HACKEN

SMART CONTRACT CODE REVIEW AND SECURITY ANALYSIS REPORT

Customer: zkMakers (Liquid Miners)
Date: January 10, 2023

Hacken OÜ
Parda 4, Kesklinn, Tallinn,
10151 Harju Maakond, Eesti,
Kesklinna, Estonia
support@hacken.io

Document

Name	Smart Contract Code Review and Security Analysis Report for zkMakers (Liquid Miners)
Approved By	Noah Jelich | Lead Solidity SC Auditor at Hacken OU
Type	ERC20 token; Staking
Platform	EVM
Language	Solidity
Methodology	Link
Website	https://zkmakers.com
Changelog	22.12.2022 - Initial Review 10.01.2023 - Second Review

Hacken OÜ
Parda 4, Kesklinn, Tallinn,
10151 Harju Maakond, Eesti,
Kesklinna, Estonia
support@hacken.io

Table of contents

Hacken OÜ
Parda 4, Kesklinn, Tallinn,
10151 Harju Maakond, Eesti,
Kesklinna, Estonia
support@hacken.io

Introduction

Hacken OÜ (Consultant) was contracted by zkMakers (Liquid Miners) (Customer) to conduct a Smart Contract Code Review and Security Analysis. This report presents the findings of the security assessment of the Customer's smart contracts.

Scope

The scope of the project is a review and security analysis of smart contracts in the repository:

Initial review scope

Repository	https://gitlab.com/arkerlabs/liquid-miners/smart-contracts
Commit	7a26fc4b249388b61624280e4eddb17f2205fd80
Whitepaper	Link
Functional Requirements	Link
Technical Requirements	Link
Contracts	File: ./contracts/ILMPoolFactory.sol SHA3: 75fd665bcca9321e70376088ac5a247e3229e8090178222546a9f25d80bb89ea File: ./contracts/IProofVerifier.sol SHA3: 85801f1012fcec4165ba6eebef3649850602218061a91ab8b62fa14334b9094c File: ./contracts/LMPool.sol SHA3: d5c2884faf225af9e07ba0e342a5a80a2bdeaa7892e7d4e067302bbb7b6a1a4c File: ./contracts/LMPoolFactory.sol SHA3: f84dc7afc9830bb3f53381259f75f25f934fe5d4303af4a0ca922dc0c76b10ba File: ./contracts/LMXToken.sol SHA3: 503cc0281bc35fcdcd59e24fd592eb6bbba518c1e4600e5f514ffaf48844136d File: ./contracts/Migrations.sol SHA3: dc1cdf247bdfe7c6d67b49b03610547befd29b0886df43f70f5b12274da76a84 File: ./contracts/ProofVerifier.sol SHA3: 65611aa727bf5226eef55d0e739ace670d9556d473b4a5894b5fb0348cea4738 File: ./contracts/TransferHelper.sol SHA3: a6228572f5252c8e7849f9db3880bc585a15097695c2e90d279487f8e8cf9ed9

Second review scope

Repository	https://gitlab.com/arkerlabs/liquid-miners/smart-contracts
Commit	6a90497de885a4c2b335a1308c00d2f0b5d9f1e6
Contracts	File: ./contracts/ILMPoolFactory.sol SHA3: 0f55de55ef52f2cd7d12bfe3f9e02e56257ec0220565060e5753ec1fa43e4a8c File: ./contracts/IProofVerifier.sol SHA3: 664911fc84765ea0420e616b92808356e088b6e06352d77948df951728650982 File: ./contracts/LMPool.sol SHA3: 5ef648de63b0d3d2e189e4df713ff275e5ce7cd2df67c84323079c161e960120 File: ./contracts/LMPoolFactory.sol SHA3: 349a34f6ad5b2b9e5214e66392a5a7731b170127c04a5c574acda80046b9b4af File: ./contracts/LMXToken.sol SHA3: 85e67098a07988e612798ffc6cbf14cc5f5a5c7f8e9eabe785ce0ccc0b6541ac File: ./contracts/Migrations.sol SHA3: 5f3c8cdf642947ca3097f97568da189582c3482bda1138428fd906c178b46026 File: ./contracts/ProofVerifier.sol SHA3: 70570463d8014ed3d59d99cd849b668f7a590c50f2d306d310f80251e8593588 File: ./contracts/TransferHelper.sol SHA3: 93ef15bfd356fe918c01196691f509a919a5addebf3c8d8b7f51c1026419f205

