we should actually return zero!

   e.set_dao_balance(dao_balance + amount);

}
```

In DAOContract.unlock:

```
// the amount a single operator would get
let unlock_per_operator: &i128 = &(operators_unlocked / operators.len() as i128);
```

Impact: In the worst case scenario, such operations could overflow, underflow, or division by zero. Even if not, it is better to revert with a clear, specific error message.
Recommendation: use the checked variant of addition/subtraction with meaningful error messages.
Customer's response: Fixed

L-05 It is possible to unlock prizes unevenly

Severity: **Low**	Impact: **Low**	Likelihood: **Low**
Files: Multiple locations	Category: Input Validation	Status: Fixed

Description: since there is no input validation which checks that the list of operators in the unlock function

```
pub fn unlock(e: Env, developer: Address, operators: Vec<Address>)
```

contains only unique values, thus it is possible for the admin to distribute multiple portions of the reward (or even the entire operator reward) to a single address.

Impact: It is possible for the admin to distribute multiple prizes to the same party by mistake or intention, bypassing the intended unlock restriction.

Recommendation: add input validation to ensure such a case would not occur

Customer's response: Fixed

Informational-Severity Issues

I-01 Typo

Severity: **Informational**	Impact: **Low**	Likelihood: **Low**
Files: DAO contract env_extensions.rs	Category: Code Quality	Status: Fixed

Description: In the function

```
fn set_last_unlock(&self, last_uplock: u64) {
    get_instance_storage(&self).set(&LAST_UNLOCK, &last_uplock);
}
```

The parameter "last_uplock" should probably be named "last_unlock".

Customer's response: Fixed

I-02 Missing comments

Severity: **Informational**	Impact: **Low**	Likelihood: **Low**
Files:	Category: Documentation	Status: Fixed

Description: In both constants

```
// 0.24% weekly distribution

const OPERATORS_SHARE: i128 = 24; //@audit (INFO) should record in a comment that we are working
with 10000 = 100%

// 0.06% weekly distribution

const DEVELOPERS_SHARE: i128 = 6; //@audit (INFO) should record that we are working with 10000 =
100%
```

It is better if we record the fact that 100% = 10000 for clarity. Similar in the function

