CERTIK

Lightchain

Preliminary Comments

CertiK Assessed on Jul 28th, 2025

CertiK Assessed on Jul 28th, 2025

Lightchain

These preliminary comments were prepared by CertiK, the leader in Web3.0 security.

Executive Summary

TYPES	ECOSYSTEM	METHODS
	EVM Compatible	Formal Verification, Manual Review, Static Analysis
LANGUAGE	**TIMELINE**	**KEY COMPONENTS**
Solidity	Delivered on 07/28/2025	N/A

CODEBASE

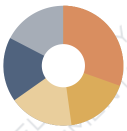
Zip file hash:

openssl dgst -sha256 Smart-Contracts-main.zip

SHA2-256(Smart-Contracts-main.zip)=

View All in Codebase Page

Vulnerability Summary



23	0	0	0	0	23
Total Findings	Resolved	Partially Resolved	Acknowledged	Declined	Pending

🟧 3	Centralization	3 Pending	Centralization findings highlight privileged roles & functions and their capabilities, or instances where the project takes custody of users' assets.
🟥 0	Critical		Critical risks are those that impact the safe functioning of a platform and must be addressed before launch. Users should not invest in any project with outstanding critical risks.
🟧 4	Major	4 Pending	Major risks may include logical errors that, under specific circumstances, could result in fund losses or loss of project control.
🟨 4	Medium	4 Pending	Medium risks may not pose a direct risk to users' funds, but they can affect the overall functioning of a platform.
🟨 4	Minor	4 Pending	Minor risks can be any of the above, but on a smaller scale. They generally do not compromise the overall integrity of the project, but they may be less efficient than other solutions.
🟦 4	Informational	4 Pending	Informational errors are often recommendations to improve the style of the code or certain operations to fall within industry best practices. They usually do not affect the overall functioning of the code.

■ 4	Discussion	4 Pending	The impact of the issue is yet to be determined, hence requires further clarifications from the project team.

TABLE OF CONTENTS | LIGHTCHAIN

CODEBASE | LIGHTCHAIN

Repository

Zip file hash:

openssl dgst -sha256 Smart-Contracts-main.zip

SHA2-256(Smart-Contracts-main.zip)= 8fc9d92f5480f90318f4b6ac6b528d9e05dd95644d0f75201a994b719f22ec92

AUDIT SCOPE | LIGHTCHAIN

9 files audited ● 9 files with Pending findings

ID	Repo	File	SHA256 Checksum
● MAC	CertiKProject/certik-audit-projects	Smart-Contracts-main/ModelAccessCredits.sol	74dcdf4396da7d11a9a417f91acd76d6f157db9b843095cc5f8cd37e8be26184
● MDA	CertiKProject/certik-audit-projects	Smart-Contracts-main/ModelDAO.sol	36f098cb6425a4127dc3d3fd86d2922cea557d4a55978f6b0ec7158d2ed45360
● MRS	CertiKProject/certik-audit-projects	Smart-Contracts-main/ModelRegistry.sol	90308b8171ddd4d217852d6211a7ad629bc5cad2be859997f8c33fec3550253c
● MRC	CertiKProject/certik-audit-projects	Smart-Contracts-main/ModelReward.sol	6e23bf213959fa11197fb4779cca01bb931e074eb5b37b4057ff01aa0298ca5c
● MSS	CertiKProject/certik-audit-projects	Smart-Contracts-main/ModelSlasher.sol	52f5dc47806fe1525ea2b071bc9f3fdfd505cdf446a6aca09eab36b46e158998
● MTS	CertiKProject/certik-audit-projects	Smart-Contracts-main/ModelToken.sol	2027701e5194f44313efbe929bcc9dd81dc546b5db0d61802e237481779612da
● MUP	CertiKProject/certik-audit-projects	Smart-Contracts-main/ModelUpgradeProxy.sol	641fa374c483c3edfbf81395026f5474cf1b7e1877a2d9014aca3d3f3cca074d
● MVR	CertiKProject/certik-audit-projects	Smart-Contracts-main/ModelValidatorRegistry.sol	cbd293912366a0331738dbfa4b739060f6488fca164f1bae442c8f209b7adfa3
● MVS	CertiKProject/certik-audit-projects	Smart-Contracts-main/ModelValidatorStakingPool.sol	f71d76e3896d369b4cb11b031b381c681a02edf196dfecfe368c562f0d38ce25

APPROACH & METHODS | LIGHTCHAIN

This report has been prepared for Lightchain to discover issues and vulnerabilities in the source code of the Lightchain project as well as any contract dependencies that were not part of an officially recognized library. A comprehensive examination has been performed, utilizing Formal Verification, Manual Review, and Static Analysis techniques.

The auditing process pays special attention to the following considerations:

- Testing the smart contracts against both common and uncommon attack vectors.
- Assessing the codebase to ensure compliance with current best practices and industry standards.
- Ensuring contract logic meets the specifications and intentions of the client.
- Cross referencing contract structure and implementation against similar smart contracts produced by industry leaders.
- Thorough line-by-line manual review of the entire codebase by industry experts.

The security assessment resulted in findings that ranged from critical to informational. We recommend addressing these findings to ensure a high level of security standards and industry practices. We suggest recommendations that could better serve the project from the security perspective:

- Testing the smart contracts against both common and uncommon attack vectors;
- Enhance general coding practices for better structures of source codes;
- Add enough unit tests to cover the possible use cases;
- Provide more comments per each function for readability, especially contracts that are verified in public;
- Provide more transparency on privileged activities once the protocol is live.

REVIEW NOTES | LIGHTCHAIN

Overview

The **Lightchain** protocol aims to build a decentralized platform for AI model validation and management, covering key functionalities such as model submission, validation, access control, and governance.

By introducing staking and slashing mechanisms, the protocol enforces responsible behavior among model validators to ensure model quality and overall protocol security. Additionally, the system incorporates token-based governance and economic incentives to support sustainable participation from both model contributors and validators.

Core Contracts

Governance Contracts

`ModelDAO.sol` : The core governance contract of the protocol, enabling proposal submission, voting, and parameter adjustments. It works in conjunction with a `Timelock` contract to enforce delayed and transparent execution of approved proposals, enhancing the security and auditability of protocol changes.

`ModelToken.sol` : The protocol's governance token (MODEL), implemented as an ERC20 token, used for voting and participating in governance decisions.

`ModelUpgradeProxy.sol` : It's designed to handle validator reporting and penalty execution. It allows users to submit reports against validators, which, upon validation, trigger corresponding punitive actions. Its functionality partially overlaps with `ModelSlasher.sol` .

Model Management Contracts

`ModelRegistry.sol` : A registry contract for model submissions and validation records. It supports on-chain storage of model hashes and related verification data to enable trustworthy model submission processes.

`ModelValidatorRegistry.sol` : Maintains a list of approved model validators within the protocol.

`ModelValidatorStakingPool.sol` : A staking pool where users can stake tokens to become validators. It also includes basic slashing functionality to discourage dishonest behavior among validators.

Access Control and Incentive Contracts

`ModelAccessCredits.sol` : Implements a credit-based access control mechanism. Users can purchase access credits using the native token, which are then consumed by designated operators during model inference.

`ModelReward.sol` : Handles reward distribution related to model contributions, including incentives for model submitters and validators.

`ModelSlasher.sol` : A dedicated module for penalizing dishonest validators. It reduces the staked amount of validators found to be misbehaving, thereby helping to maintain protocol integrity.

External Dependencies

The `Lightchain` protocol relies on a few external contracts or addresses to fulfill the needs of its business logic.

The following are third-party dependency contracts used within the contract:

- `@openzeppelin/contracts`
- `@openzeppelin/contracts-upgradeable`

The following are external addresses used within the contracts:

ModelAccessCredits:

- `_timelock`

ModelDAO:

- `token_`
- `rewardVault_`
- `rewardToken_`
- `timelock_`

ModelRegistry:

- `_governance`

ModelReward:

- `dao`
- `_modelToken`

ModelSlasher:

- `_dao`
- `_pool`
- `_treasury`

ModelUpgradeProxy:

- `_dao`
- `_pool`
- `_treasury`

ModelValidatorRegistry:

- `_dao`
- `_stakingPool`

ModelValidatorStakingPool:

- `_dao`
- `_token`

It is assumed that these contracts or addresses are trusted and implemented properly within the whole project.

Privileged Functions

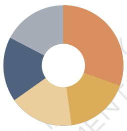
In the `Lightchain` protocol, the admin roles are adopted to ensure the dynamic runtime updates of the project, which are specified in the findings **Centralization Related Risks** and **Centralized Control of Contract Upgrade**.

The advantage of those privileged roles in the codebase is that the client reserves the ability to adjust the protocol according to the runtime required to best serve the community.

It is also worth noting the potential drawbacks of these functions, which should be clearly stated through the client's action/plan. Additionally, if the private keys of the privileged accounts are compromised, it could lead to devastating consequences for the project.

To improve the trustworthiness of the project, dynamic runtime updates in the project should be notified to the community. Any plan to invoke the aforementioned functions should also consider moving to the execution queue of the `Timelock` contract.

