Rarimo - Voting contracts

Rarimo

HALBORN

Rarimo - Voting contracts - Rarimo

Prepared by: HALBORN

Last Updated 03/12/2024

Date of Engagement by: March 28th, 2024 - March 6th, 2024

Summary

100% OF ALL REPORTED FINDINGS HAVE BEEN ADDRESSED

ALL FINDINGS	CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL
3	0	0	0	2	1

TABLE OF CONTENTS

1. Introduction

Rarimo engaged our team to undertake an in-depth security analysis of their smart contract ecosystem. This assessment commenced with the objective of thoroughly examining the security framework of Rarimo's smart contracts. Our aim was to uncover any potential vulnerabilities, scrutinize the existing security protocols, and deliver practical recommendations to bolster the security and operational efficacy of Rarimo's smart contract architecture. The scope of our assessment was rigorously confined to the smart contracts provided for our evaluation, guaranteeing a concentrated and comprehensive investigation into the security dimensions of the contracts in question.

2. Assessment Summary

Halborn was provided about one week for the engagement and assigned one full-time security engineer to review the security of the smart contracts in scope. The engineer is a blockchain and smart contract security expert with advanced penetration testing and smart contract hacking skills, and deep knowledge of multiple blockchain protocols.

The purpose of this assessment is to:

- Ensure that smart contract functions operate as intended

- Identify potential security issues with the smart contracts

During our comprehensive security assessment, we focused on evaluating critical aspects of the smart contract ecosystem to identify potential vulnerabilities, validate the robustness of the implemented security measures, and ensure alignment with best practices in smart contract development. The assessment targeted various components and functionalities, as detailed below:

Contract Upgradability And Initializers

- **Missing** *disableInitializers Call: Identified a critical oversight in several upgradable contracts (LightweightState,* `ZKPQueriesStorage`*,* `QueryMTPValidator`*,* `QueryValidator`*,* `BaseVerifier`*, and* `RegisterVerifier`*) where the* `disableInitializers` function was not called in their constructors, exposing them to risks of unauthorized re-initialization.

Access Control Mechanisms

- **Two-Step Ownership Transfer**: Recommended the adoption of a two-step ownership transfer process, such as OpenZeppelin's `Ownable2Step`, for critical contracts to enhance security during ownership changes and prevent potential mishaps in control transfers.

Validation And Handling Of Inputs

- **Lack of Input Validation**: Noted the absence of input validation for pool types in the `VotingRegistry` contract, leading to potential operational inefficiencies and management complexities due to the acceptance of invalid or unintended strings as pool types.

- **Potential Deadlock Due to Empty Pool Type**: Highlighted a specific vulnerability in the `VotingRegistry` contract where the lack of validation for non-empty `poolType_` strings could lead to a deadlock situation, particularly since other parts of the contract rely on this validation for proper functionality.

Merkle Tree Implementation And Collision Resistance

- **PoseidonSMT and SparseMerkleTree Utilization**: Reviewed the implementation of `PoseidonSMT`, which leverages the `SparseMerkleTree` library for efficient and secure data storage. Special attention was given to the handling of potential hash collisions and the robustness of the tree's integrity mechanisms.

Voting Mechanism Security

- **Unique Voting Validation**: Analyzed the voting mechanism, ensuring the system correctly validates unique votes and prevents double voting through the use of zk-SNARKs and nullifier checks.

Factory Pattern And Proxy Deployment

- **Proxy Pool Deployment and Registration**: Evaluated the `VotingFactory` contract's logic for pool deployment and registration, focusing on the security and reliability of proxy contract creation and the subsequent linking of these proxies to their implementations.

Recommendations And Remediation Actions

Our findings highlight the need for rigorous input validation, enhanced access control mechanisms, and the securing of contract initialization processes to mitigate potential security vulnerabilities. Specific recommendations include the implementation of two-step ownership transfer processes, ensuring the presence of `_disableInitializers` in all upgradable contracts, and adopting stringent validation checks for all inputs, especially those related to pool types and contract initialization parameters.

This summary aims to provide a high-level overview of the vulnerabilities identified during the assessment, along with our recommendations for enhancing the security and integrity of the smart contract ecosystem. It is crucial to address these findings promptly to safeguard against potential exploits and to maintain the trustworthiness of the system.

3. Test Approach And Methodology

Halborn performed a combination of manual and automated security testing to balance efficiency, timeliness, practicality, and accuracy regarding the scope of this assessment. While manual testing is recommended to uncover flaws in logic, process, and implementation; automated testing techniques help enhance coverage of the code and can quickly identify items that do not follow the security best practices. The following phases and associated tools were used during the assessment:

- Research into architecture and purpose.

