HACKEN

SMART CONTRACT CODE REVIEW AND SECURITY ANALYSIS REPORT

Customer: Neptune Mutual
Date: 23 October, 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 Neptune Mutual
Approved By	Ivan Bondar | Solidity SC Auditor at Hacken OÜ
Tags	Liquidity Pool
Platform	EVM
Language	Solidity
Methodology	Link
Website	https://neptunemutual.com/
Changelog	08.09.2023 - Initial Review 09.10.2023 - Second Review 23.10.2023 - Third 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 Neptune Mutual (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.

System Overview

Neptune Mutual is a liquidity pool protocol with the following contracts:

- *LiquidityGaugePool* – a protocol that manages the depositions to the liquidity pool and distribution of NPM token rewards to users who contribute liquidity to the protocol and engage in staking.
- *LiquidityGaugePoolController* – an abstract contract that is inherited by LiquidityGaugePool to manage pool info registries.
- LiquidityGaugePoolReward – an abstract contract that is used by LiquidityGaugePool to calculate rewards and update voting powers.
- LiquidityGaugePoolState – an abstract contract to be used as storage of variables.

Privileged roles

- The admin of the *LiquidityGaugePool can:*
  - *pause/unpause the contract*
  - *set all the pool info*
- The _NS_ROLES_PAUSER role of *LiquidityGaugePool contract* can:
  - pause the contract
- The _NS_ROLES_RECOVERY_AGENT role of the *LiquidityGaugePool contract can:*
  - *recover/withdraw assets (except staking and reward tokens) from the contract*

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 **9** out of **10**.

- Functional requirements are provided.
- Technical description is provided.
- NatSpec format was missing.

Code quality

- The total Code Quality score is **9** out of **10**.
- Style guide is violated.
- The development environment is configured.

Test coverage

Code coverage of the project is **100%** (branch coverage).

- Deployment and basic user interactions are covered with tests.
- Although the coverage tool result is sufficient, interactions by several users are not tested thoroughly. For instance, withdrawing rewards from different users and comparing the results is missing.

Security score

As a result of the audit, the code contains **no** issues. The security score is **10** 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: **9.7**. The system users should acknowledge all the risks summed up in the risks section of the report.

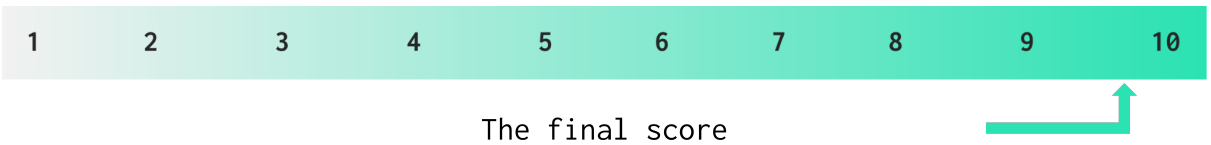

Table. The distribution of issues during the audit

Review date	Low	Medium	High	Critical
8 September 2023	2	3	9	0
9 October 2023	0	0	2	0

23 October 2023	0	0	0	0

Risks

- The LiquidityGaugePool **relies on external contracts for reward calculation**, which were **not part of this audit**. Any vulnerabilities, changes, or unexpected behaviors in these external contracts can directly impact the reward distribution in the LiquidityGaugePool. Users and the platform might face inaccurate or unintended reward distributions due to external dependencies.
- Every user **has to wait at least a minimum 100-block waiting period** after each deposit before they can withdraw funds. Even if users have previously deposited assets, the waiting period is calculated based on the most recent deposit for all, regardless of any prior deposits.