FINDINGS | LIGHTCHAIN



23	0	3	4	4	4	4	4
Total Findings	Critical	Centralization	Major	Medium	Minor	Informational	Discussion

This report has been prepared to discover issues and vulnerabilities for Lightchain. Through this audit, we have uncovered 23 issues ranging from different severity levels. Utilizing the techniques of Formal Verification, Manual Review & Static Analysis to complement rigorous manual code reviews, we discovered the following findings:

ID	Title	Category	Severity	Status
LIG-02	**Initial Token Distribution**	**Centralization**	**Centralization**	● **Pending**
LIG-03	**Centralized Control Of Contract Upgrade**	**Centralization**	**Centralization**	● **Pending**
LIG-04	**Centralization Related Risks**	**Centralization**	**Centralization**	● **Pending**
LIG-05	Incorrect Assembly Implementation	Logical Issue	Major	● Pending
LIG-06	Access Control Misalignment In Timelock Governance	Design Issue	Major	● Pending
LIG-17	Slashing Bypass Via Early Withdrawal Request Of Stakes	Logical Issue	Major	● Pending
LIG-18	Improper Authorization Logic May Block `slash` Relevant Functions	Access Control	Major	● Pending
LIG-07	Potential Front-Run On Permit Call To Cause DOS	Denial of Service	Medium	● Pending
LIG-19	Potential Delete Reports By Mistake	Logical Issue	Medium	● Pending
LIG-20	Insufficient Quorum Threshold Allows Proposal Manipulation	Governance	Medium	● Pending
LIG-23	Uncoordinated Governance Parameter Management	Logical Issue	Medium	● Pending

ID	Title	Category	Severity	Status
LIG-08	Usage Of `transfer()` For Sending Native Tokens	Coding Style	Minor	● Pending
LIG-09	No Way To Retrieve ETH From The Contract	Volatile Code	Minor	● Pending
LIG-10	Potentially Unusable Function	Design Issue	Minor	● Pending
LIG-11	Potential Mismatch Between Delegated Votes And Token Balance	Logical Issue	Minor	● Pending
LIG-12	Dead Code	Coding Issue	Informational	● Pending
LIG-13	Risk Of Insufficient Native Token Balance During Refunds	Logical Issue	Informational	● Pending
LIG-14	Use Of `code.length` Is No Longer A Reliable Contract Check	Design Issue	Informational	● Pending
LIG-21	Unnecessary Inheritance Of `Ownable`	Design Issue, Logical Issue	Informational	● Pending
LIG-01	Discussion On Validator Removal Conditions	Logical Issue	Discussion	● Pending
LIG-15	Unclear Contract Design	Design Issue	Discussion	● Pending
LIG-16	Concerns Regarding The Validator Mechanism	Design Issue	Discussion	● Pending
LIG-22	Concerns Regarding The Staking Mechanism	Logical Issue, Design Issue	Discussion	● Pending

LIG-02 | INITIAL TOKEN DISTRIBUTION

Category	Severity	Location	Status
Centralization	● Centralization	Smart-Contracts-main/ModelToken.sol: 27	● Pending

Description

All of the **MODEL** tokens are sent to the contract deployer or one or several externally-owned account (EOA) addresses. This is a centralization risk because the deployer or the owner(s) of the EOAs can distribute tokens without obtaining the consensus of the community. Any compromise to these addresses may allow a hacker to steal and sell tokens on the market, resulting in severe damage to the project.

Recommendation

It is recommended that the team be transparent regarding the initial token distribution process. The token distribution plan should be published in a public location that the community can access. The team should make efforts to restrict access to the private keys of the deployer account or EOAs. A multi-signature (⅔, ⅗) wallet can be used to prevent a single point of failure due to a private key compromise. Additionally, the team can lock up a portion of tokens, release them with a vesting schedule for long-term success, and deanonymize the project team with a third-party KYC provider to create greater accountability.

In order for CertiK to update the status of this finding during the remediation phase, please kindly provide the URL to the published token distribution plan and the multi-signature wallet address that holds the undistributed tokens. We will verify the information and update the report. Thank you.

Link to the token distribution plan: https://www…

Multi-sig wallet address: 0x…

Signer 1: 0x…

Signer 2: 0x…

Signer 3: 0x…

LIG-03 | CENTRALIZED CONTROL OF CONTRACT UPGRADE

Category	Severity	Location	Status
Centralization	● Centralization	Smart-Contracts-main/ModelSlasher.sol: 26	● Pending

Description

In the contract `ModelSlasher` , the role `admin` has the authority to update the implementation contract behind the proxy contract.

Any compromise to the `admin` account may allow a hacker to take advantage of this authority and change the implementation contract which is pointed by proxy and therefore execute potential malicious functionality in the implementation contract.

Recommendation

We recommend that the team make efforts to restrict access to the admin of the proxy contract. A strategy of combining a time-lock and a multi-signature (⅔, ⅗) wallet can be used to prevent a single point of failure due to a private key compromise. In addition, the team should be transparent and notify the community in advance whenever they plan to migrate to a new implementation contract.

Here are some feasible short-term and long-term suggestions that would mitigate the potential risk to a different level and suggestions that would permanently fully resolve the risk.

Short Term:

A combination of a time-lock and a multi signature (⅔, ⅗) wallet mitigate the risk by delaying the sensitive operation and avoiding a single point of key management failure.

- A time-lock with reasonable latency, such as 48 hours, for awareness of privileged operations;
  AND
- Assignment of privileged roles to multi-signature wallets to prevent a single point of failure due to a private key compromised;
  AND
- A medium/blog link for sharing the time-lock contract and multi-signers addresses information with the community.

For remediation and mitigated status, please provide the following information:

- Provide the deployed time-lock address.
- Provide the **gnosis** address with **ALL** the multi-signer addresses for the verification process.

- Provide a link to the **medium/blog** with all of the above information included.

Long Term:

A combination of a time-lock on the contract upgrade operation and a DAO for controlling the upgrade operation mitigate the contract upgrade risk by applying transparency and decentralization.

- A time-lock with reasonable latency, such as 48 hours, for community awareness of privileged operations;
  AND
- Introduction of a DAO, governance, or voting module to increase decentralization, transparency, and user involvement;
  AND
- A medium/blog link for sharing the time-lock contract, multi-signers addresses, and DAO information with the community.

For remediation and mitigated status, please provide the following information:

- Provide the deployed time-lock address.
- Provide the **gnosis** address with **ALL** the multi-signer addresses for the verification process.
- Provide a link to the **medium/blog** with all of the above information included.

Permanent:

Renouncing ownership of the `admin` account or removing the upgrade functionality can *fully* resolve the risk.

- Renounce the ownership and never claim back the privileged role;
  OR
- Remove the risky functionality.

Note: we recommend the project team consider the long-term solution or the permanent solution. The project team shall make a decision based on the current state of their project, timeline, and project resources.

LIG-04 | CENTRALIZATION RELATED RISKS

Category	Severity	Location	Status
Centralization	● **Centralization**	**Smart-Contracts-main/ModelAccessCredits.sol: 120~124, 175, 180, 185, 190, 198, 232, 245, 251; Smart-Contracts-main/ModelDAO.sol: 218; Smart-Contracts-main/ModelRegistry.sol: 117; Smart-Contracts-main/ModelReward.sol: 144, 194~198, 215, 234~240, 328, 335, 342, 349; Smart-Contracts-main/ModelSlasher.sol: 274~278, 372, 379, 390, 391; Smart-Contracts-main/ModelToken.sol: 37; Smart-Contracts-main/ModelUpgradeProxy.sol: 149~153, 186, 193, 205, 206; Smart-Contracts-main/ModelValidatorRegistry.sol: 125, 135, 143, 150, 151; Smart-Contracts-main/ModelValidatorStakingPool.sol: 163~167, 182~186, 196, 207, 213**	● **Pending**

Description

ModelAccessCredits.sol

In the contract `ModelAccessCredits` , the role `timelock` has authority over the following functions.

- `updateCreditPrice`
- `updateMinPurchase`
- `updateMaxPurchase`
- `updateExpirationPeriod`
- `refundCredits`
- `withdrawEth`
- `addOperator`
- `removeOperator`

Any compromise to the `timelock` account may allow the hacker to take advantage of this authority and update the basic configuration of the contract, extract the native token in the contract, or add or remove operators at will.

The role `operators` has authority over the following functions.

- `useCredits`

Any compromise to the `operators` account may allow the hacker to take advantage of this authority and use the credits of a specified user.

ModelDAO.sol

In the contract `ModelDAO` , the role `timelock` has authority over the following functions.

- `updateProtocolParams`

Any compromise to the `timelock` account may allow the hacker to take advantage of this authority and update protocol parameters at will.

Additionally, the `ModelDAO` contract inherits the `Ownable` contract from OpenZeppelin, the owner has the following authorities within the contract:

- `renounceOwnership()` : Leaves the contract without owner;
- `transferOwnership()` : Transfers ownership of the contract to a new account.

Any compromise to the `_owner` account may allow the hacker to take advantage of this authority and renounce the ownership status or transfer ownership to a new owner.

ModelRegistry.sol

In the contract `ModelRegistry` , the role `governance` has authority over the following functions.

- `validateModel`

Any compromise to the `governance` account may allow the hacker to take advantage of this authority and modify the status of a specified modelId at will.

ModelReward.sol

In the contract `ModelReward` , the role `modelDAO` has authority over the following functions.

- `withdrawModelToken`
- `updateRewardParams`
- `updateValidatorBonus`
- `issueReward`
- `addValidator`
- `removeValidator`
- `addInferencer`
- `removeInferencer`

Any compromise to the `modelDAO` account may allow the hacker to take advantage of this authority and change the configuration of the contract at will, withdraw the model token in the contract, issue rewards to specified users, or add validators and inferencers.

Additionally, the `ModelReward` contract inherits the `Ownable` contract from OpenZeppelin; the owner has the following authorities within the contract:

- `renounceOwnership()` : Leaves the contract without owner;
- `transferOwnership()` : Transfers ownership of the contract to a new account.

Any compromise to the `_owner` account may allow the hacker to take advantage of this authority and renounce the ownership status or transfer ownership to a new owner.

ModelSlasher.sol

In the contract `ModelSlasher` , the role `_dao` or the `owner (ownership is transferred to provided` _dao` in the constructor) has authority over the following functions.

- `resolveReport`
- `signalBond`
- `applyBond`
- `pause()`
- `unpause()`

Any compromise to the `_dao` account may allow the hacker to take advantage of this authority and set any bond amount, resolve the report to update the status of the report ID and fund, and modify the pause to affect the executable status of some functions of the contract.

Additionally, the `ModelSlasher` contract inherits the `Ownable2StepUpgradeable` contract from OpenZeppelin, the owner has the following authorities within the contract:

- `renounceOwnership()` : Leaves the contract without owner;
- `transferOwnership()` : Transfers ownership of the contract to a new account.

Any compromise to the `_owner` account may allow the hacker to take advantage of this authority and renounce the ownership status or transfer ownership to a new owner.

ModelToken.sol

In the contract `ModelToken` , the role `_owner` has authority over the following functions.

- `mint`

Any compromise to the `_owner` account may allow the hacker to take advantage of this authority and mint tokens to specific users under the `MAX_SUPPLY` limit.

Additionally, the `ModelToken` contract inherits the `Ownable` contract from OpenZeppelin; the owner has the following authorities within the contract:

- `renounceOwnership()` : Leaves the contract without owner;
- `transferOwnership()` : Transfers ownership of the contract to a new account.