- Smart contract manual code review and walkthrough.

- Graphing out functionality and contract logic/connectivity/functions (`solgraph`).

- Manual assessment of use and safety for the critical Solidity variables and functions in scope to identify any arithmetic related vulnerability classes.

- Manual testing by custom scripts.

- Testnet deployment (Foundry).

4. RISK METHODOLOGY

Every vulnerability and issue observed by Halborn is ranked based on **two sets** of **Metrics** and a **Severity Coefficient**. This system is inspired by the industry standard Common Vulnerability Scoring System.

The two **Metric sets** are: **Exploitability** and **Impact**. **Exploitability** captures the ease and technical means by which vulnerabilities can be exploited and **Impact** describes the consequences of a successful exploit.

The **Severity Coefficients** is designed to further refine the accuracy of the ranking with two factors: **Reversibility** and **Scope**. These capture the impact of the vulnerability on the environment as well as the number of users and smart contracts affected.

The final score is a value between 0-10 rounded up to 1 decimal place and 10 corresponding to the highest security risk. This provides an objective and accurate rating of the severity of security vulnerabilities in smart contracts.

The system is designed to assist in identifying and prioritizing vulnerabilities based on their level of risk to address the most critical issues in a timely manner.

4.1 EXPLOITABILITY

ATTACK ORIGIN (AO):

Captures whether the attack requires compromising a specific account.

ATTACK COST (AC):

Captures the cost of exploiting the vulnerability incurred by the attacker relative to sending a single transaction on the relevant blockchain. Includes but is not limited to financial and computational cost.

ATTACK COMPLEXITY (AX):

Describes the conditions beyond the attacker's control that must exist in order to exploit the vulnerability. Includes but is not limited to macro situation, available third-party liquidity and regulatory challenges.

METRICS:

EXPLOITABILIY METRIC (m_e)	METRIC VALUE	NUMERICAL VALUE
Attack Origin (AO)	Arbitrary (AO:A) Specific (AO:S)	1 0.2
Attack Cost (AC)	Low (AC:L) Medium (AC:M) High (AC:H)	1 0.67 0.33

EXPLOITABILIY METRIC (m_e)	METRIC VALUE	NUMERICAL VALUE
Attack Complexity (AX)	Low (AX:L) Medium (AX:M) High (AX:H)	1 0.67 0.33

Exploitability E is calculated using the following formula:

$$E = \prod m_e$$

4.2 IMPACT

CONFIDENTIALITY (C):

Measures the impact to the confidentiality of the information resources managed by the contract due to a successfully exploited vulnerability. Confidentiality refers to limiting access to authorized users only.

INTEGRITY (I):

Measures the impact to integrity of a successfully exploited vulnerability. Integrity refers to the trustworthiness and veracity of data stored and/or processed on-chain. Integrity impact directly affecting Deposit or Yield records is excluded.

AVAILABILITY (A):

Measures the impact to the availability of the impacted component resulting from a successfully exploited vulnerability. This metric refers to smart contract features and functionality, not state. Availability impact directly affecting Deposit or Yield is excluded.

DEPOSIT (D):

Measures the impact to the deposits made to the contract by either users or owners.

YIELD (Y):

Measures the impact to the yield generated by the contract for either users or owners.

METRICS:

IMPACT METRIC (m_I)	METRIC VALUE	NUMERICAL VALUE
Confidentiality (C)	None (I:N) Low (I:L) Medium (I:M) High (I:H) Critical (I:C)	0 0.25 0.5 0.75 1

IMPACT METRIC (m_I)	METRIC VALUE	NUMERICAL VALUE
Integrity (I)	None (I:N) Low (I:L) Medium (I:M) High (I:H) Critical (I:C)	0 0.25 0.5 0.75 1
Availability (A)	None (A:N) Low (A:L) Medium (A:M) High (A:H) Critical (A:C)	0 0.25 0.5 0.75 1
Deposit (D)	None (D:N) Low (D:L) Medium (D:M) High (D:H) Critical (D:C)	0 0.25 0.5 0.75 1
Yield (Y)	None (Y:N) Low (Y:L) Medium (Y:M) High (Y:H) Critical (Y:C)	0 0.25 0.5 0.75 1

Impact I is calculated using the following formula:

$$I = max(m_I) + \frac{\sum m_I - max(m_I)}{4}$$

4.3 SEVERITY COEFFICIENT

REVERSIBILITY (R):

Describes the share of the exploited vulnerability effects that can be reversed. For upgradeable contracts, assume the contract private key is available.