```
// calculate percentage from a given amount

fn calc_percentage(value: i128, percentage: i128) -> i128 {

   (value * percentage) / 10000 //@audit (INFO) this 10000 should be a constant

}
```

It is better if we add a constant ONE_HUNDRED_PERCENT = 10000 and a comment instead of placing the number in the code.

Customer's response: Fixed

certora

Formal Verification

Verification Notations

Formally Verified	The rule is verified for every state of the contract(s), under the assumptions of the scope/requirements in the rule.
Formally Verified After Fix	The rule was violated due to an issue in the code and was successfully verified after fixing the issue
Violated	A counter-example exists that violates one of the assertions of the rule.

Formal Verification Properties

In both verification tasks below, we verify the smart contract code (lib.rs) against our model of the Soroban host environment. This implementation of Soroban's host environment is in the trusted computing base (TCB): we do not verify that our host implementation matches Soroban's implementation, nor do we verify that it is correct wrt any formal specification. We focus only on the correctness of the smart contract code.

Additional assumptions for each contract will be listed separately below.

For both contracts, the specs are in the `src/certora_specs` directory. There are configuration files in the `conf` directory that we used for running the prover. All of this code is in the `certora` branch.

Reflector-DAO

Link: [reflector-dao-contract/src/lib.rs]():
All our specs are available here:
https://github.com/Certora/reflector-dao-contract/tree/certora

Module General Assumptions and Scope

1. Loops were unrolled at most 4 times (iterations)
2. The scope of this verification effort is src/lib.rs

First, we report on several state machine and access control properties that we expect the system to pass. P-01 lists all rules we have verified that we classify as state machine properties. We then report on the invariant that the value of LAST_BALLOT_ID is strictly increasing in P-02, and finally on the invariant that `available()` and `get_dao_balance()` cannot be negative in P-03.

Contract Properties

P-01 State machine and access control properties	
Status: Verified	

Rule Name	Status	Description	Link to rule report
certora_config_can_only_be_called_once	Verified	config() can only be called once	Report
certora_config_deposit_not_negative certora_set_deposit_must_be_non_nega	Verified	deposit is not set to a negative value by some functions	Report

tive			
certora_create_ballot_must_be_initiator	Verified	A ballot can only be created by its initiator	Report
certora_retract_ballot_must_be_initiator	Verified	A ballot can only be retracted by its initiator	Report
certora_retracted_ballot_cannot_be_voted	Verified	Cannot vote on a retracted ballot	Report
certora_retract_ballot_can_only_be_called_once	Verified	A ballot can only be retracted once	Report
certora_set_deposit_must_be_admin	Verified	deposit can only be set by an admin	Report
certora_unlock_must_be_admin	Verified	unlock() can only be called by an admin	Report
certora_vote_must_be_admin	Verified	vote() can only be called by an admin	Report
certora_retracted_ballot_cannot_be_retracted certora_accepted_ballot_cannot_be_retracted certora_retracted_ballot_cannot_be_voted certora_accepted_ballot_cannot_be_voted	Verified	Accepted and Retracted are terminal states	Report

certora_voted_ballot_was_draft certora_retracted_ballot_was_draft_or_rejected	Verified	If pre-state is Draft, post-state is either Accepted or Rejected The pre-estate of Retracted is either Draft or Rejected	Report

P-02 Invariant: The value of LAST_BALLOT_ID is strictly increasing

Status: Verified

Rule Name	Status	Description	Link to rule report
certora_ballot_id_increasing	Verified	the value of LAST_BALLOT_ID is strictly increasing	Report

P-03 Invariant: get_dao_balance(), available() are nonnegative

Status: Verified

Rule Name	Status	Description	Link to rule report
certora_invariant_balances_not_negative_config	Verified	get_dao_balance(), available() are nonnegative	Report

certora_invariant_balances_not_negative_set_deposit certora_invariant_balances_not_negative_unlock certora_invariant_balances_not_negative_available certora_invariant_balances_not_negative_claim certora_invariant_balances_not_negative_create_ballot certora_invariant_balances_not_negative_get_ballot certora_invariant_balances_not_negative_retract_ballot certora_invariant_balances_not_negative_vote			

Reflector-subscription

Link: [reflector-subscription-contract/src/lib.rs]
All our specs are available here:
https://github.com/Certora/reflector-subscription-contract/tree/certora

Module General Assumptions and Scope

- Loops were unrolled at most 4 times (iterations)
- The scope of this verification effort is src/lib.rs

- As part of this verification task, we made the following changes to the code
  - We have a sound, mock implementation of Token for which the body of each function is summarized to be `nondet`. This means that we prove the properties below without making any assumptions about the behavior of these functions. We use Rust's conditional compilation feature to use the mock implementation for verification purposes. You can see where we use them in `lib.rs` by searching for the annotation `#[cfg(not(feature = "certora"))]`.
  - We also additionally used `GhostMap<u64, u64>`, which is a ghost variable used only for verification purposes. This variable is named `GHOST_FEES_CHARGED` and is used to track the fee charged. You can see its usage in `lib.rs` and in the specs.

Contract Properties

P-01 config() can only be called once	
Status: Verified	

Rule Name	Status	Description	Link to rule report
certora_config_only_once_a certora_config_only_once_b	Verified	The contract is initialized after calling config() If the contract is already initialized then config() panics.	Report

P-02 Only admin may charge retention fees	
Status: Verified	

Rule Name	Status	Description	Link to rule report

certora_only_admin_charge_retention_fee certora_only_admin_charge_retention_fee_sanity	Verified	charge() should fail if the caller is not admin charge() should succeed if the caller is admin	Report

P-03 Properties about deposit

Status: Verified

Rule Name	Status	Description	Link to rule report
certora_deposit_changes_subscription_status_correctly certora_deposit_owner	Verified	`deposit` should correctly update the status of a subscription Only the `from` address can deposit funds	Report

P-04 Properties about charge

Status: Verified

Rule Name	Status	Description	Link to rule report
certora_charge_suspends_subscription_correctly	Verified	`charge` should correctly update the status of a subscription	Report

P-04 Calling create must activate a subscription

Status: Verified

Rule Name	Status	Description	Link to rule report
certora_create_activates_subscription	Verified	`create` must set the status of a subscription to Active.	Report

P-05 Properties about cancel

Status: Verified

Rule Name	Status	Description	Link to rule report
certora_cancel_subscription_success certora_cancel_non_owner certora_cancel_inactive certora_cancel_invalidates_charge certora_cancel_invalidates_deposit certora_cancel_invalidates_cancel certora_cancel_invalidates_get_subscription		`cancel` must correctly affect other subsequent function calls.	Report

Disclaimer

The Certora Prover takes a contract and a specification as input and formally proves that the contract satisfies the specification in all scenarios. Notably, the guarantees of the Certora Prover

are scoped to the provided specification and the Certora Prover does not check any cases not covered by the specification.

Even though we hope this information is helpful, we provide no warranty of any kind, explicit or implied. The contents of this report should not be construed as a complete guarantee that the contract is secure in all dimensions. In no event shall Certora or any of its employees be liable for any claim, damages, or other liability, whether in an action of contract, tort, or otherwise, arising from, out of, or in connection with the results reported here.

About Certora

Certora is a Web3 security company that provides industry-leading formal verification tools and smart contract audits. Certora's flagship security product, Certora Prover, is a unique SaaS product that automatically locates even the most rare & hard-to-find bugs on your smart contracts or mathematically proves their absence. The Certora Prover plugs into your standard deployment pipeline. It is helpful for smart contract developers and security researchers during auditing and bug bounties.

Certora also provides services such as auditing, formal verification projects, and incident response.