Any compromise to the `_owner` account may allow the hacker to take advantage of this authority and renounce the ownership status or transfer ownership to a new owner.

ModelUpgradeProxy.sol

In the contract `ModelUpgradeProxy` , the role `_dao` or the role `owner` has authority over the following functions.

- `resolveReport`
- `signalBond`
- `applyBond`
- `pause`
- `unpause`

Any compromise to the `_dao` account may allow the hacker to take advantage of this authority and set any bond amount, resolve the report to update the status of the report ID and fund, and modify the pause status to affect the executable status of some functions of the contract.

Additionally, the `ModelUpgradeProxy` contract inherits the `Ownable` contract from OpenZeppelin; the owner has the following authorities within the contract:

- `renounceOwnership()` : Leaves the contract without owner;
- `transferOwnership()` : Transfers ownership of the contract to a new account.

Any compromise to the `_owner` account may allow the hacker to take advantage of this authority and renounce the ownership status or transfer ownership to a new owner.

ModelValidatorRegistry.sol

In the contract `ModelValidatorRegistry` , the role `_dao` has authority over the following functions.

- `toggleStatus`
- `setMaxSilence`
- `pause`
- `unpause`

Any compromise to the `_dao` account may allow the hacker to take advantage of this authority and modify the active state of the validator, set a malicious `maxSilence` value, or modify pause status to affect the executable status of some functions of the contract.

The role `_owner` has authority over the following functions.

- `setStakingPool`
- `pause`
- `unpause`

However, according to the current logic, `stakingPool` has been set to a non-address (0) account in the constructor, so this function will not be available after the contract is deployed.

Additionally, the `ModelValidatorRegistry` contract inherits the `Ownable` contract from OpenZeppelin; the owner has the following authorities within the contract:

- `renounceOwnership()` : Leaves the contract without owner;

- `transferOwnership()` : Transfers ownership of the contract to a new account.

Any compromise to the `_owner` account may allow the hacker to take advantage of this authority and renounce the ownership status or transfer ownership to a new owner.

ModelValidatorStakingPool.sol

In the contract `ModelValidatorStakingPool` , the role `_dao` has authority over the following functions.

- `slash`
- `updateParams`
- `pause`
- `unpause`

Any compromise to the `_dao` account may allow the hacker to take advantage of this authority and slash a specified validator's active stake, update relevant configurations in the contract, and modify pause status to affect the executable status of some functions of the contract.

The role `_owner` has authority over the following functions.

- `updateDAO`
- `pause`
- `unpause`

Any compromise to the `_owner` account may allow the hacker to take advantage of this authority and update a new DAO account in the contract or modify the pause status to affect the executable status of some functions of the contract.

Additionally, the `ModelValidatorStakingPool` contract inherits the `Ownable` contract from OpenZeppelin; the owner has the following authorities within the contract:

- `renounceOwnership()` : Leaves the contract without owner;
- `transferOwnership()` : Transfers ownership of the contract to a new account.

Any compromise to the `_owner` account may allow the hacker to take advantage of this authority and renounce the ownership status or transfer ownership to a new owner.

Recommendation

The risk describes the current project design and potentially makes iterations to improve in the security operation and level of decentralization, which in most cases cannot be resolved entirely at the present stage. We advise the client to carefully manage the privileged account's private key to avoid any potential risks of being hacked. In general, we strongly recommend centralized privileges or roles in the protocol be improved via a decentralized mechanism or smart-contract-based accounts with enhanced security practices, e.g., multisignature wallets.

Indicatively, here are some feasible suggestions that would also mitigate the potential risk at a different level in terms of short-term, long-term and permanent:

Short Term:

Timelock and Multi sign (⅔, ⅗) combination *mitigate* by delaying the sensitive operation and avoiding a single point of key management failure.

- Time-lock with reasonable latency, e.g., 48 hours, for awareness on privileged operations;
  AND
- Assignment of privileged roles to multi-signature wallets to prevent a single point of failure due to the private key compromised;
  AND
- A medium/blog link for sharing the timelock contract and multi-signers addresses information with the public audience.

Long Term:

Timelock and DAO, the combination, *mitigate* by applying decentralization and transparency.

- Time-lock with reasonable latency, e.g., 48 hours, for awareness on privileged operations;
  AND
- Introduction of a DAO/governance/voting module to increase transparency and user involvement.
  AND
- A medium/blog link for sharing the timelock contract, multi-signers addresses, and DAO information with the public audience.

Permanent:

Renouncing the ownership or removing the function can be considered *fully resolved*.

- Renounce the ownership and never claim back the privileged roles.
  OR
- Remove the risky functionality.

LIG-05 | INCORRECT ASSEMBLY IMPLEMENTATION

Category	Severity	Location	Status
Logical Issue	● Major	Smart-Contracts-main/ModelSlasher.sol: 565; Smart-Contracts-main/ModelUpgradeProxy.sol: 248	● Pending

Description

By design, the protocol uses Assembly to retrieve the byte length of a string (`string`) in memory.

ModelUpgradeProxy.sol:

```
library StrLib {
    /// @dev gas-cheaper than bytes(str).length in many cases
    function bytesLength(string memory s) internal pure returns (uint256 l) {
@>      assembly { l := mload(add(s, 0x20)) }
    }
}
```

ModelSlasher.sol:

```
    function utf8Lengths(string memory str) internal pure returns (uint256
byteLength, uint256 charCount) {
        // Get raw memory pointer and length
        assembly {
@>          byteLength := mload(add(str, 0x20))
        }
        ...
    }
```

In Solidity, strings are represented in memory as follows:

- When a string (or byte array) is stored in memory, the **first 32 bytes (0x20)** store the byte length of the string (in bytes).
- The **actual string content** (each character occupying 1 byte in UTF-8 encoding) begins at the subsequent 32-byte offset.

Therefore, the original code actually returns the **first 32 bytes of the string's content** instead of the correct length value.

Proof of Concept

```
// SPDX-License-Identifier: UNLICENSED
pragma solidity ^0.8.13;

import {Test, console} from "forge-std/Test.sol";

library StrLib {
    /// @dev gas-cheaper than bytes(str).length in many cases
    function bytesLength(string memory s) internal pure returns (uint256 l) {
        assembly { l := mload(add(s, 0x20)) }
    }
}

contract CounterTest is Test {

    function setUp() public {}

    function test_string_length() external{
        string memory a = "test-lightchain";
        uint256 length = StrLib.bytesLength(a);
        console.log("length:", length);

    }
}
```

Output is:

```
Ran 1 test for test/Counter.t.sol:CounterTest
[PASS] test_string_length() (gas: 3764)
Logs:
  length:
49036020284759792915028833152838264576069831859944842693078476766562228371456

Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 622.96µs (47.13µs CPU
time)
```

Recommendation

Recommend modifying the implementation to correctly return the string's length.

LIG-06 | ACCESS CONTROL MISALIGNMENT IN TIMELOCK GOVERNANCE

Category	Severity	Location	Status
Design Issue	● Major	Smart-Contracts-main/ModelDAO.sol: 274, 293	● Pending

Description

In the `ModelDAO` contract, a governance mechanism combining `Governor` and `Timelock` is used. As a result, all actions that go through the full proposal lifecycle — including propose, vote, and queue — are ultimately executed by the pre-defined `Timelock` contract.

@openzeppelin/contracts/governance/extensions/GovernorTimelockControl.sol:

```
   function _executeOperations(
         uint256 proposalId,
         address[] memory targets,
         uint256[] memory values,
         bytes[] memory calldatas,
         bytes32 descriptionHash
     ) internal virtual override {
         // execute
         _timelock.executeBatch{value: msg.value}(targets, values, calldatas, 0,
 _timelockSalt(descriptionHash));
         // cleanup for refund
         delete _timelockIds[proposalId];
     }
```

However, the `createRewardBatch()` and `distributeRewardBatch()` functions include strict checks `require(msg.sender == address(this))`, allowing them to be called only by the contract itself. This restriction conflicts with the governance flow, where execution is performed by the `Timelock` contract rather than the contract itself. As a result, if such functions are executed via proposals or by any caller other than `address(this)`, they will fail and become permanently inaccessible.

In addition, some contracts transfer ownership or restrict access to specific functions in their constructors, assigning those permissions to a provided `dao` address.

ModelReward.sol:

```
    constructor(address dao, address _modelToken) Ownable(msg.sender) {
        require(dao != address(0), "DAO zero addr");
        require(_modelToken != address(0), "token zero addr");

        modelDAO = dao;  // Now immutable
        rewardAddress = address(this);
        modelToken = IERC20(_modelToken);

        // Initialize protocol parameters
        baseReward = 1 ether;
        epochDuration = 1 days;
        maxRewardPerEpoch = baseReward;         // 1× cap initially

        emit ContractInitialized(dao, _modelToken, baseReward, epochDuration);
    }
```

If the contract adopts the same governance model as `ModelDAO` , it is essential to ensure that the assigned `dao` address is actually the `Timelock` contract instead of the `Governor` , since the `Timelock` is the component that performs the actual execution. Failing to do so may result in governance proposals being unable to trigger critical functions as intended.