SCOPE (S):

Captures whether a vulnerability in one vulnerable contract impacts resources in other contracts.

METRICS:

SEVERITY COEFFICIENT (C)	COEFFICIENT VALUE	NUMERICAL VALUE
Reversibility (r)	None (R:N) Partial (R:P) Full (R:F)	1 0.5 0.25
Scope (s)	Changed (S:C) Unchanged (S:U)	1.25 1

Severity Coefficient C is obtained by the following product:

$$C = rs$$

The Vulnerability Severity Score S is obtained by:

$$S = min(10, EIC * 10)$$

The score is rounded up to 1 decimal places.

SEVERITY	SCORE VALUE RANGE
Critical	9 - 10
High	7 - 8.9
Medium	4.5 - 6.9
Low	2 - 4.4
Informational	0 - 1.9

5. SCOPE

FILES AND REPOSITORY

(a) Repository: voting-contracts

(b) Assessed Commit ID: 2a6d61e

(c) Contracts in scope:

- VotingRegistry.sol
- VotingFactory.sol
- Voting.sol
- Registration.sol
- utils/PoseidonSMT.sol
- iden3/LightweightState.sol
- iden3/verifiers/RegisterVerifier.sol
- iden3/validators/QueryValidator.sol
- iden3/ZKPQueriesStorage.sol
- iden3/verifiers/BaseVerifier.sol
- iden3/validators/QueryMTPValidator.sol

Out-of-Scope:

REMEDIATION COMMIT ID:

- 8
- 3383d52

Out-of-Scope: New features/implementations after the remediation commit IDs.

6. ASSESSMENT SUMMARY & FINDINGS OVERVIEW

CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL
0	0	0	2	1

SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
HAL-02 - LACK OF TWO-STEP OWNERSHIP TRANSFER	Low	RISK ACCEPTED - 03/10/2024
HAL-03 - LACK OF DISABLEINITIALIZERS IN UPGRADEABLE CONTRACTS	Low	SOLVED - 03/10/2024
HAL-01 - POTENTIAL DEADLOCK DUE TO EMPTY POOL TYPE	Informational	SOLVED - 03/10/2024

7. FINDINGS & TECH DETAILS

7.1 (HAL-02) LACK OF TWO-STEP OWNERSHIP TRANSFER

// LOW

Description

The `VotingRegistry` contract, as it stands, utilizes the `OwnableUpgradeable` contract from OpenZeppelin for its ownership management, which implements a single-step ownership transfer mechanism. This means that the ownership of the contract can be transferred in a single transaction without any requirement for the new owner to accept the ownership. While this approach simplifies the process of transferring ownership, it also introduces a risk where critical functionalities, such as setting new implementations or upgrading the contract, could inadvertently be left without an owner or transferred to an unintended address. This situation could arise from a simple operational mistake or as a result of a targeted phishing attack. The absence of a two-step ownership transfer process, where the new owner must accept the ownership before the transfer is finalized, removes a critical layer of security that could prevent unauthorized or accidental changes to the contract's ownership.

BVSS

AO:S/AC:L/AX:L/C:N/I:C/A:C/D:N/Y:N/R:N/S:C (3.1)

Recommendation

To mitigate the risks associated with the single-step ownership transfer mechanism and enhance the security posture of the `VotingRegistry` contract, it is recommended to integrate a two-step ownership transfer process. This can be achieved by adopting the `Ownable2Step` pattern, an extension of the OpenZeppelin `Ownable` contract, which introduces an intermediary step where the new owner must explicitly accept ownership before the transfer is completed. This mechanism ensures that the transfer of ownership is intentional and agreed upon by both parties, significantly reducing the risk of accidental or malicious transfers. Implementing a two-step ownership transfer process will add an additional layer of security, especially critical for contracts like `VotingRegistry` that manage sensitive functionalities such as contract upgrades and implementation settings. This change requires minimal modification to the existing contract structure but provides a significant improvement in terms of operational safety and security.

Remediation Plan

RISK ACCEPTED: The transition to a two-step ownership process may limit potential future integrations, for example, in cases where ownership needs to be transferred to the DAO.