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 specific vulnerabilities. Here are some items considered:

Item	Description	Status	Related Issues
Default Visibility	Functions and state variables visibility should be set explicitly. Visibility levels should be specified consciously.	Passed	
Integer Overflow and Underflow	If unchecked math is used, all math operations should be safe from overflows and underflows.	Passed	
Outdated Compiler Version	It is recommended to use a recent version of the Solidity compiler.	Passed	
Floating Pragma	Contracts should be deployed with the same compiler version and flags that they have been tested thoroughly.	Failed	I01
Unchecked Call Return Value	The return value of a message call should be checked.	Passed	
Access Control & Authorization	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	The contract should not be self-destructible while it has funds belonging to users.	Not Relevant	
Check-Effect-Interaction	Check-Effect-Interaction pattern should be followed if the code performs ANY external call.	Passed	
Assert Violation	Properly functioning code should never reach a failing assert statement.	Passed	
Deprecated Solidity Functions	Deprecated built-in functions should never be used.	Passed	
Delegatecall to Untrusted Callee	Delegatecalls should only be allowed to trusted addresses.	Not Relevant	
DoS (Denial of Service)	Execution of the code should never be blocked by a specific contract state unless required.	Passed	

Race Conditions	Race Conditions and Transactions Order Dependency should not be possible.	Passed	
Authorization through tx.origin	tx.origin should not be used for authorization.	Passed	
Block values as a proxy for time	Block numbers should not be used for time calculations.	Passed	
Signature Unique Id	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.	Not Relevant	
Shadowing State Variable	State variables should not be shadowed.	Passed	
Weak Sources of Randomness	Random values should never be generated from Chain Attributes or be predictable.	Not Relevant	
Incorrect Inheritance Order	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	All external calls should be performed only to trusted addresses.	Passed	
Presence of Unused Variables	The code should not contain unused variables if this is not justified by design.	Passed	
EIP Standards Violation	EIP standards should not be violated.	Passed	
Assets Integrity	Funds are protected and cannot be withdrawn without proper permissions or be locked on the contract.	Passed	
User Balances Manipulation	Contract owners or any other third party should not be able to access funds belonging to users.	Passed	
Data Consistency	Smart contract data should be consistent all over the data flow.	Passed	

Flashloan Attack	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. Contracts shouldn't rely on values that can be changed in the same transaction.	Not Relevant	
Token Supply Manipulation	Tokens can be minted only according to rules specified in a whitepaper or any other documentation provided by the Customer.	Not Relevant	
Gas Limit and Loops	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	Style guides and best practices should be followed.	Failed	I02
Requirements Compliance	The code should be compliant with the requirements provided by the Customer.	Passed	
Environment Consistency	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	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.	Not Relevant	
Tests Coverage	The code should be covered with unit tests. Test coverage should be sufficient, with both negative and positive cases covered. Usage of contracts by multiple users should be tested.	Passed	
Stable Imports	The code should not reference draft contracts, which may be changed in the future.	Passed	

Findings

Critical

No critical severity issues were found.

High

H01. Coarse-grained Access Control; Data Inconsistency

Impact	High
Likelihood	Medium

The *setPool* function in the LiquidityGaugePool contract grants entities with the *DEFAULT_ADMIN_ROLE* extensive power to modify vital parameters of the *_poolInfo* struct, including but not limited to *stakingToken*, *veToken*, and *rewardToken*.

Unrestricted modifications can potentially lead to unauthorized alterations at any moment, posing severe risks like asset freezing, economic imbalances, and unintended reward manipulations.

Furthermore, during the *initialization* of the LiquidityGaugePool.sol contract, the *_setPool* function gets invoked for the first time without adequately checking its input values for invalid or default data. Given the importance of these variables to the contract's functionality, uninitialized or default values might introduce unpredictability and erratic behavior.

Allowing such broad changes without granular controls or proper initial checks can expose the contract to unauthorized alterations, potentially eroding user trust and assets.

Path: ./src/gauge-pool/LiquidityGaugePool.sol: setPool()

Recommendation: Divide the setPool function into smaller, specialized functions to manage specific parameter updates individually. Implement granular access controls for each function based on its importance and potential impact. Introduce thorough checks during the contract's initialization phase, ensuring all vital variables are set to valid, non-default values. Reject or revert transactions that attempt to initialize with inappropriate or default values.

Found in: 5e44aba

Status: Fixed (Revised commit: 5ff90a0)

H02. Unverifiable Logic

Impact	High
Likelihood	Medium

LiquidityGaugePoolReward contract's *_updateVotingPowers* function manages the voting power of each user and it relies on the *VoteEscrowToken*, *VoteEscrowBooster* and *TokenRecovery* contracts. However, they are out-of-scope and their implementation and safety cannot be verified.