Hacken OÜ
Parda 4, Kesklinn, Tallinn,
10151 Harju Maakond, Eesti,
Kesklinna, Estonia
support@hacken.io

Severity Definitions

Risk Level	Description
Critical	Critical vulnerabilities are usually straightforward to exploit and can lead to the loss of user funds or contract state manipulation by external or internal actors.
High	High vulnerabilities are usually harder to exploit, requiring specific conditions, or have a more limited scope, but can still lead to the loss of user funds or contract state manipulation by external or internal actors.
Medium	Medium vulnerabilities are usually limited to state manipulations but cannot lead to assets loss. Major deviations from best practices are also in this category.
Low	Low vulnerabilities are related to outdated and unused code or minor gas optimization. These issues won't have a significant impact on code execution but affect the code quality

Hacken OÜ
Parda 4, Kesklinn, Tallinn,
10151 Harju Maakond, Eesti,
Kesklinna, Estonia
support@hacken.io

Executive Summary

The score measurement details can be found in the corresponding section of the scoring methodology.

Documentation quality

The total Documentation Quality score is **8** out of **10**.

- Functional requirements are mostly provided. Some details of the tokenomics are left unclarified.
- Technical description is provided.

Code quality

The total Code Quality score is **8** out of **10**.

- The development environment is configured.
- Several template code patterns were found.

Test coverage

Code coverage of the project is **85.87%** (branch coverage).

- Deployment and basic user interactions are covered with tests.
- Negative cases coverage is missing.
- The outcome of some test cases depends on the timestamp, which can sometimes cause the test to fail.

Security score

As a result of the audit, the code contains **1** medium, and **1** low issues. The security score is **9** out of **10**.

All found issues are displayed in the “Findings” section.

Summary

According to the assessment, the Customer's smart contract has the following score: **8.2**

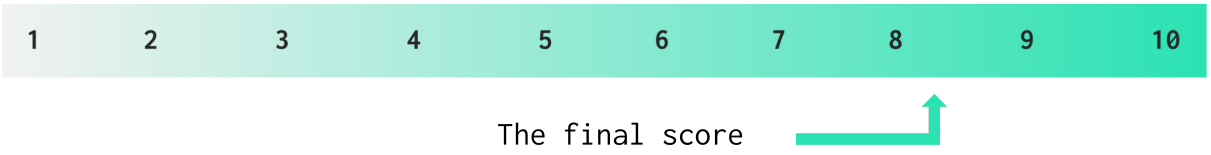

Table. The distribution of issues during the audit

Review date	Low	Medium	High	Critical
15 December 2022	8	7	5	1
10 January 2023	1	1	0	0

Hacken OÜ
Parda 4, Kesklinn, Tallinn,
10151 Harju Maakond, Eesti,
Kesklinna, Estonia
support@hacken.io

Checked Items

We have audited the Customers' smart contracts for commonly known and more specific vulnerabilities. Here are some items considered:

Item	Type	Description	Status
Default Visibility	SWC-100 SWC-108	Functions and state variables visibility should be set explicitly. Visibility levels should be specified consciously.	Passed
Integer Overflow and Underflow	SWC-101	If unchecked math is used, all math operations should be safe from overflows and underflows.	Passed
Outdated Compiler Version	SWC-102	It is recommended to use a recent version of the Solidity compiler.	Passed
Floating Pragma	SWC-103	Contracts should be deployed with the same compiler version and flags that they have been tested thoroughly.	Passed
Unchecked Call Return Value	SWC-104	The return value of a message call should be checked.	Passed
Access Control & Authorization	CWE-284	Ownership takeover should not be possible. All crucial functions should be protected. Users could not affect data that belongs to other users.	Passed
SELFDESTRUCT Instruction	SWC-106	The contract should not be self-destructible while it has funds belonging to users.	Not Relevant
Check-Effect-Interaction	SWC-107	Check-Effect-Interaction pattern should be followed if the code performs ANY external call.	Passed
Assert Violation	SWC-110	Properly functioning code should never reach a failing assert statement.	Passed
Deprecated Solidity Functions	SWC-111	Deprecated built-in functions should never be used.	Passed
Delegatecall to Untrusted Callee	SWC-112	Delegatecalls should only be allowed to trusted addresses.	Not Relevant
DoS (Denial of Service)	SWC-113 SWC-128	Execution of the code should never be blocked by a specific contract state unless required.	Passed
Race Conditions	SWC-114	Race Conditions and Transactions Order Dependency should not be possible.	Passed
Authorization through tx.origin	SWC-115	tx.origin should not be used for authorization.	Passed

Block values as a proxy for time	SWC-116	Block numbers should not be used for time calculations.	Passed
Signature Unique Id	SWC-117 SWC-121 SWC-122 EIP-155 EIP-712	Signed messages should always have a unique id. A transaction hash should not be used as a unique id. Chain identifiers should always be used. All parameters from the signature should be used in signer recovery. EIP-712 should be followed during a signer verification.	Passed
Shadowing State Variable	SWC-119	State variables should not be shadowed.	Passed
Weak Sources of Randomness	SWC-120	Random values should never be generated from Chain Attributes or be predictable.	Not Relevant
Incorrect Inheritance Order	SWC-125	When inheriting multiple contracts, especially if they have identical functions, a developer should carefully specify inheritance in the correct order.	Passed
Calls Only to Trusted Addresses	EEA-Level-2 SWC-126	All external calls should be performed only to trusted addresses.	Passed
Presence of Unused Variables	SWC-131	The code should not contain unused variables if this is not justified by design.	Passed
EIP Standards Violation	EIP	EIP standards should not be violated.	Passed
Assets Integrity	**Custom**	Funds are protected and cannot be withdrawn without proper permissions or be locked on the contract.	Passed
User Balances Manipulation	**Custom**	Contract owners or any other third party should not be able to access funds belonging to users.	Passed
Data Consistency	**Custom**	Smart contract data should be consistent all over the data flow.	Passed
Flashloan Attack	**Custom**	When working with exchange rates, they should be received from a trusted source and not be vulnerable to short-term rate changes that can be achieved by using flash loans. Oracles should be used.	Not Relevant
Token Supply Manipulation	**Custom**	Tokens can be minted only according to rules specified in a whitepaper or any other documentation provided by the Customer.	Passed
Gas Limit and Loops	**Custom**	Transaction execution costs should not depend dramatically on the amount of data stored on the contract. There should not be any cases when execution fails due to the block Gas limit.	Passed
Style Guide Violation	**Custom**	Style guides and best practices should be followed.	Passed

Requirements Compliance	**Custom**	The code should be compliant with the requirements provided by the Customer.	Passed
Environment Consistency	**Custom**	The project should contain a configured development environment with a comprehensive description of how to compile, build and deploy the code.	Passed
Secure Oracles Usage	**Custom**	The code should have the ability to pause specific data feeds that it relies on. This should be done to protect a contract from compromised oracles.	Passed
Tests Coverage	**Custom**	The code should be covered with unit tests. Test coverage should be 100%, with both negative and positive cases covered. Usage of contracts by multiple users should be tested.	Failed
Stable Imports	**Custom**	The code should not reference draft contracts, which may be changed in the future.	Passed

System Overview

zkMakers (Liquid Miners) is a two-part protocol that aims to improve the transparency and traceability of trading activity on cryptocurrency exchanges. The Trading Proof protocol uses ZK-SNARKs to actively verify trading activity on-chain . The Trade-to-Earn Marketplace incentivizes retail traders to generate trading activity on tokens listed on exchanges, creating a transparent and traceable market for volume and liquidity. Token issuers, exchanges, and communities can create reward pools to promote the liquidity and volume of specific token pairs, and traders can earn rewards for trading these pairs and helping projects in need of volume and liquidity:

- *LMXToken* – simple ERC-20 token that mints all initial supply to a deployer. Additional minting is not allowed.
  It has the following attributes:
  - Name: $ZKM (Liquid Miners) Token
  - Symbol: LMX
  - Decimals: 18
  - Total supply: N/A.
- *LMPoolFactory* – a factory contract that allows for the creation of different liquidity mining pools. It features a proof verifier to authenticate the legitimacy of trading activity on exchanges.
- *LMPool* – A contract for a rewards pool for users based on their trading activity.

Privileged roles

- The oracle node role signs messages and submits proofs to the LMPoolFactory contract.
- The owner of the system has the ability to set fees, withdraw tokens, and add and remove blockchains and exchanges. Create and remove oracles, and accept or reject reward tokens. Additionally, the owner has the ability to add and remove other owners from the system.

Recommendations

- Instead of implementing custom ownership in the Migrations.sol contract, use a verified and audited OpenZeppelin implementation.

Risks

- The entirety of the ZKM token (LMX Token) supply is minted to and controlled by the owners.
- In case of a leak of the zero-knowledge oracle API keys, it is possible for fake proofs to be sent to the contract.
- It is possible for users to engage in fraudulent activity, such as making fake trades, in order to earn rewards.

- The **trading reward distribution** system is managed by an **out-of-scope** system and **we cannot verify the validity of its logic**.

Findings

■■■■ Critical

1. EIP Standard Violation

The code does not implement EIP-712 standards and instead uses another hashing and verification mechanism (EIP-191). It may be possible to use the same signature on different blockchains. Therefore, it is more appropriate to use the EIP-712 standard.

Path: ./contracts/ProofVerifier.sol

Recommendation: Implement the EIP-712 standard.

Status: Fixed (Revised commit: 6a90497de885a4c2b335a1308c00d2f0b5d9f1e6)

■■■ High

1. Invalid Calculations

The pendingOracleReward and pendingRebateReward calculations are not sensitive enough. The function calculates the percentage using this formula: *oraclesEpochContribution[_user][epoch] * 100 / oraclesEpochTotalContribution[epoch]*. However, because Solidity does not support decimal points, any calculation results containing a decimal point, such as 10.64 or 55.75, will be rounded to a whole number without the decimal point.

This may lead to calculation errors which may result in imbalances.

Path: ./contracts/LMPool.sol : pendingOracleReward(), pendingRebateReward()

Recommendation: Use *10000* or *tokenDecimal* instead of *100* and adjust the rest of the code accordingly.

Status: Fixed (Revised commit: 6a90497de885a4c2b335a1308c00d2f0b5d9f1e6)

2. Invalid Calculations

The *updatePool* function does not properly update the *accTokenPerShare* variable. This function can be called externally, which allows for potentially malicious behavior. If someone calls this function and provides a random value, the *lastRewardEpoch* value will be updated, but the *accTokenPerShare* value is not be updated. This can lead to incorrect calculations when users try to claim rewards.

This may lead to calculation errors which may result in imbalances.

Path: ./contracts/LMPool.sol : updatePool()

Recommendation: Re-implement the function logic.

Status: Fixed (Revised commit: 6a90497de885a4c2b335a1308c00d2f0b5d9f1e6)

3. **Requirements Violation**

The system deducts fees for promoters, oracles, and custom tokens. These fees are not documented in the provided documents. Thus a user may add a lower than the determined amount of rewards to the pool.

This can lead users to deposit a smaller amount of rewards.

Path: ./contracts/LMPoolFactory

Recommendation: Inform the users about those fees in the public documentation.