Proof of Concept

Simulate the execution of the `createRewardBatch()` function through governance and timelock mechanism.

```
contract ModelDAOTest is ModelBaseTest {
    address public voter1 = makeAddr("voter1");
    address public voter2 = makeAddr("voter2");
    address public voter3 = makeAddr("voter3");

    function setUp() public override{
        super.setUp();
        timelock.grantRole(timelock.PROPOSER_ROLE(), address(dao));
        timelock.grantRole(timelock.EXECUTOR_ROLE(), address(dao));
        timelock.revokeRole(timelock.DEFAULT_ADMIN_ROLE(), address(this));

        // Setup initial token distribution
        vm.startPrank(initialVotingHolder);
        votingToken.transfer(voter1, 10_000 ether);
        votingToken.transfer(voter2, 8_000 ether);
        votingToken.transfer(voter3, 5_000 ether);
        votingToken.transfer(proposerA, 5_000 ether);
        vm.stopPrank();
    }

    function testCreateRewardBatchThroughGovernance() public {
        address[] memory recipients = new address[](2);
        recipients[0] = voter1;
        recipients[1] = voter2;

        uint256[] memory amounts = new uint256[](2);
        amounts[0] = 100 ether;
        amounts[1] = 200 ether;

        vm.startPrank(initialVotingHolder);
        votingToken.transfer(voter1, 1_000_000 ether);
        votingToken.transfer(voter2, 800_000 ether);
        votingToken.transfer(voter3, 500_000 ether);
        vm.stopPrank();

        vm.startPrank(voter1);
        votingToken.delegate(voter1);
        vm.stopPrank();

        vm.startPrank(voter2);
        votingToken.delegate(voter2);
        vm.stopPrank();

        vm.startPrank(voter3);
        votingToken.delegate(voter3);
        vm.stopPrank();

        vm.startPrank(proposerA);
        votingToken.delegate(proposerA);
        vm.stopPrank();

        vm.roll(block.number + 1);

        // encode proposal parameters
        address[] memory targets = new address[](1);
        targets[0] = address(dao);

        uint256[] memory values = new uint256[](1);
        values[0] = 0;

        bytes[] memory calldatas = new bytes[](1);
        calldatas[0] = abi.encodeWithSelector(
            dao.createRewardBatch.selector,
            recipients,
            amounts
        );

        vm.startPrank(proposerA);
        uint256 proposalId = dao.propose(
            targets,
            values,
            calldatas,
            "Create reward batch through governance"
        );
        vm.stopPrank();

        vm.roll(block.number + dao.votingDelay() + 1);

        // voters vote for this proposal
        vm.startPrank(voter1);
        dao.castVote(proposalId, 1);
        vm.stopPrank();

        vm.startPrank(voter2);
        dao.castVote(proposalId, 1);
        vm.stopPrank();

        vm.startPrank(voter3);
        dao.castVote(proposalId, 1);
        vm.stopPrank();

        (, uint256 forVotes,) = dao.proposalVotes(proposalId);
        // console.log("Voting results - For:", forVotes);

        vm.roll(block.number + dao.votingPeriod() + 1);
        uint8 proposalState = uint8(dao.state(proposalId));
        vm.startPrank(proposerA);
        bytes32 descriptionHash = keccak256(bytes("Create reward batch through
governance"));
        dao.queue(targets, values, calldatas, descriptionHash);
        vm.stopPrank();

        uint256 minDelay = timelock.getMinDelay();
        vm.warp(block.timestamp + minDelay + 1);

        proposalState = uint8(dao.state(proposalId));

        vm.startPrank(executorA);
        // execution will fail because of timelock != dao
        vm.expectRevert("governance only");
        dao.execute(targets, values, calldatas, descriptionHash);
        vm.stopPrank();

    }
}
```

```
[PASS] testCreateRewardBatchThroughGovernance() (gas: 840640)
Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 9.05ms (1.08ms CPU
time)
```

Recommendation

It is recommended to ensure that all governance-executed functions are compatible with being called by the `Timelock` contract. Additionally, when configuring ownership or access control to a DAO address, make sure the designated address matches the `Timelock` executor to avoid disabling critical functionality.

You may refer to OpenZeppelin's documentation on **GovernorTimelockControl** for detailed guidance on correctly configuring timelock-based governance.

LIG-17 | SLASHING BYPASS VIA EARLY WITHDRAWAL REQUEST OF STAKES

Category	Severity	Location	Status
Logical Issue	● Major	Smart-Contracts-main/ModelSlasher.sol: 292~303; Smart-Contracts-main/ModelUpgradeProxy.sol: 163~167	● Pending

Description

In the `resolveReport` function, if the `uphold` flag is set to true, it triggers a call to the `slash()` function in the staking pool to penalize the validator by deducting a portion of their staked `stakingToken` .

```
    function resolveReport(bytes32 id, bool uphold, string calldata slashReason)
        external
        onlyOwner
        nonReentrant
    {
        ...

        if (uphold) {
            // Check current slashable amount
            uint256 slashableAmount = getSlashableAmount(r.validator, r.amount);

            // If any amount can be slashed, proceed with slashing
@>          if (slashableAmount > 0) {
                // Attempt to slash what we can
                stakingPool.slash(r.validator, slashableAmount, slashReason);

                // If partial slash, emit event
                if (slashableAmount < r.amount) {
                    emit PartialSlash(id, r.amount, slashableAmount);
                }
            }

            // Always refund reporter's bond for valid report
            payable(r.reporter).sendValue(bond);
            emit BondRefunded(id, r.reporter);
        } else {
            ...
        }
        emit ReportResolved(id, uphold);
    }
```

However, if the staking pool shares the same logic as the `ModelValidatorStakingPool` contract, a validator could front-run the call to invoke `requestWithdraw()` to withdraw all his staked tokens, which updates `stakes.amount` to 0.

ModelValidatorStakingPool.sol:

```
    function requestWithdraw(uint256 amount) external whenNotPaused {
        require(amount > 0, "zero");
        StakeInfo storage s = stakes[msg.sender];

        // disallow overlapping or cumulative requests
        require(s.pending == 0, "pending exists");
        require(amount <= s.amount, "exceeds stake");

        s.amount      -= amount;
        s.pending      = amount;
        s.unlockTime   = uint64(block.timestamp + unstakeDelay);
        totalStaked   -= amount;

        emit ModelUnstakeRequested(msg.sender, amount, s.unlockTime);

        if (s.amount < minStake && s.amount == 0) {
            _validators.remove(msg.sender);
        }
    }
```

As a result, the computed `slashableAmount` would be 0, causing the `slash()` function to be skipped and allowing the validator to evade punishment.

```
    function getSlashableAmount(
        address validator,
        uint256 requestedAmount
    ) public view returns (uint256 slashableAmount) {
        (uint256 liveStake,,) = stakingPool.stakes(validator);
        return liveStake >= requestedAmount ? requestedAmount : liveStake;
    }
```

This issue also exists in the `ModelUpgradeProxy` contract, where validators can avoid being slashed in a similar way.

Proof of Concept

In this proof of concept, Alice first staked tokens in the staking pool, and later Bob submitted a report designating Alice as the validator. However, Alice preemptively called `requestWithdraw()` , reducing her staked amount to zero. As a result, the slashing logic was bypassed, and no penalty was applied.

```
    function testSlashBypass() public {
        stakingToken.mint(Alice, 1_000 ether);
        vm.deal(Alice, 1 ether);
        vm.deal(Bob, 1 ether);

        uint256 alice_stakingToken_balance_before_ops =
stakingToken.balanceOf(Alice);
        uint256 bob_ETH_balance_before_ops = Bob.balance;

        vm.startPrank(Alice);

        stakingToken.approve(address(stakingPool), 1_000 ether);
        stakingPool.stake(1_000 ether);
        vm.stopPrank();

        // Bob submits a report, the validator is Alice
        vm.startPrank(Bob);
        bytes32 id = slasher.submitReport{value: 1 ether}(
            Alice,
            1 ether,
            "Just for testing",
            bytes32(uint256(1))
        );
        vm.stopPrank();

        // Alice requests withdraw in advance
        vm.startPrank(Alice);
        stakingPool.requestWithdraw(1_000 ether);
        vm.stopPrank();

        // Reminder: The specific logic of the entire voting and timelock is not
simulated here, and dao is used directly to complete the call to this privileged
function
        vm.startPrank(address(dao));
        slasher.resolveReport(id, true, "Just for testing");
        vm.stopPrank();

        vm.warp(block.timestamp+7 days);
        vm.startPrank(Alice);
        stakingPool.withdraw();
        vm.stopPrank();

        uint256 bob_ETH_balance_after_ops = Bob.balance;
        uint256 alice_stakingToken_balance_after_ops =
stakingToken.balanceOf(Alice);

        // after all submit report or slash, Alice and Bob's balances remain the
same as at the beginning.
```

```
        assertEq(alice_stakingToken_balance_before_ops,
alice_stakingToken_balance_after_ops);
        assertEq(bob_ETH_balance_before_ops, bob_ETH_balance_after_ops);
    }
```

```
[PASS] testSlashBypass() (gas: 564392)
Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 2.89ms (229.54µs CPU
time)
```

Recommendation

It is recommended to revisit the overall staking and slashing logic to ensure that a validator cannot evade punishment by manipulating withdrawal timing.

LIG-18 IMPROPER AUTHORIZATION LOGIC MAY BLOCK `slash` RELEVANT FUNCTIONS

Category	Severity	Location	Status
Access Control	● Major	Smart-Contracts-main/ModelValidatorStakingPool.sol: 163~167	● Pending

Description

The `slash` function is restricted to be callable only by the DAO account. If the current contract follows the same logic as the `stakingPool` in the `ModelSlasher` and `ModelUpgradeProxy`, then both this contract and another one will fail to invoke slash during the execution of `resolveReport` due to too strict access control issues.

```
    function slash(
        address validator,
        uint256 amount,
        string calldata reason
    ) external onlyDAO nonReentrant {
        ...
    }
```

As a result, even when `upheld == true` and `slashableAmount > 0`, the `resolveReport` function will remain unexecutable, preventing valid reports from being properly resolved.

```
    function resolveReport(bytes32 id, bool uphold, string calldata slashReason)
        external
        onlyOwner
        nonReentrant
    {
        ...

        if (uphold) {
            // Check current slashable amount
            uint256 slashableAmount = getSlashableAmount(r.validator, r.amount);

            // If any amount can be slashed, proceed with slashing
            if (slashableAmount > 0) {
                // Attempt to slash what we can
@>              stakingPool.slash(r.validator, slashableAmount, slashReason);
                ...
            }

            ...
        } else {
            ...
        }
        emit ReportResolved(id, uphold);
    }
```

Proof of Concept

```
    function testSlashRevert() public {
        stakingToken.mint(Alice, 1_000 ether);
        vm.deal(Alice, 1 ether);
        vm.deal(Bob, 1 ether);

        vm.startPrank(Alice);

        stakingToken.approve(address(stakingPool), 1_000 ether);
        stakingPool.stake(1_000 ether);
        vm.stopPrank();

        // Bob submits a report, the validator is Alice
        vm.startPrank(Bob);
        bytes32 id = slasher.submitReport{value: 1 ether}(
            Alice,
            1 ether,
            "Just for testing",
            bytes32(uint256(1))
        );
        vm.stopPrank();

        // Reminder: The specific logic of the entire voting and timelock is not
simulated here, and dao is used directly to complete the call to this privileged
function
        vm.startPrank(address(dao));
        vm.expectRevert("DAO only");
        slasher.resolveReport(id, true, "Just for testing");
        vm.stopPrank();
    }
```

```
[PASS] testSlashRevert() (gas: 544023)
Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 8.26ms (826.25µs CPU
time)
```

Recommendation

It is recommended to ensure that the authorization logic for slashing is compatible across contracts, or to separate slashing execution from privileged access paths to avoid blocking report resolution under valid conditions.