GaugeControllerRegistry contract is the *registry* address in the system and is responsible for setting the epochs, but it is out of scope. Therefore, its implementation and safety cannot be verified.

Without a thorough evaluation of these external dependencies, it may be challenging to identify and mitigate potential vulnerabilities, which could in turn pose risks to the overall system's stability and security.

Path: ./src/gauge-pool/LiquidityGaugePoolReward.sol

./src/gauge-pool/LiquidityGaugePool.sol

Recommendation: Add the mentioned contracts to the scope.

Found in: 5e44aba

Status: Mitigated (The Customer accepted all the risks and did not add the mentioned contracts to the scope.) (Revised commit: 5ff90a0)

H03. Denial Of Service; Highly Permissive Owner Access

Impact	**High**
Likelihood	**Medium**

The current implementation of the platform fee lacks constraints, permitting it to be set to any value, including values exceeding 100%. This oversight poses a critical risk as fees exceeding 100% would lead to a scenario where the system locks users' tokens.

This will result in reversion with *PlatformFeeTooHighError* error and cause Denial of Service.

Such lockups occur because the withdraw function will inevitably fail due to insufficient balances, causing the contract to be unable to return more than what users initially deposited.

Additionally, current design of the system allows for altering fee amounts for unclaimed rewards. Unpredictable fee changes can undermine user trust, potentially leading to reduced platform activity or complete withdrawals.

Paths:
./src/gauge-pool/LiquidityGaugePool.sol: setPool()
./src/gauge-pool/LiquidityGaugePool.sol : _withdrawRewards()

Recommendation: Ensure that any earned but unclaimed rewards are shielded from platform fee modifications. Changes in the fee structure should only be applicable to new rewards earned after the

change. Define and set a maximum reasonable limit for the platformFee based on the platform's operational needs and profitability models. For instance, consider capping the fee at a maximum of 20% or another value deemed appropriate, ensuring a fair distribution to users and preventing excessive charges.

Found in: 5e44aba

Status: Fixed (The maximum fee rate is capped to 20%.)(Revised commit: 5ff90a0)

H04. Undocumented Functionality

Impact	**High**
Likelihood	**Medium**

The *DEFAULT_ADMIN_ROLE* can alter vital parameters such as *platformFee*, *veBoostRatio*, *veToken*, *registry* address, *key*, *name*, and *info*. The implications and necessity of these changes are not documented, leading to potential unintended consequences.

Adjustments to these settings can disrupt the normal operations of the platform. Changes, especially to parameters like *veBoostRatio* and *veToken*, can affect the rewards of the users.

The platform fee mechanism, a crucial part of user interactions with the platform, remains unmentioned in the provided documentation.

Path: ./src/gauge-pool/LiquidityGaugePool.sol: setPool()

Recommendation: Remove any settings that are unnecessary for the system. For every remaining admin-controllable parameter, create comprehensive documentation outlining its purpose, the implications of changing it, and the scenarios under which it might need adjustment. Incorporate detailed documentation about the platform fee mechanism, elucidating its purpose, calculation, and implications. A clear understanding of the fee system is essential for users and should be made easily accessible.

Found in: 5e44aba

Status: Mitigated (The client provided reasons for updating certain essential variables, which can be summarized as follows:

- setPool is updated to prevent access during an active epoch.
- platformFee represents fees collected from user rewards, which are subsequently transferred to the treasury.
- veToken refers to an ERC20 token with a lockup mechanism that can increase a user's voting power by up to 4x.
- veBoostRatio is a numerical factor used to adjust the weight of a user's voting power during reward calculations.
- Registry denotes the address authorized to call the setEpoch method on the Liquidity Gauge Pool, enabling the initiation of an epoch on the pool.

Although the platform fee can still be changed, the issue is mitigated because the fee is capped to 20%.) (Revised commit: 5ff90a0)

H05. Missing Storage Gaps

Impact	Medium
Likelihood	High

When working with upgradeable contracts, it is necessary to introduce storage gaps to allow for storage extension during upgrades.