7.2 (HAL-03) LACK OF DISABLEINITIALIZERS IN UPGRADEABLE CONTRACTS

// LOW

Description

In the context of upgradeable smart contracts using OpenZeppelin's upgradeable patterns, it's crucial to ensure that initializers cannot be re-invoked after deployment, to prevent unauthorized or unintended re-initialization which could lead to severe security vulnerabilities, such as unauthorized control or self-destruction of the contract, affecting all dependent proxies. The contracts `LightweightState`, `ZKPQueriesStorage`, `QueryMTPValidator`, `QueryValidator`, `BaseVerifier`, and `RegisterVerifier` were identified as not calling `_disableInitializers` in their constructors. This omission leaves these contracts vulnerable to potential phishing attacks or allows for scenarios where, in the presence of an open `delegatecall`, the contract could be forced to self-destruct, rendering all associated proxies non-functional. This oversight is particularly critical given that these contracts are part of a system designed to be upgradeable, where the integrity and continuity of the contract logic are paramount.

BVSS

AO:S/AC:L/AX:L/C:N/I:H/A:H/D:N/Y:N/R:N/S:C (2.3)

Recommendation

To mitigate these risks and enhance the security posture of the upgradeable contract ecosystem, it is recommended that all contracts, particularly those identified, incorporate the `_disableInitializers` function call within their constructors or initialization logic. This practice should be uniformly applied to ensure that once an upgradeable contract is deployed, its initialization function cannot be called more than once. This will effectively seal the contract against unauthorized re-initialization attempts, which could compromise the contract's integrity and the security of the system.
In scenarios where contracts inherit from others, care must be taken to ensure that `_disableInitializers` is called appropriately to prevent re-initialization while avoiding redundant or conflicting calls in the inheritance chain. A strategic approach would be to place the `_disableInitializers` call in the most derived contract that is intended for deployment, ensuring that base contracts do not inadvertently disable initializers for contracts that extend them. This approach helps maintain the flexibility and security of the upgradeable contracts framework, providing robust protection against a range of attacks targeting the re-initialization vulnerability.

Remediation Plan

SOLVED: The issue was solved by the Rarimo team.

Remediation Hash

https://github.com/rarimo/voting-contracts/pull/8

7.3 (HAL-01) POTENTIAL DEADLOCK DUE TO EMPTY POOL TYPE

// INFORMATIONAL

Description

The `VotingRegistry` contract's `setNewImplementations` function allows for the setting of new implementations for given pool types without validating that the provided `poolType_` strings are non-empty. This lack of validation poses a risk of creating a deadlock scenario, particularly because other parts of the contract, such as `bindVotingToRegistration`, do enforce a check for non-zero length of pool type strings (using `bytes(_typeByPool[registration_]).length > 0`). If an empty string is used as a pool type in `setNewImplementations`, it may lead to situations where certain functionalities cannot be executed as expected due to the inability to match the empty pool type with any meaningful category or implementation, effectively creating inconsistencies in the contract's logic and potentially hindering the management and categorization of pools.

BVSS

AO:S/AC:L/AX:L/C:N/I:H/A:N/D:N/Y:N/R:N/S:U (1.5)

Recommendation

To address this issue and prevent the potential for deadlocks or inconsistencies within the `VotingRegistry` contract, it is recommended to introduce a validation check within the `setNewImplementations` function to ensure that none of the `poolTypes_` elements are empty strings. This can be achieved by adding a simple condition that iterates over the `poolTypes_` array and checks each element for a non-zero length before proceeding with the assignment of new implementations. Such a validation mechanism will align with the existing checks in other parts of the contract, ensuring consistency across the contract's logic and preventing the registration of implementations under invalid or empty pool types. Implementing this recommendation will enhance the contract's reliability by ensuring that all pool types have meaningful and valid identifiers, thereby facilitating more efficient management and operation of the pool categorization functionality.

Remediation Plan

SOLVED: The issue was solved by the Rarimo team.

Remediation Hash

https://github.com/rarimo/voting-contracts/commit/3383d520feb5d27be811e02a2bebb196481395e8

8. REVIEW NOTES

PoseidonSMT

- It is using the `SparseMerkleTree` library from `solarity`.
- The `__PoseidonSMT_init` does call the `setHashers` which sets the hash for 2 and 3 elements to the `poseidon` implementation for that amount of elements.
- The internal `_add` function does hash the value given with `poseidon` (from registration the `commitment_`) and uses this as the key.
- Does expose some public functions such as `getProof`, `getRoot` and `getNodeByKey` for accessibility.

Registration

- During `register` the `commitment_` is being used on the `PoseidonSMT` contract to generate a key using poseidon. Mathematically speaking, there is the possibility that two different `commitment` could lead to the same poseidon hash. However, data length on the `commitment` is the same as the poseidon hash entropy which reduces the probablity. Moreover, in the world case scenario were that collision would happen, `SparseMerkleTree` would revert with "the key already exists".
- `_validateRegistrationParams` does verify period and timestamp.
- `getRegistrationStatus` is a standard state getter based on `registrationInfo` information. `COMMITMENT` period is stated to be between `commitmentStartTime` and `commitmentEndTime` (the former inclusive).