LIG-07 | POTENTIAL FRONT-RUN ON PERMIT CALL TO CAUSE DOS

Category	Severity	Location	Status
Denial of Service	● Medium	Smart-Contracts-main/ModelValidatorStakingPool.sol: 86~88	● Pending

Description

Contracts interfacing with ERC20 tokens may implement a feature in the `stakeWithPermit` function where users provide signatures to approve `ERC20Permit(address(stakingToken))` transfers, subsequently invoking the `permit` function to finalize the approval.

However, the contract fails to address the possibility of transaction front-running, where an unauthorized party could preemptively execute the `permit` function before the intended transaction is processed.

If malicious actors observe this process, they can preemptively execute the `permit` function with the user's signature and nonce. This unauthorized front-running as a griefing attack not only consumes the nonce, rendering the legitimate user's intended transaction invalid but also can lead to a targeted Denial of Service (DOS) for the affected function, as the user's approval process is effectively disrupted.

Recommendation

To mitigate the risk of a griefing attack, it is recommended to verify the current allowance for the token before invoking the `permit` function, ensuring that the signature has not already been utilized by a malicious actor.

LIG-19 | POTENTIAL DELETE REPORTS BY MISTAKE

Category	Severity	Location	Status
Logical Issue	● Medium	Smart-Contracts-main/ModelSlasher.sol: 504~506, 504~506	● Pending

Description

In the `ModelSlasher` contract, the `leaveBatch()` function allows users to batch cancel reports in a pending state. Users can specify a starting index and the number of pending reports to cancel.

However, when `startIndex + batchSize >= userIds.length` , the function will remove all of the user's reports, including those not in a pending state, which contradicts the intended design of the function.

```
    function leaveBatch(
        uint256 startIndex,
        uint256 maxBatch
    ) external whenNotPaused nonReentrant returns (
        int256 nextIndex,
        uint256 processedCount,
        uint256 refundAmount
    ) {
        ...
        nextIndex = startIndex + batchSize < userIds.length ?
            int256(startIndex + batchSize) : -1;

        // If this was the last batch, clear the reports array
        if (nextIndex == -1) {
@>          delete userReports[msg.sender];
        }

        return (nextIndex, processedCount, refundTotal);
    }
```

Recommendation

We recommend redesigning this logic to prevent such issues.

LIG-20 | INSUFFICIENT QUORUM THRESHOLD ALLOWS PROPOSAL MANIPULATION

Category	Severity	Location	Status
Governance	● Medium	Smart-Contracts-main/ModelDAO.sol: 78	● Pending

Description

In the governance mechanism, a proposal can only proceed if the combined number of `forVotes` and `abstainVotes` meets or exceeds the quorum. However, in the constructor of the `ModelDAO` contract, the quorum is set to 4% of the current votingToken total supply. If the token supply is relatively low, a single user or a small group of users could meet the quorum requirement solely with their voting power, thereby influencing proposal outcomes and potentially executing malicious actions.

Recommendation

It is recommended to set a higher quorum threshold at deployment or enforce additional safeguards, especially when the system has low initial token distribution. Additionally, monitoring the total voting token supply and dynamically adjusting the quorum requirement can further mitigate the risk of governance capture.

LIG-23 | UNCOORDINATED GOVERNANCE PARAMETER MANAGEMENT

Category	Severity	Location	Status
Logical Issue	● Medium	Smart-Contracts-main/ModelDAO.sol: 93~103	● Pending

Description

According to the protocol design, the `ModelDAO` contract defines a set of parameters intended to serve as global configuration for the system.

```
        protocolParams = ProtocolParams({
            minStakeAmount:      1_000 ether,
            maxStakeAmount:    100_000 ether,
            rewardMultiplier:        100,
            slashingPenalty:          50,
            proposalThreshold:    1_000 ether,
            votingPeriod:     MIN_VOTING_PERIOD,
            quorumNumerator:           4,
            minValidators:             3,
            minValidationScore:       80
        });
```

However, identical parameters are redefined and maintained independently across multiple contracts. This indicates that each contract operates with its own configuration, rather than consistently referencing values from the central governance source (`ModelDAO`).

Such an approach may deviate from the original design intent of centralized governance control and introduces potential risks related to configuration inconsistency, upgrade complexity, and governance fragmentation.

Recommendation

Standardize critical parameter access by ensuring all relevant contracts reference configuration values directly from the `ModelDAO` contract or a dedicated configuration module. This will enhance maintainability, consistency, and governance transparency.

LIG-08 | USAGE OF `transfer()` FOR SENDING NATIVE TOKENS

Category	Severity	Location	Status
Coding Style	● Minor	Smart-Contracts-main/ModelAccessCredits.sol: 112, 226, 240	● Pending

Description

After EIP-1884 was included in the Istanbul hard fork, it is not recommended to use `.transfer()` or `.send()` for transferring native tokens as these functions have a hard-coded value for gas costs making them obsolete as they are forwarding a fixed amount of gas, specifically `2300` . This can cause issues in case the linked statements are meant to be able to transfer funds to other contracts instead of EOAs.

Recommendation

We recommend using the `sendValue()` function of `Address` contract from OpenZeppelin. See https://github.com/OpenZeppelin/openzeppelin-contracts/blob/v5.4.0/contracts/utils/Address.sol.

LIG-09 | NO WAY TO RETRIEVE ETH FROM THE CONTRACT

Category	Severity	Location	Status
Volatile Code	● Minor	Smart-Contracts-main/ModelReward.sol: 533; Smart-Contracts-main/ModelValidatorStakingPool.sol: 233	● Pending

Description

The identified contracts have at least one payable function that does not utilize the forwarded ETH. Additionally, these contracts lack a mechanism to withdraw ETH. As a result, any ETH sent to these contracts may become permanently trapped.

Recommendation

Consider adding a withdraw function to contracts that are capable of receiving ether.

LIG-10 | POTENTIALLY UNUSABLE FUNCTION

Category	Severity	Location	Status
Design Issue	● Minor	Smart-Contracts-main/ModelValidatorRegistry.sol: 135~140	● Pending

Description

In the `ModelValidatorRegistry` contract, because `_stakingPool` is validated to be a non-zero address during deployment, `stakingPool` will never be the zero address.

```
    constructor(address _dao, address _stakingPool) Ownable(msg.sender) {
@>      require(_dao != address(0) && _stakingPool != address(0), "zero addr");

        ...
        stakingPool = IModelValidatorStakingPool(_stakingPool);

        emit StakingPoolSet(_stakingPool);
        emit DAOChanged(address(0), _dao);
    }
```

However, the `setStakingPool()` function requires `stakingPool` to be zero, which makes `setStakingPool()` permanently unusable.

```
    function setStakingPool(address pool) external onlyOwner {
@>      require(address(stakingPool) == address(0), "already set");
        require(pool != address(0) && IModelValidatorStakingPool(pool).minStake() >
0, "bad pool");
        stakingPool = IModelValidatorStakingPool(pool);
        emit StakingPoolSet(pool);
    }
```

Recommendation

Recommend redesigning this function according to requirements.

LIG-11 | POTENTIAL MISMATCH BETWEEN DELEGATED VOTES AND TOKEN BALANCE

Category	Severity	Location	Status
Logical Issue	● Minor	Smart-Contracts-main/ModelToken.sol: 42~44, 50~52	● Pending

Description

When transferring tokens, if the `to` address has no delegatee (i.e., the delegatee is the zero address), then after the transfer is completed, the delegated votes for the `to` address remain zero.

```
    function _update(address from, address to, uint256 value) internal
override(ERC20, ERC20Votes) {
        super._update(from, to, value);
    }
```

Although users can manually call the `delegate()` function to update their delegated votes, in some scenarios, the absence of this step may cause the entire transaction to fail.

Additionally, this behavior is inconsistent with the design of the `mint` function.

```
    function mint(address to, uint256 amount) external onlyOwner {
        require(to != address(0), "zero addr");
        require(totalSupply() + amount <= MAX_SUPPLY, "cap exceeded");
        _mint(to, amount);

        if (delegates(to) == address(0)) {
            _delegate(to, to);
        }
    }
```

Proof of Concept

```
// SPDX-License-Identifier: UNLICENSED
pragma solidity ^0.8.13;

import {Test, console} from "forge-std/Test.sol";
import "../src/Smart-Contracts-main/ModelToken.sol";

contract PoCTest is Test {
    ModelToken token;
    address owner = address(0x999);
    address alice = address(0x111);
    address bob = address(0x222);

    function setUp() public {
        token = new ModelToken(owner, 10000000e18);
    }

    function test_votes() external {
        uint256 aliceVotes;
        uint256 bobVotes;
        vm.startPrank(owner);
        token.mint(alice, 1000e18);
        aliceVotes = token.getVotes(alice);
        console.log("alice votes after mint:", aliceVotes/1e18);
        console.log("bob votes:", bobVotes/1e18);

        vm.startPrank(alice);
        token.transfer(bob, token.balanceOf(alice));
        aliceVotes = token.getVotes(alice);
        bobVotes = token.getVotes(bob);
        console.log("alice votes after transfer to bob:", aliceVotes/1e18);
        console.log("bob votes:", bobVotes/1e18);

    }
}
```

Output is:

```
Ran 1 test for test/PoC.t.sol:PoCTest
[PASS] test_votes() (gas: 167242)
Logs:
  alice votes after mint: 1000
  bob votes: 0
  alice votes after transfer to bob: 0
  bob votes: 0

Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 885.54µs (216.88µs CPU
time)

Ran 1 test suite in 128.45ms (885.54µs CPU time): 1 tests passed, 0 failed, 0
skipped (1 total tests)
```

Recommendation

It is recommended to modify the design so that the delegated votes of a user are updated in real-time along with token transfers.