Storage gaps are a convention for reserving storage slots in a base contract, allowing future versions of that contract to use up those slots without affecting the storage layout of child contracts.

Paths:
./src/gauge-pool/LiquidityGaugePoolController.sol
./src/gauge-pool/LiquidityGaugePoolState.sol
./src/gauge-pool/LiquidityGaugePoolReward.sol
./src/util/TokenRecovery.sol
./src/util/WithPausability.sol

Recommendation: Introduce Storage Gaps in the affected contract.

To create a storage gap, declare a fixed-size array in the base contract with an initial number of slots. This can be an array of *uint256* so that each element reserves a 32 byte slot. Use the name *__gap* or a name starting with *__gap_* for the array so that OpenZeppelin Upgrades will recognize the gap.

To help determine the proper storage gap size in the new version of your contract, you can simply attempt an upgrade using *upgradeProxy* or just run the validations with *validateUpgrade* (see docs for Hardhat). If a storage gap is not being reduced properly, you will see an error message indicating the expected size of the storage gap.

Found in: 5e44aba

Status: Fixed (Storage gaps are added to the LiquidityGaugePoolState contract.)) (Revised commit: e90464d)

H06. Highly Permissive Owner Access

Impact	High
Likelihood	Medium

Entities with *DEFAULT_ADMIN_ROLE* permissions can modify crucial variables, such as *stakingToken* and *rewardToken*, even after users have made deposits into the pool.

If the *stakingToken* is changed after deposits, users might become unable to withdraw their original staked assets.

Changing the *rewardToken* might result in users receiving rewards in a token they did not anticipate or desire, potentially altering the economic value of their rewards.

Path: ./src/gauge-pool/LiquidityGaugePool.sol: setPool()

Recommendation: Allow modification of *stakingToken* and *rewardToken* only during the initialization phase of the pool. Once the pool becomes active and starts accepting deposits, lock these parameters to prevent changes or implement a mechanism to check if the pool has active deposits. If there are active deposits, prevent changes to *stakingToken* and *rewardToken*. Implement a validation check to ensure that the *stakingToken* is not equivalent to the *rewardToken*.

Found in: 5e44aba

Status: Fixed (The modification of the parameters are protected and proper checks that controls if the rewards are not distributed from the staked tokens are implemented.) (Revised commit: e90464d)

H07. Mishandled Edge Case

Impact	**High**
Likelihood	**Medium**

The *deposit* function in the contract updates the *_lastDepositHeights* for a user every time they deposit tokens. This mechanism unintentionally extends the withdrawal lockup period for users who make consecutive deposits.

Users who want to top up their deposits or make regular contributions can inadvertently extend their lockup period. This could lead to a scenario where users might not be able to access their funds when needed, especially if they are unaware of this behavior.

Path: ./src/gauge-pool/LiquidityGaugePool.sol: deposit(), withdraw(), exit()

Recommendation: Transit from a single deposit height system to an array-based approach where each deposit gets its unique timestamp. This way, every deposit is treated as a separate entity with its lockup period. Implement a withdrawal function that allows users to specify the index of the deposit they want to withdraw from, ensuring that only deposits that have matured past their lockup period can be withdrawn.

Found in: 5e44aba

Status: Mitigated (The Client has stated their intention to mandate a waiting period of 100 blocks for users immediately following each user deposit.) (Revised commit: 5ff90a0)

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

H08. Highly Permissive Owner Access

Impact	High
Likelihood	Medium

Entities possessing *DEFAULT_ADMIN_ROLE* permissions can change the *lockupPeriodInBlocks*, potentially trapping user funds by extending the withdrawal lockup period unexpectedly.

An extension of *lockupPeriodInBlocks* could lead to users' funds being inaccessible for longer than anticipated, disrupting their financial plans. Users may lose trust in the platform due to unpredictable changes in withdrawal timelines.

Path: ./src/gauge-pool/LiquidityGaugePool.sol: setPool()

Recommendation: Implement a mechanism that ensures that any change to lockupPeriodInBlocks only affects new deposits. Implement a maximum cap on the *lockupPeriodInBlocks* to ensure there are boundaries to how long funds can be locked.

