Audit Report

LiquidLayer

December 2023

Repository	https://github.com/LiquidLayerOff/lila-blockchain
Commit	622c287b8294e5033c47ef4234b1f9ec66667cf4
Audited by	© cyberscope

Table of Contents

Table of Contents **1**
Review **2**
- Audit Updates 2
- Source Files 2

Overview **3**
- Key Functionalities and Operations 3
  - Checkpoint Management and Voting System 3
  - Admin Authorization and Security Measures 3
  - Checkpoint Verification and Registration 3
  - Utility Functions for Network and Admin Management 4

Findings Breakdown **5**
Diagnostics **6**
- TVO - Threshold Validation Oversight 7
  - Description 7
  - Recommendation 8
- CPR - Centralized Parameter Risk 9
  - Description 9
  - Recommendation 11
- URBU - Unchecked Recent Block Usage 12
  - Description 12
  - Recommendation 12
- MPV - Missing Parameter Validation 14
  - Description 14
  - Recommendation 14
- MEM - Missing Error Messages 16
  - Description 16
  - Recommendation 16
- L04 - Conformance to Solidity Naming Conventions 17
  - Description 17
  - Recommendation 17
- L19 - Stable Compiler Version 19
  - Description 19
  - Recommendation 19

Functions Analysis **20**
Flow Graph **21**
Summary **22**
Disclaimer **23**
About Cyberscope **24**

Review

Repository	https://github.com/LiquidLayerOff/lila-blockchain
Commit	622c287b8294e5033c47ef4234b1f9ec66667cf4

Audit Updates

Initial Audit	12 Dec 2023

Source Files

Filename	SHA256
oracle.sol	4e7235c8dbd24bcaf065e19002737d184f2f9b75a2e54614050ede8a3bbdd2a9

Overview

The `CheckpointOracle` contract, implementation in the blockchain, is primarily designed to manage blockchain checkpoints. These checkpoints are crucial markers within the blockchain, signifying specific blocks that are recognized as valid and confirmed. The main goal of this contract is to enhance network security, facilitate faster synchronization of new nodes, and ensure stability and consistency in the blockchain's state.

Key Functionalities and Operations

Checkpoint Management and Voting System

A pivotal feature of the contract is its ability to create and manage checkpoints. This process is not unilateral but involves a voting system among designated admins. The contract uses a list of these admins to maintain decentralized control over which checkpoints are established. When a new checkpoint is proposed, it must garner votes from these admins. Each vote is logged as an event, ensuring transparency in the decision-making process. The voting mechanism is crucial for maintaining the trustless nature of the blockchain, where no single entity has absolute control over the confirmation of checkpoints.

Admin Authorization and Security Measures

The contract includes a robust security framework, primarily revolving around admin authorization. Only authorized admins can propose and vote on checkpoints, and their addresses are stored in a list for reference. This admin list is crucial for maintaining the integrity and security of the checkpoint system. The contract also implements various checks to ensure the validity of the proposed checkpoints, such as replay protection and block hash verification.

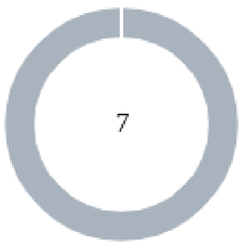
Checkpoint Verification and Registration

The contract is equipped to handle the registration of new checkpoints, provided they pass the necessary security checks and admin voting. These checkpoints are then recorded in the blockchain, along with associated information like the block height and checkpoint hash. This function is vital for marking certain blocks as valid, which helps in preventing possible chain reorganizations and attacks.

Utility Functions for Network and Admin Management

Additional utility functions in the contract include the ability to retrieve the latest stable checkpoint and to obtain a list of all admin addresses. These features are designed to assist in network management and ensure that users and other contracts interacting with CheckpointOracle can easily access important information regarding the current state of the blockchain and the entities involved in checkpoint management.