LIG-12 | DEAD CODE

Category	Severity	Location	Status
Coding Issue	● Informational	Smart-Contracts-main/ModelDAO.sol: 351~354	● Pending

Description

One or more internal functions are not used.

```
    function _afterExecute(uint256 id) internal {
```

Recommendation

We recommend removing those unused functions.

LIG-13 RISK OF INSUFFICIENT NATIVE TOKEN BALANCE DURING REFUNDS

Category	Severity	Location	Status
Logical Issue	● Informational	Smart-Contracts-main/ModelAccessCredits.sol: 209; Smart-Contracts-main/ModelSlasher.sol: 306, 310	● Pending

Description

ModelAccessCredits.sol

In the `refundCredits` function of the `ModelAccessCredits` contract, users are allowed to refund their credits, and the contract calculates the amount of native token to be returned based on the current `creditPrice`. If the current `creditPrice` is higher than the price at which the user originally purchased the credits, the contract will also refund the difference. According to the comments, this appears to be the intended design.

However, it is important for the team to ensure that sufficient native tokens are replenished in the contract in a timely manner. Otherwise, user operations might revert due to insufficient balance, or worse, result in one user unintentionally withdrawing another user's native tokens.

ModelSlasher.sol

Additionally, in the `resolveReport` function of the `ModelSlasher` contract, the current `bond` value is used as the refund amount returned to the reporter or treasury. However, since the bond can still be updated after contract deployment, if this is the intended design, the team should ensure that the contract holds sufficient native tokens to cover the increased refund amount when the bond is raised.

Recommendation

It is recommended to ensure that the contract maintains a sufficient balance of native tokens at all times.

LIG-14 USE OF `code.length` IS NO LONGER A RELIABLE CONTRACT CHECK

Category	Severity	Location	Status
Design Issue	● Informational	Smart-Contracts-main/ModelAccessCredits.sol: 57~60; Smart-Contracts-main/ModelValidatorRegistry.sol: 67~71	● Pending

Description

The constructor uses `extcodesize(account) > 0` to validate the provided `_timelock` and `_dao` addresses as contracts. However, following the introduction of EIP-7702, externally owned accounts (EOAs) can temporarily or permanently have contract code, which may cause this check to succeed even if the address does not implement the expected interface. If such an address is accepted, subsequent contract interactions may revert, potentially rendering the contract unusable.

Although these addresses are set by the admin during deployment, it is still important to alert the development team to this potential issue.

Recommendation

It is recommended to use stricter validation methods or rely on `extcodesize(account) > 0` only as a basic sanity check to confirm that an input is not a plain EOA.

LIG-21 | UNNECESSARY INHERITANCE OF `Ownable`

Category	Severity	Location	Status
Design Issue, Logical Issue	● Informational	Smart-Contracts-main/ModelDAO.sol: 25; Smart-Contracts-main/ModelReward.sol: 14	● Pending

Description

The `ModelDAO` contract serves as the governance core of the protocol, aiming to support decentralized decision-making. According to its design, proposals and voting are carried out by token holders, and proposal execution is managed by a `Timelock` contract.

Given this decentralized governance structure, inheriting from the `Ownable` contract is unnecessary and contradicts the intended trust-minimized design. The presence of an owner role introduces centralized control that should not exist in a DAO context.

This issue also exists in the `ModelReward` contract, where all privileged functions are executed by the DAO account.

Recommendation

It is recommended to remove the inheritance of `Ownable` from the `ModelDAO` and `ModelReward` contracts to align with the protocol's decentralized governance principles.

LIG-01 | DISCUSSION ON VALIDATOR REMOVAL CONDITIONS

Category	Severity	Location	Status
Logical Issue	● Discussion	Smart-Contracts-main/ModelValidatorStakingPool.sol: 136	● Pending

Description

According to the design, when the staked amount of the validator is 0, the validator will be removed from the validator set.

```
    function requestWithdraw(uint256 amount) external whenNotPaused {
        ...

@>      if (s.amount < minStake && s.amount == 0) {
            _validators.remove(msg.sender);
        }
    }
```

However, since `minStake` is always larger than 0, the `s.amount < minStake` judgment is meaningless.

Based on the design, the user must stake an amount greater than `minStake` to qualify as a validator. In theory, when a validator initiates a withdrawal or is penalized, causing their staked amount to fall below the `minStake` threshold (not zero amount), they should be automatically removed from the validator set.

Recommendation

We would like the team to confirm the intended conditions and mechanisms for validator removal under such scenarios.

LIG-15 | UNCLEAR CONTRACT DESIGN

Category	Severity	Location	Status
Design Issue	● Discussion	Smart-Contracts-main/ModelAccessCredits.sol; Smart-Contracts-main/ModelRegistry.sol; Smart-Contracts-main/ModelUpgradeProxy.sol	● Pending

Description

In the **Lightchain** protocol, some contract components appear to lack integration with each other or clear alignment with practical use cases. We have identified the following contracts where the design raises questions:

ModelUpgradeProxy

- Based on its name, this contract seems intended to serve as a proxy for an upgradeable contract. However, it does not contain any functionality typically associated with upgradeable proxies. This raises the question: is the contract misnamed, or has it been implemented incorrectly?
- The contract's actual function allows users to report validators, and if the report is validated, the validator is penalized. This behavior closely overlaps with the functionality provided by the `ModelSlasher` contract. We recommend the team re-evaluate whether the design and role of `ModelUpgradeProxy` are correctly defined and implemented.

ModelAccessCredits

- This contract allows users to purchase credits using native tokens, which can then be consumed by operators. However, the purpose and application of these credits are not clearly articulated within the protocol.

ModelRegistry

- The `submitModel()` and `storeResponse()` functions allow any user to submit arbitrary hash strings. However, the real-world use cases for these functions remain unclear.

Recommendation

We ask the team to provide a detailed explanation of these functions' roles within the broader protocol to help us better understand the system's design rationale.

LIG-16 | CONCERNS REGARDING THE VALIDATOR MECHANISM

Category	Severity	Location	Status
Design Issue	● Discussion	Smart-Contracts-main/ModelAccessCredits.sol; Smart-Contracts-main/ModelDAO.sol; Smart-Contracts-main/ModelValidatorRegistry.sol; Smart-Contracts-main/ModelValidatorStakingPool.sol	● Pending

Description

The **Lightchain** protocol introduces a validator mechanism. In most blockchain systems, it plays a critical role in ensuring protocol security and governance. However, in `Lightchain` , this mechanism is not centrally managed by a unified contract. Instead, multiple contracts maintain separate validator or operator lists without explicit interaction or synchronization between them. This fragmented design raises the following concerns:

ModelAccessCredits

- This contract defines an `operator` role that can consume user-purchased credits. It is unclear whether this role is functionally equivalent to a `validator` . Should validators be more responsible for this action than the operator role?

ModelDAO

- In this contract, holders of `ModelToken` can submit proposals and vote. We would like to know whether such governance rights should instead belong to the validators, given their presumed role in protocol security and decision-making.

ModelValidatorStakingPool

- Users can stake to become validators. However, the validator role derived from staking is not reflected in any clear responsibilities or interactions elsewhere in the system.

ModelValidatorRegistry

- After becoming a validator via the `ModelValidatorStakingPool` , users can register again in this contract as validators. It is unclear what distinction or additional responsibility this second layer of registration introduces.

Recommendation

We recommend the team conduct a thorough review of the validator design across all contracts and evaluate whether the validator logic should be centralized and consistently managed within a single contract or module. This would help ensure clarity, maintainability, and protocol integrity.

LIG-22 | CONCERNS REGARDING THE STAKING MECHANISM

Category	Severity	Location	Status
Logical Issue, Design Issue	● Discussion	Smart-Contracts-main/ModelReward.sol: 234~240; Smart-Contracts-main/ModelValidatorStakingPool.sol: 107~108	● Pending

Description

In the `ModelValidatorStakingPool` contract, users can stake tokens to become validators, with the allowed staking amount ranging from 1,000 to 100,000 tokens. However, throughout the `Lightchain` protocol, we have not identified any logic or mechanism that makes use of the validator's staking amount.

Based on this observation, we would like to raise the following questions regarding the staking design:

1. **Does the amount of tokens staked by a validator have any impact on their role or privileges within the protocol?** For instance, does a higher staking amount grant the validator more voting power, higher influence in validation, or increased access to rewards?

2. **In the `ModelReward` contract, the DAO can call `issueReward()` to distribute rewards. This function requires both a user address and a validator address as inputs, but the reward is only transferred to the user.** Is this behavior consistent with the intended design? If so, what is the designated reward mechanism for validators?

Recommendation

We hope the team will provide clarification on these questions to help us better understand the intended role and incentive structure of validators within the protocol.

FORMAL VERIFICATION | LIGHTCHAIN

Formal guarantees about the behavior of smart contracts can be obtained by reasoning about properties relating to the entire contract (e.g. contract invariants) or to specific functions of the contract. Once such properties are proven to be valid, they guarantee that the contract behaves as specified by the property. As part of this audit, we applied formal verification to prove that important functions in the smart contracts adhere to their expected behaviors.

Considered Functions And Scope

In the following, we provide a description of the properties that have been used in this audit. They are grouped according to the type of contract they apply to.

Verification of Standard Ownable Properties

We verified *partial* properties of the public interfaces of those token contracts that implement the Ownable interface. This involves:

- function `owner` that returns the current owner,
- functions `renounceOwnership` that removes ownership,
- function `transferOwnership` that transfers the ownership to a new owner.

The properties that were considered within the scope of this audit are as follows:

Property Name	Title
ownable-renounceownership-correct	Ownership is Removed
ownable-renounce-ownership-is-permanent	Once Renounced, Ownership Cannot be Regained
ownable-owner-succeed-normal	`owner` Always Succeeds
ownable-transferownership-correct	Ownership is Transferred

Verification of ERC-20 Compliance

We verified properties of the public interface of those token contracts that implement the ERC-20 interface. This covers

- Functions `transfer` and `transferFrom` that are widely used for token transfers,
- functions `approve` and `allowance` that enable the owner of an account to delegate a certain subset of her tokens to another account (i.e. to grant an allowance), and
- the functions `balanceOf` and `totalSupply`, which are verified to correctly reflect the internal state of the contract.