Found in: 5e44aba

Status: Fixed (The *lockupPeriodInBlocks* parameter is now a constant with the value of 100.)(Revised commit: 5ff90a0)

H09. Unrestricted Token Recovery

Impact	High
Likelihood	Medium

The recoverToken function grants entities with the *_NS_ROLES_RECOVERY_AGENT* role the power to withdraw any token from the contract. This includes critical tokens such as staking and reward tokens.

Entities with the *_NS_ROLES_RECOVERY_AGENT* role could potentially misuse this function to siphon off staking or reward tokens. This may result in financial losses for users and erode trust in the platform.

Path: ./src/gauge-pool/LiquidityGaugePool.sol: recoverToken()

Recommendation: Update the *recoverToken* function to exclude vital tokens such as staking and reward tokens. Only allow the function to recover tokens that should not be in the contract (e.g., mistakenly sent tokens).

Found in: 5e44aba

Status: Fixed (Revised commit: 5ff90a0)

Medium

M01. Race Condition

Impact	Medium
Likelihood	Low

The *_withdrawRewards* function calculates the *platformFee* based on the current state of the *_poolInfo.platformFee* variable. Given the dynamic nature of blockchain states, it is possible that the platformFee variable could be changed by an admin between when a user initiates a transaction and when it gets mined, leading to unexpected fee deductions.

A user intending to withdraw their rewards may be subjected to a different platform fee than anticipated. This can result in unexpected deductions, which could erode user trust in the system and lead to potential financial loss for the user.

Path: ./src/gauge-pool/LiquidityGaugePool.sol: withdrawRewards(), exit()

Recommendation: Add an additional parameter to the *withdrawRewards* and exit functions, specifying the expected *platformFee*. This way, the transaction would only succeed if the fee matches the user's expectation at the time of execution. Alternatively, introduce a mechanism that delays platform fee changes, ensuring users are informed in advance and have a grace period to withdraw their rewards before the new fees come into effect.

Found in: 5e44aba

Status: Mitigated (A mechanism has been introduced that restricts any modifications to the pool fee during the duration of an active epoch. This safeguard ensures that users won't face sudden or unexpected fee changes for any transaction within that epoch. However, it's essential for users to note that if rewards are not withdrawn within the same epoch in which they deposited, they might encounter a different fee structure in subsequent epochs. Despite this, the mechanism aids in reinforcing trust by guaranteeing the stability of the fee structure during each epoch.) (Revised commit: 5ff90a0)

M02. Highly Permissive Owner Access

Impact	Medium
Likelihood	Medium

The LiquidityGaugePool contract's pause functionality impacts not only deposits but also withdrawals, including both token withdrawals and reward withdrawals.

Pausing withdrawals, especially in emergencies or uncertain situations, can raise panic and mistrust among users. It denies users

access to their staked assets and earned rewards, potentially causing financial and reputational damage to the platform.

Path: ./src/gauge-pool/LiquidityGaugePool.sol: withdrawRewards(), exit(), withdraw()

Recommendation: While pausing new deposits might be acceptable under certain conditions, users should always have the capability to withdraw their assets and rewards. Ensure that the pause mechanism only impacts deposits, leaving withdrawals unaffected.

Found in: 5e44aba

Status: Fixed (Revised commit: 5ff90a0)

M03. Lack of Emergency Withdrawal Mechanism

Impact	High
Likelihood	Low

The LiquidityGaugePool contract does not feature a mechanism for emergency withdrawals, excluding rewards. Users may be left unable to access their assets if issues arise with reward calculation.

In the event of failures or unforeseen issues with the reward calculation (part of rewards calculation logic is in out-of-scope contracts), users would be unable to withdraw their primary deposited assets. This situation can lead to panic, potential financial losses for users, and erode trust in the platform.

Path: ./src/gauge-pool/LiquidityGaugePool.sol

Recommendation: Implement an *emergencyWithdraw* function. This function should allow users to retrieve their deposited assets without claiming the rewards. This provides a safety mechanism, ensuring users can always access their primary assets irrespective of the platform's state or issues with other dependent contracts. Even with an emergency withdrawal, ensure thorough checks are in place to prevent any misuse of this function.

