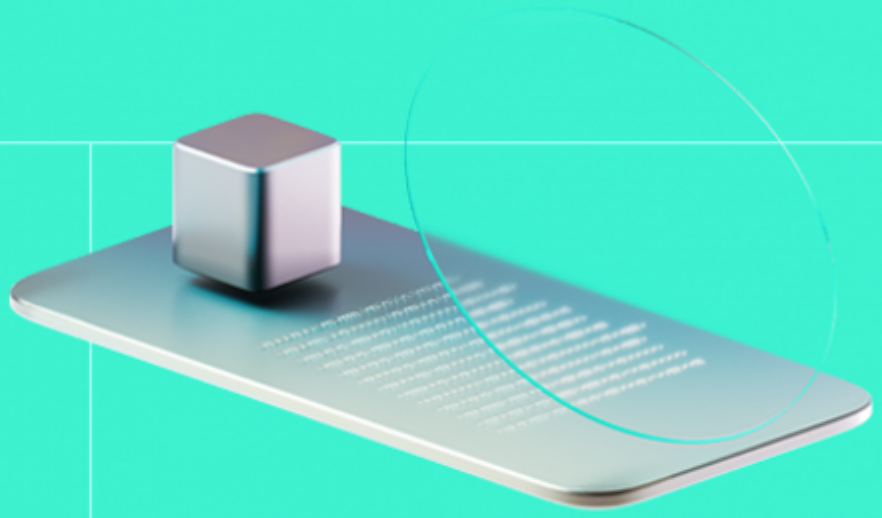
HACKEN

Smart Contract Code Review And Security Analysis Report



Customer: GOAT Network

Date: 04/11/2024

We express our gratitude to the GOAT Network team for the collaborative engagement that enabled the execution of this Smart Contract Security Assessment.

The Goat Network is the first BTC L2 solution that offers sustainable BTC yield. The `Locking` contract facilitates the validators' assets lockout, designed to be utilised by the consensus layer.

Document

Name	Smart Contract Code Review and Security Analysis Report for GOAT Network
Audited By	Grzegorz Trawinski, Nataliia Balashova
Approved By	Przemyslaw Swiatowiec
Website	https://www.goat.network/
Changelog	31/10/2024 - Preliminary Report, 04/11/2024 - Final Report
Platform	Goat Network
Language	Solidity
Tags	Consensus, L2, Validator
Methodology	https://hackenio.cc/sc_methodology

Review Scope

Repository	https://github.com/GOATNetwork/goat-contracts
Commit	6d68a950806eca27ca4265280976bd2d94514be1
Retest Commit	n/a

Audit Summary

The system users should acknowledge all the risks summed up in the risks section of the report

5	0	5	0
Total Findings	Resolved	Accepted	Mitigated

Findings by Severity

Severity	Count
Critical	0
High	0
Medium	2
Low	3

Vulnerability	Severity
F-2024-6870 - Reward Capping without Unclaimed Accrual	Medium
F-2024-6885 - The Approval Cannot Be Reverted	Medium
F-2024-6869 - The Locking Contract Lacks Two-Step Ownership Transfer	Low
F-2024-6881 - Lack of Compliance With EIP-712 Standard	Low
F-2024-6884 - The changeValidatorOwner Function Lacks Two-Step Ownership Transfer	Low

Documentation quality

- Functional requirements are partially provided.
- Technical description was not provided.
- White-paper is available.
- Economics beige-paper is available.

Code quality

- The code quality is sufficient.

Test coverage

Code coverage of the contracts in scope is nearly **90%** (branch coverage).

Table of Contents

System Overview

The `Locking` contract facilitates the validators' assets lockout, designed to be utilised by the consensus layer. The solution has the following contracts:

- Locking.sol - Allows approved validators to lock assets within the contract. Based on external processing, validator can later claim reward. Unlock asset functionality is also available. It is protected by the `RateLimiter`.
- pubkey.sol - a utility library that allows to calculate EthAddress and ConsAddress from public key.
- Executor.sol - contains constants representing fixed addresses of `Relayer` and `Locking` contracts.
- RateLimiter.sol - contains functionality extending access control possibilities by limiting transactions.