Findings Breakdown



●	Critical	0
●	Medium	0
●	Minor / Informative	7

Severity		Unresolved	Acknowledged	Resolved	Other
●	Critical	0	0	0	0
●	Medium	0	0	0	0
●	Minor / Informative	7	0	0	0

Diagnostics

● Critical ● Medium ● Minor / Informative

Severity	Code	Description	Status
●	TVO	Threshold Validation Oversight	Unresolved
●	CPR	Centralized Parameter Risk	Unresolved
●	URBU	Unchecked Recent Block Usage	Unresolved
●	MPV	Missing Parameter Validation	Unresolved
●	MEM	Missing Error Messages	Unresolved
●	L04	Conformance to Solidity Naming Conventions	Unresolved
●	L19	Stable Compiler Version	Unresolved

TVO - Threshold Validation Oversight

Criticality	Minor / Informative
Location	oracle.sol#L26,114
Status	Unresolved

Description

The contract is currently processing the `threshold` variable without adequate validation to ensure it aligns with the structure of the `adminList`. Specifically, the `threshold` variable, set in the constructor, represents the minimum number of signatures required to validate actions such as setting a new checkpoint. However, there is no check to ensure that the `threshold` is less than or equal to the length of the `adminList`. This oversight could lead to a situation where the `threshold` is set higher than the number of available admins, making it impossible to reach the required number of signatures for critical operations, thereby rendering the contract's multisignature functionality ineffective.

```
 constructor(address[] memory _adminlist, uint _sectionSize, uint
_processConfirms, uint _threshold) public {
        for (uint i = 0; i < _adminlist.length; i++) {
            admins[_adminlist[i]] = true;
            adminList.push(_adminlist[i]);
        }
        ...
        threshold = _threshold;
    }

    function SetCheckpoint(
        uint _recentNumber,
        bytes32 _recentHash,
        bytes32 _hash,
        uint64 _sectionIndex,
        uint8[] memory v,
        bytes32[] memory r,
        bytes32[] memory s)
        public
        returns (bool)
    {
        ...
        for (uint idx = 0; idx < v.length; idx++){
            ...
            // Sufficient signatures present, update latest checkpoint.
            if (idx+1 >= threshold){
                hash = _hash;
                height = block.number;
                sectionIndex = _sectionIndex;
                return true;
            }
        }
        ...
    }
```

Recommendation

It is recommended to introduce a validation check in the constructor to ensure that the `threshold` does not exceed the length of the `adminList`. This can be implemented as a `require` statement that compares the threshold with `_adminlist`.length. Such validation will prevent setting a `threshold` that is unattainable, ensuring the contract remains functional and secure. This change will safeguard against potential deadlocks in the contract's operation, maintaining the integrity and intended functionality of the multi-signature process.

CPR - Centralized Parameter Risk

Criticality	Minor / Informative
Location	oracle.sol#L19,50
Status	Unresolved

Description

The contract is designed to include multiple filters within the `SetCheckpoint` function that ensure the validity of the checkpoints being set. These filters are vital for the contract's functionality, as they prevent the registration of future, old, stale, or invalid checkpoints. However, the effectiveness of these filters is heavily dependent on the correct initialization and management of certain parameters such as `sectionSize` , `processConfirms` , and `sectionIndex` . The current implementation relies on these parameters being correctly set at the time of the contract's deployment. Any misconfiguration or improper setting of these values can lead to vulnerabilities, such as incorrect filtering, which might compromise the security and integrity of the checkpointing process. This reliance on parameters set during initialization introduces a degree of centralization and associated risks, as it entrusts the deployer with substantial control over the contract's operational parameters.