Found in: 5e44aba

Status: Fixed (Revised commit: 5ff90a0)

Low

L01. CEI Pattern Violation

Impact	Low
Likelihood	Low

In the *deposit* function, a CEI pattern violation has been detected, although it doesn't immediately present a reentrancy risk. The amount

to be deposited is taken after the corresponding state(variables/mappings) is updated. To resolve this issue, it is advisable to refactor the affected code to conform to the CEI pattern, thereby enhancing code readability and alignment with recognized coding standards.

Path: ./src/gauge-pool/LiquidityGaugePool.sol: deposit()

Recommendation: Before changing any internal state, it should be ensured that the user has the necessary tokens and has set the proper allowances.

Found in: 5e44aba

Status: Fixed (Revised commit: 5ff90a0)

L02. Redundant Complexity

Impact	Low
Likelihood	Low

The setEpoch function in the provided code presents both efficiency and design concerns. Specifying epoch numbers manually can lead to irregular epoch numbers and a code that looks complex unnecessarily and this can make it challenging to track and understand the progression of epochs.

Path: ./src/gauge-pool/LiquidityGaugePool.sol: setEpoch()

Recommendation: Use an internal mechanism to increment the epoch number by one instead of manually setting it. This will ensure regular and sequential epoch numbers, making it easier to track the contract's state.

Found in: 5e44aba

Status: Mitigated (The epoch number is validated off-chain. The Customer stated that this implementation needed since the epoch is managed from the registry and a pool can be inactive for some epochs.)(Revised commit: 5ff90a0)

Informational

I01. Floating Pragma

The project uses floating pragmas `^0.8.12`.

This may result in the contracts being deployed using the wrong pragma version, which is different from the one they were tested with. For example, they might be deployed using an outdated pragma version, which may include bugs that affect the system negatively.

Path: ./src/gauge-pool/*.sol

Recommendation: Consider locking the pragma version whenever possible and avoid using a floating pragma in the final deployment. Consider known bugs (https://github.com/ethereum/solidity/releases) for the compiler version that is chosen.

Found in: 5e44aba

Status: Reported (Floating pragmas are present.) (Revised commit: 5ff90a0)

I02. Solidity Style Guides Violation

Contract readability and code quality are influenced significantly by adherence to established style guidelines. In Solidity programming, there exist certain norms for code arrangement and ordering. These guidelines help to maintain a consistent structure across different contracts, libraries, or interfaces, making it easier for developers and auditors to understand and interact with the code.

The suggested order of elements within each contract, library, or interface is as follows:

- Type declarations
- State variables
- Events
- Modifiers
- Functions

Functions should be ordered and grouped by their visibility as follows:

- Constructor
- Receive function (if exists)
- Fallback function (if exists)
- External functions
- Public functions
- Internal functions
- Private functions

Within each grouping, view and pure functions should be placed at the end.

Furthermore, following the Solidity naming convention and adding NatSpec annotations for all functions are strongly recommended. These measures aid in the comprehension of code and enhance overall code quality.

Path: ./src/gauge-pool/LiquidityGaugePool.sol

Recommendation: Consistent adherence to the official Solidity style guide is recommended. This enhances readability and maintainability of the code, facilitating seamless interaction with the contracts. Providing comprehensive NatSpec annotations for functions and

following Solidity's naming conventions further enrich the quality of the code.

Found in: 5e44aba

Status: Reported (Revised commit: 5ff90a0)

I03. Unnecessary Functionality

The LiquidityGaugePool.sol contract inherits from WithPausability.sol which contains a redundant function _setPausers. This excess functionality is not utilized and might increase Gas costs unnecessarily. A more gas-efficient option would be to inherit from PausableUpgradeable.sol.

Every unnecessary function or variable in a smart contract increases the contract deployment and interaction costs due to extra bytecode.

Path: ./src/gauge-pool/LiquidityGaugePool.sol

Recommendation: Replace the inheritance of WithPausability with PausableUpgradeable.sol to remove the extraneous functionality and potentially reduce Gas costs.