The properties that were considered within the scope of this audit are as follows:

Property Name	Title
erc20-approve-never-return-false	`approve` Never Returns `false`
erc20-approve-revert-zero	`approve` Prevents Approvals For the Zero Address
erc20-approve-false	If `approve` Returns `false` , the Contract's State Is Unchanged
erc20-balanceof-succeed-always	`balanceOf` Always Succeeds
erc20-approve-correct-amount	`approve` Updates the Approval Mapping Correctly
erc20-allowance-correct-value	`allowance` Returns Correct Value
erc20-totalsupply-succeed-always	`totalSupply` Always Succeeds
erc20-allowance-succeed-always	`allowance` Always Succeeds
erc20-approve-succeed-normal	`approve` Succeeds for Valid Inputs
erc20-totalsupply-correct-value	`totalSupply` Returns the Value of the Corresponding State Variable
erc20-balanceof-correct-value	`balanceOf` Returns the Correct Value
erc20-balanceof-change-state	`balanceOf` Does Not Change the Contract's State
erc20-totalsupply-change-state	`totalSupply` Does Not Change the Contract's State
erc20-allowance-change-state	`allowance` Does Not Change the Contract's State

Verification Results

For the following contracts, formal verification established that each of the properties that were in scope of this audit (see scope) are valid:

Detailed Results For Contract ModelReward (Smart-Contracts-main/ModelReward.sol) In SHA256 Checksum 89be0b23404dd5d4a4730bfdfb71dcafe20a3871

Verification of Standard Ownable Properties

Detailed Results for Function `owner`

Property Name	Final Result	Remarks
ownable-owner-succeed-normal	● True	

Detailed Results for Function `renounceOwnership`

Property Name	Final Result	Remarks
ownable-renounceownership-correct	● True	
ownable-renounce-ownership-is-permanent	● True	

Detailed Results for Function `transferOwnership`

Property Name	Final Result	Remarks
ownable-transferownership-correct	● True	

Detailed Results For Contract ModelToken (Smart-Contracts-main/ModelToken.sol) In SHA256 Checksum 89be0b23404dd5d4a4730bfdfb71dcafe20a3871

Verification of ERC-20 Compliance

Detailed Results for Function `approve`

Property Name	Final Result	Remarks
erc20-approve-never-return-false	● True	
erc20-approve-revert-zero	● True	
erc20-approve-false	● True	
erc20-approve-correct-amount	● True	
erc20-approve-succeed-normal	● True	

Detailed Results for Function `balanceOf`

Property Name	Final Result	Remarks
erc20-balanceof-succeed-always	● True	
erc20-balanceof-correct-value	● True	
erc20-balanceof-change-state	● True	

Detailed Results for Function `allowance`

Property Name	Final Result	Remarks
erc20-allowance-correct-value	● True	
erc20-allowance-succeed-always	● True	
erc20-allowance-change-state	● True	

Detailed Results for Function `totalSupply`

Property Name	Final Result	Remarks
erc20-totalsupply-succeed-always	● True	
erc20-totalsupply-correct-value	● True	
erc20-totalsupply-change-state	● True	

In the remainder of this section, we list all contracts where formal verification of at least one property was not successful. There are several reasons why this could happen:

- False: The property is violated by the project.
- Inconclusive: The proof engine cannot prove or disprove the property due to timeouts or exceptions.
- Inapplicable: The property does not apply to the project.

Detailed Results For Contract ModelUpgradeProxy (Smart-Contracts-main/ModelUpgradeProxy.sol) In SHA256 Checksum 89be0b23404dd5d4a4730bfdfb71dcafe20a3871

Verification of Standard Ownable Properties

Detailed Results for Function `renounceOwnership`

Property Name	Final Result	Remarks
ownable-renounceownership-correct	● True	
ownable-renounce-ownership-is-permanent	● Inconclusive	

Detailed Results for Function `owner`

Property Name	Final Result	Remarks
ownable-owner-succeed-normal	● True	

Detailed Results for Function `transferOwnership`

Property Name	Final Result	Remarks
ownable-transferownership-correct	● True	

Detailed Results For Contract ModelValidatorRegistry (Smart-Contracts-main/ModelValidatorRegistry.sol) In SHA256 Checksum 89be0b23404dd5d4a4730bfdfb71dcafe20a3871

Verification of Standard Ownable Properties

Detailed Results for Function `renounceOwnership`

Property Name	Final Result	Remarks
ownable-renounceownership-correct	● True	
ownable-renounce-ownership-is-permanent	● Inconclusive	

Detailed Results for Function `owner`

Property Name	Final Result	Remarks
ownable-owner-succeed-normal	● True	

Detailed Results for Function `transferOwnership`

Property Name	Final Result	Remarks
ownable-transferownership-correct	● True	

Detailed Results For Contract ModelValidatorStakingPool (Smart-Contracts-main/ModelValidatorStakingPool.sol) In SHA256 Checksum 89be0b23404dd5d4a4730bfdfb71dcafe20a3871

Verification of Standard Ownable Properties

Detailed Results for Function `owner`

Property Name	Final Result	Remarks
ownable-owner-succeed-normal	● True	

Detailed Results for Function `renounceOwnership`

Property Name	Final Result	Remarks
ownable-renounceownership-correct	● True	
ownable-renounce-ownership-is-permanent	● Inconclusive	

Detailed Results for Function `transferOwnership`

Property Name	Final Result	Remarks
ownable-transferownership-correct	● True	

APPENDIX | LIGHTCHAIN

Finding Categories

Categories	Description
Coding Style	Coding Style findings may not affect code behavior, but indicate areas where coding practices can be improved to make the code more understandable and maintainable.
Coding Issue	Coding Issue findings are about general code quality including, but not limited to, coding mistakes, compile errors, and performance issues.
Denial of Service	Denial of Service findings indicate that an attacker may prevent the program from operating correctly or responding to legitimate requests.
Access Control	Access Control findings are about security vulnerabilities that make protected assets unsafe.
Volatile Code	Volatile Code findings refer to segments of code that behave unexpectedly on certain edge cases and may result in vulnerabilities.
Logical Issue	Logical Issue findings indicate general implementation issues related to the program logic.
Centralization	Centralization findings detail the design choices of designating privileged roles or other centralized controls over the code.
Governance	Governance findings indicate issues related to the management of the code.
Design Issue	Design Issue findings indicate general issues at the design level beyond program logic that are not covered by other finding categories.

Checksum Calculation Method

The "Checksum" field in the "Audit Scope" section is calculated as the SHA-256 (Secure Hash Algorithm 2 with digest size of 256 bits) digest of the content of each file hosted in the listed source repository under the specified commit.

The result is hexadecimal encoded and is the same as the output of the Linux "sha256sum" command against the target file.

Details on Formal Verification

Some Solidity smart contracts from this project have been formally verified. Each such contract was compiled into a mathematical model that reflects all its possible behaviors with respect to the property. The model takes into account the semantics of the Solidity instructions found in the contract. All verification results that we report are based on that model.

The following assumptions and simplifications apply to our model:

- Certain low-level calls and inline assembly are not supported and may lead to a contract not being formally verified.
- We model the semantics of the Solidity source code and not the semantics of the EVM bytecode in a compiled contract.

Formalism for property specifications

All properties are expressed in a behavioral interface specification language that CertiK has developed for Solidity, which allows us to specify the behavior of each function in terms of the contract state and its parameters and return values, as well as contract properties that are maintained by every observable state transition. Observable state transitions occur when the contract's external interface is invoked and the invocation does not revert, and when the contract's Ether balance is changed by the EVM due to another contract's "self-destruct" invocation. The specification language has the usual Boolean connectives, as well as the operator `\old` (used to denote the state of a variable before a state transition), and several types of specification clause:

Apart from the Boolean connectives and the modal operators "always" (written `[]`) and "eventually" (written `<>`), we use the following predicates to reason about the validity of atomic propositions. They are evaluated on the contract's state whenever a discrete time step occurs:

- `requires [cond]` - the condition `cond`, which refers to a function's parameters, return values, and contract state variables, must hold when a function is invoked in order for it to exhibit a specified behavior.
- `ensures [cond]` - the condition `cond`, which refers to a function's parameters, return values, and both `\old` and current contract state variables, is guaranteed to hold when a function returns if the corresponding requires condition held when it was invoked.
- `invariant [cond]` - the condition `cond`, which refers only to contract state variables, is guaranteed to hold at every observable contract state.
- `constraint [cond]` - the condition `cond`, which refers to both `\old` and current contract state variables, is guaranteed to hold at every observable contract state except for the initial state after construction (because there is no previous state); constraints are used to restrict how contract state can change over time.

Description of the Analyzed ERC-20 Properties

Properties related to function `approve`

erc20-approve-correct-amount

All non-reverting calls of the form `approve(spender, amount)` that return `true` must correctly update the allowance mapping according to the address `msg.sender` and the values of `spender` and `amount`.

Specification:

```
requires spender != address(0);
ensures \result ==> allowance(msg.sender, \old(spender)) == \old(amount);
```

erc20-approve-false

If function `approve` returns `false` to signal a failure, it must undo all state changes that it incurred before returning to the caller.

Specification:

```
ensures !\result ==> \assigned (\nothing);
```

erc20-approve-never-return-false

The function `approve` must never returns `false`.

Specification:

```
ensures \result;
```

erc20-approve-revert-zero

All calls of the form `approve(spender, amount)` must fail if the address in `spender` is the zero address.

Specification:

```
ensures \old(spender) == address(0) ==> !\result;
```

erc20-approve-succeed-normal

All calls of the form `approve(spender, amount)` must succeed, if

- the address in `spender` is not the zero address and
- the execution does not run out of gas.

Specification:

```
requires spender != address(0);
ensures \result;
reverts_only_when false;
```

Properties related to function `balanceOf`

erc20-balanceof-change-state

Function `balanceOf` must not change any of the contract's state variables.

Specification:

```
assignable \nothing;
```

erc20-balanceof-correct-value

Invocations of `balanceOf(owner)` must return the value that is held in the contract's balance mapping for address `owner`.

Specification:

```
ensures \result == balanceOf(\old(account));
```

erc20-balanceof-succeed-always

Function `balanceOf` must always succeed if it does not run out of gas.

Specification:

```
reverts_only_when false;
```

Properties related to function `allowance`

erc20-allowance-change-state

Function `allowance` must not change any of the contract's state variables.