The contract's `SetCheckpoint` function, designed for multi-signature verification, is susceptible to centralized control. The owner possesses the authority to initialize addresses used for the multi-signature functionality. However, the concern arises as the owner can set these addresses arbitrarily, including their own. This undermines the intended purpose of multi-signature security, as the owner retains the ability to unilaterally influence or compromise the multi-signature process.

```
   constructor(address[] memory _adminlist, uint _sectionSize, uint
_processConfirms, uint _threshold) public {
        for (uint i = 0; i < _adminlist.length; i++) {
            admins[_adminlist[i]] = true;
            adminList.push(_adminlist[i]);
        }
        sectionSize = _sectionSize;
        processConfirms = _processConfirms;
        threshold = _threshold;
    }

    function SetCheckpoint(
        uint _recentNumber,
        bytes32 _recentHash,
        bytes32 _hash,
        uint64 _sectionIndex,
        uint8[] memory v,
        bytes32[] memory r,
        bytes32[] memory s)
        public
        returns (bool)
    {
        // Ensure the sender is authorized.
        require(admins[msg.sender]);

        // These checks replay protection, so it cannot be replayed on
forks,
        // accidentally or intentionally
        require(blockhash(_recentNumber) == _recentHash);

        // Ensure the batch of signatures are valid.
        require(v.length == r.length);
        require(v.length == s.length);

        // Filter out "future" checkpoint.
        if (block.number <
(_sectionIndex+1)*sectionSize+processConfirms) {
            return false;
        }
        // Filter out "old" announcement
        if (_sectionIndex < sectionIndex) {
            return false;
        }
        // Filter out "stale" announcement
        if (_sectionIndex == sectionIndex && (_sectionIndex != 0 ||
height != 0)) {
            return false;
        }
        // Filter out "invalid" announcement
        if (_hash == ""){
            return false;
        }
    ...
```

Recommendation

It is recommended to introduce a mechanism for dynamically updating these critical parameters (`sectionSize` , `processConfirms` , `sectionIndex`) post-deployment, preferably through a consensus-driven governance process involving multiple parties or stakeholders. Implementing access controls or a multi-signature scheme for these updates can further decentralize control and reduce the risk of centralized manipulation or errors. Additionally, integrating a thorough validation and sanity checks both at the time of setting and using these parameters can prevent potential misconfigurations. These measures will enhance the contract's adaptability and resilience, ensuring that it remains secure and functional in the face of changing conditions and requirements. Furthermor, the team is advised to implement a consensus mechanism among current admins for any changes, ensuring no single entity can manipulate the multisig functionality. This enhances security and maintains the integrity of the multisignature process.

URBU - Unchecked Recent Block Usage

Criticality	Minor / Informative
Location	oracle.sol#L50
Status	Unresolved

Description

The contract uses the `_recentNumber` variable in the `SetCheckpoint` function to provide replay protection. This approach checks the block hash of `_recentNumber` against a provided `_recentHash` to ensure that the checkpoint data is current and to prevent its reuse on a forked or older version of the chain. However, due to Ethereum's limitation of only allowing access to the hashes of the most recent 256 blocks, it is crucial that `_recentNumber` refers to a block within this range. If `_recentNumber` references an older block, the contract would be unable to retrieve its hash, leading to potential failures in the execution of this function.

```
function SetCheckpoint(
        uint _recentNumber,
        bytes32 _recentHash,
        bytes32 _hash,
        uint64 _sectionIndex,
        uint8[] memory v,
        bytes32[] memory r,
        bytes32[] memory s)
        public
        returns (bool)
    {
        // Ensure the sender is authorized.
        require(admins[msg.sender]);

        // These checks replay protection, so it cannot be replayed
on forks,
        // accidentally or intentionally
        require(blockhash(_recentNumber) == _recentHash);
        ...
```

Recommendation

It is recommended to add a validation check in the `SetCheckpoint` function to ensure that `_recentNumber` is within the last 256 blocks. This can be achieved by comparing `_recentNumber` to the current block number and ensuring that the difference falls within the permissible range. Additionally, clear documentation and error messages should be provided to indicate the reason for failure if the provided `_recentNumber` is out of the acceptable range. This validation will reinforce the contract's replay protection mechanism and maintain the integrity of the checkpointing process.