Status: Fixed (Revised commit: 6a90497de885a4c2b335a1308c00d2f0b5d9f1e6)

4. **Requirements Violation**

The claim() function, claims the rewards to msg.sender in a given epoch. While doing so, it checks if the rewards can be claimed in the given epoch. This check controls if the current epoch end is greater than the delay + the given epoch. However, it never checks if the reward amount is already claimed for the given epoch. This means that a user can repeatedly calculate and claim rewards from a given epoc.

This can allow users to claim rewards from the same epoch multiple times.

Path: ./contracts/LMPool.sol : claim()

Recommendation: Implement checks to prevent users from claiming rewards multiple times.

Status: Fixed (Revised commit: 6a90497de885a4c2b335a1308c00d2f0b5d9f1e6)

5. **Access Control Violation**

The contract owner can withdraw funds from the contract. This withdrawal can be done anytime without informing the users, leading to sudden balance changes.

This can lead to sudden depletion in the contract.

Path: ./contracts/LMPoolFactory.sol : withdraw()

Recommendation: Remove this functionality or inform users in the public documentation.

Status: Fixed (Revised commit: 6a90497de885a4c2b335a1308c00d2f0b5d9f1e6)

Medium

1. **Best Practice Violation**

   The function does not use the SafeERC20 library to check the result of ERC20 token transfers. Tokens may not follow the ERC20 standard and return a false value in case of transfer failure or not return any value at all.

   Path: ./contracts/LMPoolFactory.sol : withdraw()

   Recommendation: Use the SafeERC20 library to interact with tokens safely.

   Status: Fixed (Revised commit: 6a90497de885a4c2b335a1308c00d2f0b5d9f1e6)

2. **Best Practice Violation**

   The total fees should not exceed the amount input variable. It is possible for the poolFee, promotersRewards, and oracleRewards variables to sum up to a value greater than the amount variable.

   This may lead to contract reverts.

   Path: ./contracts/LMPoolFactory.sol : addRewards()

   Recommendation: Implement necessary checks and set a reasonable limit on the fee amounts.

   Status: Fixed (Revised commit: 6a90497de885a4c2b335a1308c00d2f0b5d9f1e6)

3. **Inconsistent Data**

   While setting the last completed migration timestamp, the provided value can be lower than the currently active value.

   Path: ./smart-contracts-master/contracts/Migrations.sol: setCompleted

   Recommendation: Implement necessary checks to prevent this.

   Status: Reported

4. **Inconsistent Data**

   The events should be emitted in case of state variable updates, but in the functions listed below, event emitting is missing.

   Path: ./smart-contracts-master/contracts/LMPool.sol: addRewards()

   Recommendation: Emit necessary events for the issued functions.

   Status: Fixed (Revised commit: 6a90497de885a4c2b335a1308c00d2f0b5d9f1e6)

5. **Best Practice Violation**

   During proof submission, the related function submitProof() performs external calls and then updates the contract state.

Path: ./smart-contracts-master/contracts/LMPool.sol: submitProof()

Recommendation: Re-implement functions according to the Checks-Effects-Interactions pattern or use ReentrancyGuards.

Status: Fixed (Revised commit: 6a90497de885a4c2b335a1308c00d2f0b5d9f1e6)

6. Inefficient Gas Model

The multiClaim(), multiClaimRebateRewards(), and multiClaimOracleRewards() functions accept user-supplied arrays as inputs and iterate over them.

Path: ./smart-contracts-master/contracts/LMPool.sol: multiClaim(), multiClaimRebateRewards(), multiClaimOracleRewards()

Recommendation: Implement loop limitation.

Status: Fixed (Revised commit: 6a90497de885a4c2b335a1308c00d2f0b5d9f1e6)

7. Contradiction

While creating a dynamic pool the system applies the following check: require(acceptedRewardTokens[_rewardToken] || (_chainId == CONTRACT_DEPLOYED_CHAIN && _rewardToken == _pairTokenA || _rewardToken == _pairTokenB).

This check only applies to the end condition to _rewardToken == _pairTokenA and _chainId == CONTRACT_DEPLOYED_CHAIN.