Privileged roles

- The owner of the `Locking` contract can enable claiming, fill contract with reward tokens, approve a validator, add new token, update token's weight, limit and threshold.
- The validator's owner within the `Locking` contract can lock and unlock tokens, update the validator's owner address, claim the rewards.
- The consensus guard can distribute rewards and unlocked tokens to the specified recipients.

Potential Risks

- **Interactions with External DeFi Protocols**: Dependence on external DeFi protocols inherits their risks and vulnerabilities. This might lead to direct financial losses if these protocols are exploited, indirectly affecting the audited project.
- **Single Points of Failure and Control**: The project is fully or partially centralized, introducing single points of failure and control. This centralization can lead to vulnerabilities in decision-making and operational processes, making the system more susceptible to targeted attacks or manipulation.
- **Centralized Oracles as Data Sources**: The protocol utilizes centralized oracles for external data inputs. Dependence on a singular or limited set of data sources can introduce accuracy and manipulation risks, potentially affecting the DApp's operations and decision-making processes.
- **Centralized Governance Mechanisms**: The smart contract system incorporates governance mechanisms that are not fully decentralized, potentially allowing a small number of participants to exert disproportionate influence over key decisions and changes, limiting community engagement and consensus.
- **External control over rewards distribution**: The external entity controls whether and when the reward will be distributed upon requesting the claim by the validator.
- **External control over tokens unlock**: The external entity controls whether and when the tokens will be unlocked upon requesting the unlock by the validator.

Findings

Vulnerability Details

F-2024-6870 - Reward Capping without Unclaimed Accrual - Medium

Description:

In the `distributeReward` function , if the requested reward amount (`goat`) is higher than the available reward (`remainReward`), the function caps `goat` to the `remainReward` level, effectively reducing the reward amount granted to the recipient.

```
function distributeReward(
        uint64 id,
        address recipient,
        uint256 goat,
        uint256 gasReward
    ) external override ConsensusGuard(id) {
        if (remainReward < goat) {
            goat = remainReward;
        }
        /.../
    }
```

However, this design leads to a situation where the difference between the requested `goat` and the capped `goat` is neither recorded nor retained in the unclaimed rewards. This means that the unclaimed reward balance does not reflect the amount initially earned but not fully distributed.

For instance, if a recipient earns 100 units of reward, but only 50 remain, `goat` is capped at 50, and the remaining 50 are not tracked in the `unclaimed` mapping. This can result in under-distribution where the recipient cannot later claim the difference if the `remainReward` balance is replenished.

Assets:

- contracts/locking/Locking.sol [https://github.com/GOATNetwork/goat-contracts/blob/main/contracts/locking/Locking.sol]

Status: Accepted

Classification

Impact:	3/5
Likelihood:	3/5
Exploitability:	Independent
Complexity:	Medium
Severity:	Medium

Recommendations

Remediation: It is recommended in the `distributeReward` function, to track any portion of the requested reward (`goat`) that cannot be distributed due to insufficient `remainReward`.

Resolution: The Client's team acknowledged the finding.

F-2024-6885 - The Approval Cannot Be Reverted - Medium

Description: The `Locking` contract allows privileged account to `approve` particular validator. Upon approval, the validator can call the `create` function. However, there is no functionality to remove particular validator from the `approvals` collection. Thus, in the event of malicious behaviour of validator or incorrect address approved, there is no possibility to revert this configuration.

```
function approve(address validator) external override onlyOwner {
    require(!approvals[validator], "approved");
    approvals[validator] = true;
    emit Approval(validator);
}
```

Assets:

- contracts/interfaces/Locking.sol [https://github.com/GOATNetwork/goat-contracts/blob/main/contracts/locking/Locking.sol]

Status: Accepted

Classification

Impact: 3/5

Likelihood: 4/5

Exploitability: Independent

Complexity: Simple

Severity: Medium

Recommendations

Remediation: It is recommended to implement a functionality allowing the protocol's owner to remove particular address from the `approvals` collection.

Resolution: The Client's team acknowledged the finding.

F-2024-6869 - The Locking Contract Lacks Two-Step Ownership Transfer - Low

Description: The solution implements single step ownership transfer. Thus, accidental transfer of ownership to unverified and incorrect address may result in loss of ownership. In such a case, access to every function protected by the ownership check will be permanently lost.