MPV - Missing Parameter Validation

Criticality	Minor / Informative
Location	oracle.sol#L19
Status	Unresolved

Description

The contract is configured to initialize `processConfirms` and `sectionSize` directly from the constructor parameters without enforcing checks for their values. This approach presents potential security vulnerabilities. Specifically, if `processConfirms` is set to a value lower than the Ethereum standard (typically `256` confirmations), it could increase the risk of chain reorganizations, as it may not provide a sufficient buffer against short-term chain fluctuations and possible reorgs. Likewise, an incorrect setting of `sectionSize`, which determines the frequency of checkpoint creation, could lead to either overly frequent or infrequent checkpoints. This misconfiguration can impact the operational efficiency and the security protocol of the blockchain, especially if it deviates significantly from established practices like the Ethereum standard of `32,768` blocks for checkpoint intervals.

```
    constructor(address[] memory _adminlist, uint _sectionSize, uint
_processConfirms, uint _threshold) public {
        for (uint i = 0; i < _adminlist.length; i++) {
            admins[_adminlist[i]] = true;
            adminList.push(_adminlist[i]);
        }
        sectionSize = _sectionSize;
        processConfirms = _processConfirms;
        threshold = _threshold;
    }
```

Recommendation

It is recommended to implement validation checks within the constructor to ensure that both `processConfirms` and `sectionSize` meet their respective minimum thresholds and adhere to established standards. For `processConfirms`, this threshold should ideally align with or exceed the standard `256` confirmations. For `sectionSize`, the value should be carefully chosen to balance security and operational efficiency, potentially aligning with the Ethereum standard of `32,768` blocks. Incorporating these validations will significantly enhance the contract's security and reliability. Clear documentation of these requirements in the code and any accompanying documentation is also advised to inform future maintainers or auditors about the importance of these settings for the contract's proper functioning.

MEM - Missing Error Messages

Criticality	Minor / Informative
Location	oracle.sol#L62,66,69,70,108,109
Status	Unresolved

Description

The contract is missing error messages. There are no error messages to accurately reflect the problem, making it difficult to identify and fix the issue. As a result, the users will not be able to find the root cause of the error.

```
require(admins[msg.sender])
require(blockhash(_recentNumber) == _recentHash)
require(v.length == r.length)
require(v.length == s.length)
require(admins[signer])
require(uint256(signer) > uint256(lastVoter))
```

Recommendation

The team is suggested to provide a descriptive message to the errors. This message can be used to provide additional context about the error that occurred or to explain why the contract execution was halted. This can be useful for debugging and for providing more information to users that interact with the contract.

L04 - Conformance to Solidity Naming Conventions

Criticality	Minor / Informative
Location	oracle.sol#L35,50,51,52,53,54,129
Status	Unresolved

Description

The Solidity style guide is a set of guidelines for writing clean and consistent Solidity code. Adhering to a style guide can help improve the readability and maintainability of the Solidity code, making it easier for others to understand and work with.

The followings are a few key points from the Solidity style guide:

1. Use camelCase for function and variable names, with the first letter in lowercase (e.g., myVariable, updateCounter).
2. Use PascalCase for contract, struct, and enum names, with the first letter in uppercase (e.g., MyContract, UserStruct, ErrorEnum).
3. Use uppercase for constant variables and enums (e.g., MAX_VALUE, ERROR_CODE).
4. Use indentation to improve readability and structure.
5. Use spaces between operators and after commas.
6. Use comments to explain the purpose and behavior of the code.
7. Keep lines short (around 120 characters) to improve readability.

```
function GetLatestCheckpoint()
    view
    public
    returns(uint64, bytes32, uint) {
        return (sectionIndex, hash, height);
    }
...
```

Recommendation

By following the Solidity naming convention guidelines, the codebase increased the readability, maintainability, and makes it easier to work with.

Find more information on the Solidity documentation
https://docs.soliditylang.org/en/v0.8.17/style-guide.html#naming-convention.

L19 - Stable Compiler Version

Criticality	Minor / Informative
Location	oracle.sol#L1
Status	Unresolved

Description

The `^` symbol indicates that any version of Solidity that is compatible with the specified version (i.e., any version that is a higher minor or patch version) can be used to compile the contract. The version lock is a mechanism that allows the author to specify a minimum version of the Solidity compiler that must be used to compile the contract code. This is useful because it ensures that the contract will be compiled using a version of the compiler that is known to be compatible with the code.

```
pragma solidity ^0.6.0;
```