In order to apply it to both reward token cases, it should follow the following format: require(acceptedRewardTokens[_rewardToken] || (_chainId == CONTRACT_DEPLOYED_CHAIN && (_rewardToken == _pairTokenA || _rewardToken == _pairTokenB))

Path: ./smart-contracts-master/contracts/LMPoolFactory.sol: createDynamicPool()

Recommendation: Modify the require statement accordingly.

Status: Fixed (Revised commit: 6a90497de885a4c2b335a1308c00d2f0b5d9f1e6)

Low

1. Floating Pragma

The project uses floating pragmas ^0.8.0 and >=0.6.0

Paths: ./contracts/LMPoolFactory.sol,

./contracts/ILMPoolFactory.sol,

./contracts/IProofVerifier.sol,

./contracts/LMPool.sol,

./contracts/LMPoolFactory.sol,

./contracts/LMXToken.sol,

./contracts/ProofVerifier.sol,

./contracts/TransferHelper.sol

Recommendation: Consider locking the pragma version whenever possible and avoid using a floating pragma in the final deployment.

Status: Fixed (Revised commit: 6a90497de885a4c2b335a1308c00d2f0b5d9f1e6)

2. **Misleading Error Messages**

"Can't send more than 90 epochs at the same time" messages in require conditions contradict a condition.

This makes the code harder to test and debug.

Path: ./contracts/LMPool.sol: addRewards()

Recommendation: Refactor messages in require conditions to align with code behavior.

Status: Fixed (Revised commit: 6a90497de885a4c2b335a1308c00d2f0b5d9f1e6)

3. **Functions that Can Be Declared External**

In order to save Gas, public functions that are never called in the contract should be declared as external.

This makes the code harder to test and debug.

Path: ./contracts/LMPool.sol: getPromotersEpochTotalContribution(), getPromoterEpochContribution(), getOracleEpochContribution(), getOraclesEpochTotalContribution(), getLastEpoch(), getEpochDuration()

Recommendation: Use the external attribute for functions never called from the contract.

Status: Fixed (Revised commit: 6a90497de885a4c2b335a1308c00d2f0b5d9f1e6)

4. **Redundant Mathematical Operation**

The mathematical operation `*epochDuration + 1*` is redundant.

Path: ./contracts/LMPool.sol: getEpoch()

Recommendation: Remove redundant mathematical operations.

Status: Reported

5. **State Variables Default Visibility**

Variable *precision, lastRewardEpoch* visibility is not specified. Specifying the state variables' visibility helps to catch incorrect assumptions about who can access the variable.

This lowers the contract`s code quality and readability.

Path: ./contracts/LMPool.sol: precision, lastRewardEpoch

Recommendation: Specify variables as public, internal, or private. Explicitly define visibility for all state variables.

Status: Fixed (Revised commit: 6a90497de885a4c2b335a1308c00d2f0b5d9f1e6)

6. Missing Zero Address Validation

Address parameters are being used without checking against the possibility of 0x0.

This can lead to unwanted external calls to 0x0.

Path: ./contracts/LMPoolFactory.sol: submitProof()

Recommendation: Implement zero address checks.

Status: Fixed (Revised commit: 6a90497de885a4c2b335a1308c00d2f0b5d9f1e6)

7. Outdated Compiler Version

Using an outdated compiler version can be problematic, especially if publicly disclosed bugs and issues affect the current compiler version. The project uses a compiler version >=0.6.0.

Path: ./smart-contracts-master/contracts/TransferHelper.sol

Recommendation: Use a contemporary compiler version.

Status: Fixed (Revised commit: 6a90497de885a4c2b335a1308c00d2f0b5d9f1e6)

8. Division by Zero

It is possible to perform zero divisions while calculating the pending amount during proof submissions.

Path: ./smart-contracts-master/contracts/LMPool.sol: submitProof

Recommendation: Implement a check to prevent zero division.

Status: Fixed (Revised commit: 6a90497de885a4c2b335a1308c00d2f0b5d9f1e6)

Disclaimers

Hacken Disclaimer

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

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

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

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

Technical Disclaimer

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