Specification:

```
assignable \nothing;
```

erc20-allowance-correct-value

Invocations of `allowance(owner, spender)` must return the allowance that address `spender` has over tokens held by address `owner`.

Specification:

```
ensures \result == allowance(\old(owner), \old(spender));
```

erc20-allowance-succeed-always

Function `allowance` must always succeed, assuming that its execution does not run out of gas.

Specification:

```
reverts_only_when false;
```

Properties related to function `totalSupply`

erc20-totalsupply-change-state

The `totalSupply` function in contract ModelToken must not change any state variables.

Specification:

```
assignable \nothing;
```

erc20-totalsupply-correct-value

The `totalSupply` function must return the value that is held in the corresponding state variable of contract ModelToken.

Specification:

```
ensures \result == totalSupply();
```

erc20-totalsupply-succeed-always

The function `totalSupply` must always succeeds, assuming that its execution does not run out of gas.

Specification:

```
reverts_only_when false;
```

Description of the Analyzed Ownable Properties

Properties related to function `renounceOwnership`

ownable-renounce-ownership-is-permanent

The contract must prohibit regaining of ownership once it has been renounced.

Specification:

```
constraint \old(owner()) == address(0) ==> owner() == address(0);
```

ownable-renounceownership-correct

Invocations of `renounceOwnership()` must set ownership to address(0).

Specification:

```
ensures this.owner() == address(0);
```

Properties related to function `owner`

ownable-owner-succeed-normal

Function `owner` must always succeed if it does not run out of gas.

Specification:

```
reverts_only_when false;
```

Properties related to function `transferOwnership`

ownable-transferownership-correct

Invocations of `transferOwnership(newOwner)` must transfer the ownership to the `newOwner`.

Specification:

```
ensures this.owner() == newOwner;
```

DISCLAIMER | CERTIK

This report is subject to the terms and conditions (including without limitation, description of services, confidentiality, disclaimer and limitation of liability) set forth in the Services Agreement, or the scope of services, and terms and conditions provided to you (“Customer” or the “Company”) in connection with the Agreement. This report provided in connection with the Services set forth in the Agreement shall be used by the Company only to the extent permitted under the terms and conditions set forth in the Agreement. This report may not be transmitted, disclosed, referred to or relied upon by any person for any purposes, nor may copies be delivered to any other person other than the Company, without CertiK’s prior written consent in each instance.

This report is not, nor should be considered, an “endorsement” or “disapproval” of any particular project or team. This report is not, nor should be considered, an indication of the economics or value of any “product” or “asset” created by any team or project that contracts CertiK to perform a security assessment. This report does not provide any warranty or guarantee regarding the absolute bug-free nature of the technology analyzed, nor do they provide any indication of the technologies proprietors, business, business model or legal compliance.

This report should not be used in any way to make decisions around investment or involvement with any particular project. This report in no way provides investment advice, nor should be leveraged as investment advice of any sort. This report represents an extensive assessing process intending to help our customers increase the quality of their code while reducing the high level of risk presented by cryptographic tokens and blockchain technology.

Blockchain technology and cryptographic assets present a high level of ongoing risk. CertiK’s position is that each company and individual are responsible for their own due diligence and continuous security. CertiK’s goal is to help reduce the attack vectors and the high level of variance associated with utilizing new and consistently changing technologies, and in no way claims any guarantee of security or functionality of the technology we agree to analyze.

The assessment services provided by CertiK is subject to dependencies and under continuing development. You agree that your access and/or use, including but not limited to any services, reports, and materials, will be at your sole risk on an as-is, where-is, and as-available basis. Cryptographic tokens are emergent technologies and carry with them high levels of technical risk and uncertainty. The assessment reports could include false positives, false negatives, and other unpredictable results. The services may access, and depend upon, multiple layers of third-parties.

ALL SERVICES, THE LABELS, THE ASSESSMENT REPORT, WORK PRODUCT, OR OTHER MATERIALS, OR ANY PRODUCTS OR RESULTS OF THE USE THEREOF ARE PROVIDED “AS IS” AND “AS AVAILABLE” AND WITH ALL FAULTS AND DEFECTS WITHOUT WARRANTY OF ANY KIND. TO THE MAXIMUM EXTENT PERMITTED UNDER APPLICABLE LAW, CERTIK HEREBY DISCLAIMS ALL WARRANTIES, WHETHER EXPRESS, IMPLIED, STATUTORY, OR OTHERWISE WITH RESPECT TO THE SERVICES, ASSESSMENT REPORT, OR OTHER MATERIALS. WITHOUT LIMITING THE FOREGOING, CERTIK SPECIFICALLY DISCLAIMS ALL IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, TITLE AND NON-INFRINGEMENT, AND ALL WARRANTIES ARISING FROM COURSE OF DEALING, USAGE, OR TRADE PRACTICE. WITHOUT LIMITING THE FOREGOING, CERTIK MAKES NO WARRANTY OF ANY KIND THAT THE SERVICES, THE LABELS, THE ASSESSMENT REPORT, WORK PRODUCT, OR OTHER MATERIALS, OR ANY PRODUCTS OR RESULTS OF THE USE THEREOF, WILL MEET CUSTOMER’S OR ANY OTHER PERSON’S REQUIREMENTS, ACHIEVE ANY INTENDED RESULT, BE COMPATIBLE OR WORK WITH ANY SOFTWARE, SYSTEM, OR OTHER SERVICES, OR BE SECURE, ACCURATE, COMPLETE, FREE OF HARMFUL CODE, OR ERROR-FREE. WITHOUT LIMITATION TO THE FOREGOING, CERTIK PROVIDES NO WARRANTY OR

UNDERTAKING, AND MAKES NO REPRESENTATION OF ANY KIND THAT THE SERVICE WILL MEET CUSTOMER'S REQUIREMENTS, ACHIEVE ANY INTENDED RESULTS, BE COMPATIBLE OR WORK WITH ANY OTHER SOFTWARE, APPLICATIONS, SYSTEMS OR SERVICES, OPERATE WITHOUT INTERRUPTION, MEET ANY PERFORMANCE OR RELIABILITY STANDARDS OR BE ERROR FREE OR THAT ANY ERRORS OR DEFECTS CAN OR WILL BE CORRECTED.

WITHOUT LIMITING THE FOREGOING, NEITHER CERTIK NOR ANY OF CERTIK'S AGENTS MAKES ANY REPRESENTATION OR WARRANTY OF ANY KIND, EXPRESS OR IMPLIED AS TO THE ACCURACY, RELIABILITY, OR CURRENCY OF ANY INFORMATION OR CONTENT PROVIDED THROUGH THE SERVICE. CERTIK WILL ASSUME NO LIABILITY OR RESPONSIBILITY FOR (I) ANY ERRORS, MISTAKES, OR INACCURACIES OF CONTENT AND MATERIALS OR FOR ANY LOSS OR DAMAGE OF ANY KIND INCURRED AS A RESULT OF THE USE OF ANY CONTENT, OR (II) ANY PERSONAL INJURY OR PROPERTY DAMAGE, OF ANY NATURE WHATSOEVER, RESULTING FROM CUSTOMER'S ACCESS TO OR USE OF THE SERVICES, ASSESSMENT REPORT, OR OTHER MATERIALS.

ALL THIRD-PARTY MATERIALS ARE PROVIDED "AS IS" AND ANY REPRESENTATION OR WARRANTY OF OR CONCERNING ANY THIRD-PARTY MATERIALS IS STRICTLY BETWEEN CUSTOMER AND THE THIRD-PARTY OWNER OR DISTRIBUTOR OF THE THIRD-PARTY MATERIALS.

THE SERVICES, ASSESSMENT REPORT, AND ANY OTHER MATERIALS HEREUNDER ARE SOLELY PROVIDED TO CUSTOMER AND MAY NOT BE RELIED ON BY ANY OTHER PERSON OR FOR ANY PURPOSE NOT SPECIFICALLY IDENTIFIED IN THIS AGREEMENT, NOR MAY COPIES BE DELIVERED TO, ANY OTHER PERSON WITHOUT CERTIK'S PRIOR WRITTEN CONSENT IN EACH INSTANCE.

NO THIRD PARTY OR ANYONE ACTING ON BEHALF OF ANY THEREOF, SHALL BE A THIRD PARTY OR OTHER BENEFICIARY OF SUCH SERVICES, ASSESSMENT REPORT, AND ANY ACCOMPANYING MATERIALS AND NO SUCH THIRD PARTY SHALL HAVE ANY RIGHTS OF CONTRIBUTION AGAINST CERTIK WITH RESPECT TO SUCH SERVICES, ASSESSMENT REPORT, AND ANY ACCOMPANYING MATERIALS.

THE REPRESENTATIONS AND WARRANTIES OF CERTIK CONTAINED IN THIS AGREEMENT ARE SOLELY FOR THE BENEFIT OF CUSTOMER. ACCORDINGLY, NO THIRD PARTY OR ANYONE ACTING ON BEHALF OF ANY THEREOF, SHALL BE A THIRD PARTY OR OTHER BENEFICIARY OF SUCH REPRESENTATIONS AND WARRANTIES AND NO SUCH THIRD PARTY SHALL HAVE ANY RIGHTS OF CONTRIBUTION AGAINST CERTIK WITH RESPECT TO SUCH REPRESENTATIONS OR WARRANTIES OR ANY MATTER SUBJECT TO OR RESULTING IN INDEMNIFICATION UNDER THIS AGREEMENT OR OTHERWISE.

FOR AVOIDANCE OF DOUBT, THE SERVICES, INCLUDING ANY ASSOCIATED ASSESSMENT REPORTS OR MATERIALS, SHALL NOT BE CONSIDERED OR RELIED UPON AS ANY FORM OF FINANCIAL, TAX, LEGAL, REGULATORY, OR OTHER ADVICE.

Elevating Your Entire Web3 Journey

Founded in 2017 by leading academics in the field of Computer Science from both Yale and Columbia University, CertiK is a leading blockchain security company that serves to verify the security and correctness of smart contracts and blockchain-based protocols. Through the utilization of our world-class technical expertise, alongside our proprietary, innovative tech, we're able to support the success of our clients with best-in-class security, all whilst realizing our overarching vision; provable trust for all throughout all facets of blockchain.