Assets:

- contracts/locking/Locking.sol [https://github.com/GOATNetwork/goat-contracts/blob/main/contracts/locking/Locking.sol]

Status: Accepted

Classification

Impact: 4/5

Likelihood: 1/5

Exploitability: Independent

Complexity: Simple

Severity: Low

Recommendations

Remediation: It is recommended to implement a two-step ownership transfer pattern within the solution, such as OpenZepplin’s Ownable2Step.

Resolution: The Client's team acknowledged the finding.

F-2024-6881 - Lack of Compliance With EIP-712 Standard - Low

Description:

The `create` makes use of signature-based authorisation.. However, the signature currently makes use only of input data and `block.chainid`.

```
    function create(
        bytes32[2] calldata pubkey,
        bytes32 sigR,
        bytes32 sigS,
        uint8 sigV
    ) external payable override {
        require(threshold.length > 0, "not started");

        address validator = pubkey.ConsAddress();
        bytes32 hash = keccak256(
            abi.encodePacked(block.chainid, validator, msg.sender)
        );
        require(
            pubkey.EthAddress() == ECDSA.recover(hash, sigV, sigR, sigS),
            "signer mismatched"
        );
        ...
```

Thus, such implementation is not compliant with the EIP-712: Typed structured data hashing and signing. The EIP-712 assumes that the hash compared with the signature should include a domain separator, created with:

- name
- version
- blockchain ID
- contract's address

Lack of compliance may lead to signature replay in other smart contracts and blockchains. It is also considered a deviation from leading security practices.

The EIP-712 standard introduces the concepts of `typehash` and `domain separator`. The `typehash` provides an additional level of security by ensuring that different structures do not have the same digest hash. In turn, the `domain separator` contains information about which contract is going to validate the signed message. The domain separator includes user-defined name and version, the chain ID, and the address of the contract that will validate the signature. Eventually, the usage of the chain ID prevents cross-chain replay attacks.

Assets:

- contracts/locking/Locking.sol [https://github.com/GOATNetwork/goat-contracts/blob/main/contracts/locking/Locking.sol]

Status: Accepted

Classification

Impact: 2/5

Likelihood: 2/5

Exploitability: Independent

Complexity: Simple

Severity: Low

Recommendations

Remediation: It is recommended to implement signature based authorisation compliant with the EIP-712 standard to prevent any signature replay between smart contracts on the same blockchain.

Resolution: The Client's team acknowledged the finding.

F-2024-6884 - The changeValidatorOwner Function Lacks Two-Step Ownership Transfer - Low

Description: The `Locking` contract has the `changeValidatorOwner` function that allows validator owner to transfer ownership within single step. Thus, accidental transfer of ownership to unverified and incorrect address may result in loss of validator's ownership. In such a case, access to every function protected by the `OnlyValidatorOwner` modifier, such as `claim`, will be permanently lost.

```
function changeValidatorOwner(
    address validator,
    address newOwner
) external OnlyValidatorOwner(validator) {
    require(newOwner != address(0), "invalid address");
    owners[validator] = newOwner;
    emit ChangeValidatorOwner(validator, newOwner);
}
```

Assets:

- contracts/locking/Locking.sol [https://github.com/GOATNetwork/goat-contracts/blob/main/contracts/locking/Locking.sol]

Status: Accepted

Classification

Impact: 4/5

Likelihood: 1/5

Exploitability: Independent

Complexity: Simple

Severity: Low

Recommendations

Remediation: It is recommended to implement a two-step validator's ownership transfer pattern.

Resolution: The Client's team acknowledged the finding.

Observation Details

F-2024-6865 - Incomplete Interface Definition - Info

Description:

The `ILocking` interface is missing several key functions that are present in the `Locking` contract implementation. Interfaces are crucial for defining the external contract's interaction surface, especially when other contracts or external services depend on these function signatures for proper interaction.

In this case, the following functions are missing from the interface:

1. `changeValidatorOwner(address validator, address newOwner)` This function allows a validator's ownership to be transferred to a new address.
2. `openClaim()`: This function enables the contract owner to open the reward claim process, making rewards claimable by the validators.
3. `reclaim()`: This function allows users to reclaim unclaimed "Goat" token rewards after the claim period is open.