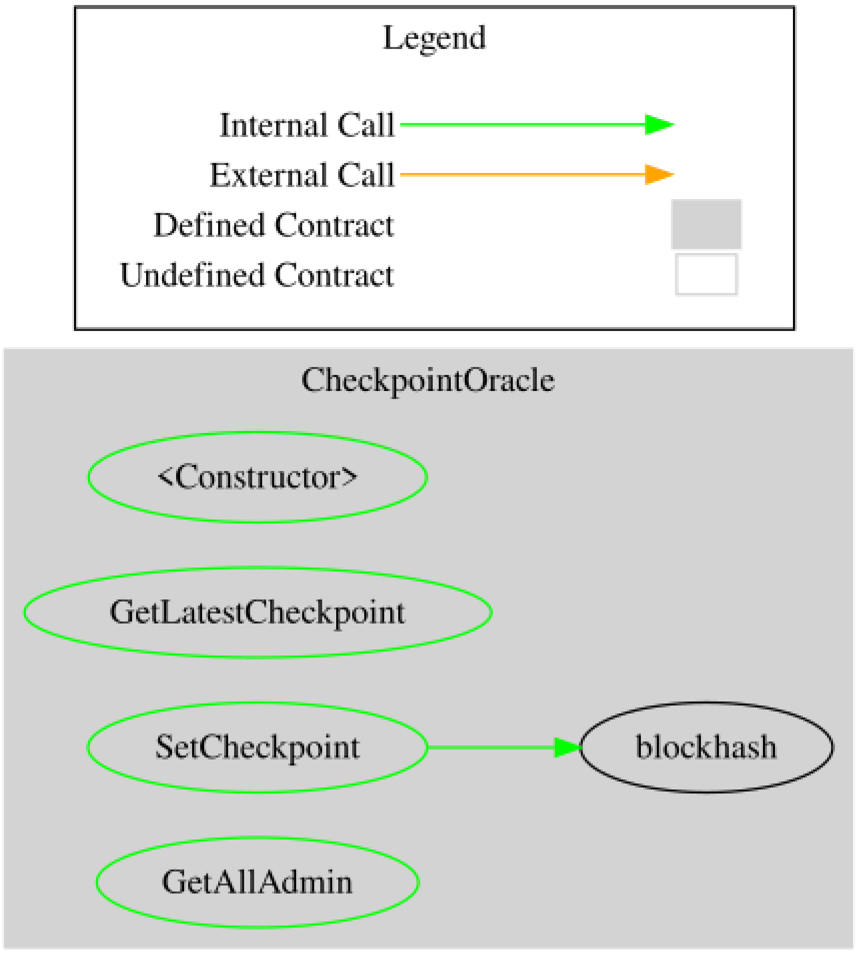
Recommendation

The team is advised to lock the pragma to ensure the stability of the codebase. The locked pragma version ensures that the contract will not be deployed with an unexpected version. An unexpected version may produce vulnerabilities and undiscovered bugs. The compiler should be configured to the lowest version that provides all the required functionality for the codebase. As a result, the project will be compiled in a well-tested LTS (Long Term Support) environment.

Functions Analysis

Contract	Type	Bases		
	Function Name	**Visibility**	**Mutability**	**Modifiers**
CheckpointOracle	Implementation			
		Public	✓	-
	GetLatestCheckpoint	Public		-
	SetCheckpoint	Public	✓	-
	GetAllAdmin	Public		-

Flow Graph


Legend
Internal Call
External Call
Defined Contract
Undefined Contract
CheckpointOracle
<Constructor>
GetLatestCheckpoint
SetCheckpoint
blockhash
GetAllAdmin

Summary

The CheckpointOracle contract serves as a decentralized, secure, and transparent system for managing blockchain checkpoints. Its functionalities are geared towards ensuring the security, stability, and consistency of the blockchain network, making it an essential component in maintaining the integrity of the blockchain system. This audit investigates security issues, business logic concerns, and potential improvements.

Disclaimer

About Cyberscope

Cyberscope is a blockchain cybersecurity company that was founded with the vision to make web3.0 a safer place for investors and developers. Since its launch, it has worked with thousands of projects and is estimated to have secured tens of millions of investors' funds.

Cyberscope is one of the leading smart contract audit firms in the crypto space and has built a high-profile network of clients and partners.

The Cyberscope team

https://www.cyberscope.io