Voting

- The init function does allow setting the `candidates` for the voting phase.
- `getProposalStatus` is a standard state getter based on `votingInfo` information. `PENDING` period is stated to be between `votingStartTime` and `votingEndTime` (the former inclusive).
- The `vote` function will verify the parameters using the ZK verifier. It will also make sure that the `nullifiers` is not present already, allowing only one vote per verifier nullifier. It will also count the candidates an total votes, verifying if the candidate exists as per the contract deployment specification.

VotingFactory

- `_createPool` does `_deploy` and then `_register` the deployed address.
- The `_deploy` should be checking for `getPoolImplementation` address different than 0 to prevent unhandled errors when calling the implementation on that address.
- Both registration and voting are using the same `VotingRegistry` to track pools. Which can be risky on type confusion (Using a type for a different deployment purpose).
- When registering with a salt, the code does combine the actual provided parameter salt and the sender address into a hash. This is the salt used to deploy the proxy contract with create2.
- `_createPoolWithSalt` will use the `_deploy2` to instantiate the contract with empty data. It will then call the proxy with the initial data. There is no reason to separate both actions and does require the use of `verifyCallResult` to verify call result.
- The owner for uups upgradability is the same as the `votingRegistry` owner.
- The internal `_register` will call `addProxyPool` on the `votingRegistry`

VotingRegistry

- `getPoolImplementation` is used by teh factory to fetch the implementation address for a given type, including both registration and voting deployments.

- `setNewImplementations` is the setter, batchable.
- `addProxyPool` adds to the enumerable mappings the new pool address.

LightweightState

- `changeSigner` will verify a signature and change the `signer`.
- Will allow changing the source address for signature validation.
- `signedTransitState` will allow setting a new merkle root for the current state (ĝist`and identities).
- `verifyStatesMerkleData` should probably verify that the `computedRoot_` is the current root if we want to make sure the merkle tree is active.

ZKPQueriesStorage

- Will be storing `QueryMTPValidator` addresses.
- Does initialise the `lightweightState` address. Used on integration contracts.
- `setZKPQuery` will allow overriding previous `_queriesInfo`. The `removeZKPQuery` function will make sure to remove and clean the storage.
- `getQueryHashRaw` will calculate a hash given some parameters for the query.
- The contract getters should probably check if the `queryId_` parameter exist or not.

QueryValidator

- `setVerifier` and `setIdentitesStatesUpdateTime` can only be called by the owner.
- `verify` will first call the verifier `verifyProof` to check the ZK proof.
- `queryHash_` is verified with the parameter (both user controlled).
- The validation parameters, that were used for the ZK proof are compared with the `statesMerkleData`
- `_checkGistRoot` will verify that the `validationParams_.gistRoot` is stored under `lightweightState`.
- `_verifyStatesMerkleData` will verify that `issuerId`, `issuerState` and `createdAtTimestamp` do indeed verify the merkle proofs.
- If the computed root is not the current `identitiesStatesRoot` it will verify if the current `identitiesStatesRoot` timestamp has expired as per `identitesStatesUpdateTime` period (set on this contract). The root will be considered invalid then.
- `_checkGistRoot` used on the `verify` will make sure that merkle root is equal and none zero for the given proofs.

QueryMTPValidator

- Extends `QueryValidator` and adds some static getters to it.

BaseVerifier

- `setZKPQueriesStorage` and `updateAllowedIssuers` is only callable by the owner.
- `_transitState` will verify that the new timetamp is higher than previous. Also if the identity root does not exist. If it does exist, or timestamp is lower the code will not revert. `proof` is being used to verify the `newIdentitiesStatesRoot`.

RegisterVerifier

- Does extend `BaseVerifier`.
- The `RegisterProofInfo` struct does contain `registrationContractAddress` which is being used to verify the caller under `onlyVoting` modifier.

- The `_verify` will check for parameters being valid and fill the `CircuitQuery` data and compute the verification hash. The used data is the isAdult flag and the `issuingAuthority` and `documentNullifier`, which is enough to verify a unique identity.
- `_validateRegistrationFields` will verify that the proofs are appropriated for the actual `queryValidator_` address.

Halborn strongly recommends conducting a follow-up assessment of the project either within six months or immediately following any material changes to the codebase, whichever comes first. This approach is crucial for maintaining the project's integrity and addressing potential vulnerabilities introduced by code modifications.