Assets:

- contracts/interfaces/Locking.sol [https://github.com/GOATNetwork/goat-contracts/blob/main/contracts/locking/Locking.sol]

Status: Accepted

Recommendations

Remediation: It is recommended to add the missing functions to the `ILocking` interface to ensure all essential interactions are exposed.

Resolution: The Client's team acknowledged the finding.

F-2024-6868 - Floating Pragma - Info

Description:

In Solidity development, the pragma directive specifies the compiler version to be used, ensuring consistent compilation and reducing the risk of issues caused by version changes. However, using a floating pragma (e.g., `^0.8.24`) introduces uncertainty, as it allows contracts to be compiled with any version within a specified range. This can result in discrepancies between the compiler used in testing and the one used in deployment, increasing the likelihood of vulnerabilities or unexpected behavior due to changes in compiler versions.

The project currently uses floating pragma declarations (`^0.8.24`) in its Solidity contracts. This increases the risk of deploying with a compiler version different from the one tested, potentially reintroducing known bugs from older versions or causing unexpected behavior with newer versions. These inconsistencies could result in security vulnerabilities, system instability, or financial loss. Locking the pragma version to a specific, tested version is essential to prevent these risks and ensure consistent contract behavior.

Assets:

- contracts/interfaces/Locking.sol [https://github.com/GOATNetwork/goat-contracts/blob/main/contracts/locking/Locking.sol]
- contracts/library/codec/pubkey.sol [https://github.com/GOATNetwork/goat-contracts/blob/main/contracts/locking/Locking.sol]
- contracts/library/utils/RateLimiter.sol [https://github.com/GOATNetwork/goat-contracts/blob/main/contracts/locking/Locking.sol]
- contracts/library/constants/Executor.sol [https://github.com/GOATNetwork/goat-contracts/blob/main/contracts/locking/Locking.sol]
- contracts/locking/Locking.sol [https://github.com/GOATNetwork/goat-contracts/blob/main/contracts/locking/Locking.sol]

Status: Accepted

Recommendations

Remediation:

It is recommended to **lock the pragma version** to the specific version that was used during development and testing. This ensures that the contract will always be compiled with a known, stable compiler version, preventing unexpected changes in behavior due to

compiler updates. For example, instead of using `^0.8.24`, explicitly define the version with `pragma solidity 0.8.24;`.

Before selecting a version, review known bugs and vulnerabilities associated with each Solidity compiler release. This can be done by referencing the official Solidity compiler release notes: Solidity GitHub releases or Solidity Bugs by Version. Choose a compiler version with a good track record for stability and security.

Resolution: The Client's team acknowledged the finding.

F-2024-6871 - Custom Errors in Solidity for Gas Efficiency - Info

Description:

Starting from Solidity version 0.8.4, the language introduced a feature known as "custom errors". These custom errors provide a way for developers to define more descriptive and semantically meaningful error conditions without relying on string messages. Prior to this version, developers often used the `require` statement with string error messages to handle specific conditions or validations. However, every unique string used as a revert reason consumes gas, making transactions more expensive.

```
    function openClaim() external onlyOwner {
        require(!claimable, "claim is open");
        claimable = true;
        emit OpenClaim();
    }
```

Custom errors, on the other hand, are identified by their name and the types of their parameters only, and they do not have the overhead of string storage. This means that, when using custom errors instead of `require` statements with string messages, the gas consumption can be significantly reduced, leading to more gas-efficient contracts.

Assets:

- contracts/locking/Locking.sol [https://github.com/GOATNetwork/goat-contracts/blob/main/contracts/locking/Locking.sol]

Status: Accepted

Recommendations

Remediation:

It is recommended to use custom errors instead of revert strings to reduce gas costs, especially during contract deployment. Custom errors can be defined using the error keyword and can include dynamic information.

```
error ClaimIsOpenError();
...
    function openClaim() external onlyOwner {
        if (claimable) {
            revert ClaimIsOpenError;
        }
        claimable = true;
```

```
        emit OpenClaim();
    }
```

Resolution: The Client's team acknowledged the finding.

F-2024-6875 - Event Emitted Before State Change - Info

Description:

In the `setThreshold` and `setTokenWeight` functions, events are emitted before the actual state change occurs. This ordering can lead to inconsistencies between the emitted event logs and the actual contract state. Emitting events before the change can cause issues if the transaction fails or reverts after the event is emitted, resulting in an event log that incorrectly suggests a state modification that never took place.

```
function setThreshold(address token, uint256 amount) external override onlyOw
ner {
    require(tokens[token].exist, "token not found");

    uint256 thres = tokens[token].threshold;
    require(thres != amount, "no changes");

    tokens[token].threshold = amount;
    emit UpdateTokenThreshold(token, amount); // Event emitted before finishi
ng all updates

    if (thres == 0 && amount > 0) {
        require(threshold.length < MAX_TOKEN_SIZE, "threshold length too larg
e");
        threshold.push(token); // Additional state update happens here
        return;
    }

    if (thres > 0 && amount > 0) {
        return;
    }

    // Additional logic for removing from the threshold list omitted for brev
ity
}
```

```
function setTokenWeight(address token, uint64 weight) external override onlyO
wner {
    require(tokens[token].exist, "token not found");
    require(weight < MAX_WEIGHT, "invalid weight");

    emit UpdateTokenWeight(token, weight); // Event emitted before the state
change

    if (weight != 0) {
        tokens[token].weight = weight; // State change occurs here
```

```
        return;
    }

    // Additional logic for deleting and updating thresholds omitted for brev
ity
}
```

While this does not directly affect the contract's security, it can mislead users or systems relying on event logs for state tracking, potentially complicating debugging, monitoring, and off-chain data accuracy.

Assets:

- contracts/locking/Locking.sol [https://github.com/GOATNetwork/goat-contracts/blob/main/contracts/locking/Locking.sol]

Status: Accepted

Recommendations

Remediation: It is recommended to reorder the `setThreshold` and `setTokenWeight` functions so that state changes occur before events are emitted, ensuring that event logs accurately reflect the actual state of the contract. This adjustment will improve the reliability of event logs and maintain alignment with best practices in Solidity.

F-2024-6880 - The OnlyValidatorOwner Modifier Has Redundant Assertion - Info

Description: The `OnlyValidatorOwner` modifier has two assertions implemented to check whether owner is not zero address and whether it is equal to the `msg.sender`. However, the first assertion appears to be redundant, as second covers it entirely. The `msg.sender` cannot have zero address anyway.

```
    modifier OnlyValidatorOwner(address validator) {
        address owner = owners[validator];
        require(owner != address(0), "validator not found");
        require(owner == msg.sender, "not validator owner");
        _;
    }
```

Assets:

- contracts/locking/Locking.sol [https://github.com/GOATNetwork/goat-contracts/blob/main/contracts/locking/Locking.sol]

Status: Accepted

Recommendations

Remediation: It is recommended to remove redundant assertion to save some Gas during the runtime.

Resolution: The Client's team acknowledged the finding.

Disclaimers

Hacken Disclaimer

The smart contracts given for audit have been analyzed based on best industry practices at the time of the writing of this report, with cybersecurity vulnerabilities and issues in smart contract source code, the details of which are disclosed in this report (Source Code); the Source Code compilation, deployment, and functionality (performing the intended functions).

The report contains no statements or warranties on the identification of all vulnerabilities and security of the code. The report covers the code submitted and reviewed, so it may not be relevant after any modifications. Do not consider this report as a final and sufficient assessment regarding the utility and safety of the code, bug-free status, or any other contract statements.

While we have done our best in conducting the analysis and producing this report, it is important to note that you should not rely on this report only — we recommend proceeding with several independent audits and a public bug bounty program to ensure the security of smart contracts.

English is the original language of the report. The Consultant is not responsible for the correctness of the translated versions.

Technical Disclaimer

Smart contracts are deployed and executed on a blockchain platform. The platform, its programming language, and other software related to the smart contract can have vulnerabilities that can lead to hacks. Thus, the Consultant cannot guarantee the explicit security of the audited smart contracts.

Appendix 1. Definitions

Severities

When auditing smart contracts, Hacken is using a risk-based approach that considers **Likelihood**, **Impact**, **Exploitability** and **Complexity** metrics to evaluate findings and score severities.

Reference on how risk scoring is done is available through the repository in our Github organization:

hknio/severity-formula

Severity	Description
Critical	Critical vulnerabilities are usually straightforward to exploit and can lead to the loss of user funds or contract state manipulation.
High	High vulnerabilities are usually harder to exploit, requiring specific conditions, or have a more limited scope, but can still lead to the loss of user funds or contract state manipulation.
Medium	Medium vulnerabilities are usually limited to state manipulations and, in most cases, cannot lead to asset loss. Contradictions and requirements violations. Major deviations from best practices are also in this category.
Low	Major deviations from best practices or major Gas inefficiency. These issues will not have a significant impact on code execution.

Potential Risks

The "Potential Risks" section identifies issues that are not direct security vulnerabilities but could still affect the project’s performance, reliability, or user trust. These risks arise from design choices, architectural decisions, or operational practices that, while not immediately exploitable, may lead to problems under certain conditions. Additionally, potential risks can impact the quality of the audit itself, as they may involve external factors or components beyond the scope of the audit, leading to incomplete assessments or oversight of key areas. This section aims to provide a broader perspective on factors that could affect the project's long-term security, functionality, and the comprehensiveness of the audit findings.

Appendix 2. Scope

The scope of the project includes the following smart contracts from the provided repository:

Scope Details	
Repository	https://github.com/GOATNetwork/goat-contracts
Commit	6d68a950806eca27ca4265280976bd2d94514be1
Retest Commit	n/a
Whitepaper	https://www.goat.network/whitepaper, https://www.goat.network/econpaper
Requirements	n/a
Technical Requirements	n/a

Asset	Type
contracts/interfaces/Locking.sol [https://github.com/GOATNetwork/goat-contracts/blob/main/contracts/locking/Locking.sol]	Smart Contract
contracts/library/codec/pubkey.sol [https://github.com/GOATNetwork/goat-contracts/blob/main/contracts/locking/Locking.sol]	Smart Contract
contracts/library/constants/Executor.sol [https://github.com/GOATNetwork/goat-contracts/blob/main/contracts/locking/Locking.sol]	Smart Contract
contracts/library/utils/BaseAccess.sol [https://github.com/GOATNetwork/goat-contracts/blob/main/contracts/locking/Locking.sol]	Smart Contract
contracts/library/utils/RateLimiter.sol [https://github.com/GOATNetwork/goat-contracts/blob/main/contracts/locking/Locking.sol]	Smart Contract
contracts/locking/Locking.sol [https://github.com/GOATNetwork/goat-contracts/blob/main/contracts/locking/Locking.sol]	Smart Contract

Appendix 3. Additional Valuables

Verification of System Invariants

During the audit of GOAT Network Hacken followed its methodology by performing fuzz-testing on the project's main functions. Foundry, a tool used for fuzz-testing, was employed to check how the protocol behaves under various inputs. Due to the complex and dynamic interactions within the protocol, unexpected edge cases might arise. Therefore, it was important to use fuzz-testing to ensure that several system invariants hold true in all situations.

Fuzz-testing allows the input of many random data points into the system, helping to identify issues that regular testing might miss. A specific Echidna fuzzing suite was prepared for this task, and throughout the assessment, 5 invariants were tested over 50,000 runs. This thorough testing ensured that the system works correctly even with unexpected or unusual inputs.

Invariant	Test Result	Run Count
New token can added with any valid input without revert.	Passed	50k
Token threshold can be updated with any valid input without revert.	Passed	50k
Token weight can be updated with any valid input without revert.	Passed	50k
New validator can be created, then additional amount can be locked and unlocked.	Passed	50k
New validator can be created, then additional amount can be locked within new token and unlocked.	Passed	50k

Additional Recommendations

The smart contracts in the scope of this audit could benefit from the introduction of automatic emergency actions for critical activities, such as unauthorized operations like ownership changes or proxy upgrades, as well as unexpected fund manipulations, including large withdrawals or minting events. Adding such mechanisms would enable the protocol to react automatically to unusual activity, ensuring that the contract remains secure and functions as intended.

To improve functionality, these emergency actions could be designed to trigger under specific conditions, such as:

- Detecting changes to ownership or critical permissions.
- Monitoring large or unexpected transactions and minting events.
- Pausing operations when irregularities are identified.

These enhancements would provide an added layer of security, making the contract more robust and better equipped to handle unexpected situations while maintaining smooth operations.