Found in: 5e44aba

Status: Fixed (Revised commit: 5ff90a0)

I04. Accumulation of Dust Values

The reward calculation logic in setEpoch could lead to small discrepancies (dust) in token amounts because of Solidity's integer division rounding behavior.

Over time, with many epochs and reward calculations, these discrepancies might accumulate, leading to a non-trivial amount of tokens being "locked" in the contract and not distributed as rewards.

Path: ./src/gauge-pool/LiquidityGaugePool.sol: setEpoch()

Recommendation: Implement a function or mechanism to "collect" and redistribute or handle any accumulated dust in the contract.

Found in: 5e44aba

Status: Reported (Revised commit: 5ff90a0)

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

Disclaimers

Hacken Disclaimer

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

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

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

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

Technical Disclaimer

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

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

Appendix 1. Severity Definitions

When auditing smart contracts Hacken is using a risk-based approach that considers the potential impact of any vulnerabilities and the likelihood of them being exploited. The matrix of impact and likelihood is a commonly used tool in risk management to help assess and prioritize risks.

The impact of a vulnerability refers to the potential harm that could result if it were to be exploited. For smart contracts, this could include the loss of funds or assets, unauthorized access or control, or reputational damage.

The likelihood of a vulnerability being exploited is determined by considering the likelihood of an attack occurring, the level of skill or resources required to exploit the vulnerability, and the presence of any mitigating controls that could reduce the likelihood of exploitation.

Risk Level	High Impact	Medium Impact	Low Impact
High Likelihood	**Critical**	**High**	**Medium**
Medium Likelihood	**High**	**Medium**	**Low**
Low Likelihood	**Medium**	**Low**	**Low**

Risk Levels

Critical: Critical vulnerabilities are usually straightforward to exploit and can lead to the loss of user funds or contract state manipulation.

High: High vulnerabilities are usually harder to exploit, requiring specific conditions, or have a more limited scope, but can still lead to the loss of user funds or contract state manipulation.

Medium: Medium vulnerabilities are usually limited to state manipulations and, in most cases, cannot lead to asset loss. Contradictions and requirements violations. Major deviations from best practices are also in this category.

Low: Major deviations from best practices or major Gas inefficiency. These issues won't have a significant impact on code execution, don’t affect security score but can affect code quality score.

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

Impact Levels

High Impact: Risks that have a high impact are associated with financial losses, reputational damage, or major alterations to contract state. High impact issues typically involve invalid calculations, denial of service, token supply manipulation, and data consistency, but are not limited to those categories.

Medium Impact: Risks that have a medium impact could result in financial losses, reputational damage, or minor contract state manipulation. These risks can also be associated with undocumented behavior or violations of requirements.

Low Impact: Risks that have a low impact cannot lead to financial losses or state manipulation. These risks are typically related to unscalable functionality, contradictions, inconsistent data, or major violations of best practices.

Likelihood Levels

High Likelihood: Risks that have a high likelihood are those that are expected to occur frequently or are very likely to occur. These risks could be the result of known vulnerabilities or weaknesses in the contract, or could be the result of external factors such as attacks or exploits targeting similar contracts.

Medium Likelihood: Risks that have a medium likelihood are those that are possible but not as likely to occur as those in the high likelihood category. These risks could be the result of less severe vulnerabilities or weaknesses in the contract, or could be the result of less targeted attacks or exploits.

Low Likelihood: Risks that have a low likelihood are those that are unlikely to occur, but still possible. These risks could be the result of very specific or complex vulnerabilities or weaknesses in the contract, or could be the result of highly targeted attacks or exploits.

Informational

Informational issues are mostly connected to violations of best practices, typos in code, violations of code style, and dead or redundant code.

Informational issues are not affecting the score, but addressing them will be beneficial for the project.

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

Appendix 2. Scope

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

Initial review scope

Repository	https://github.com/neptune-mutual-blue/periphery/
Commit	5e44aba0427820d3f4c466d8ce1ac703d8d165be
Requirements	Link Link
Technical Requirements	Link Link
Contracts	File: src/gauge-pool/LiquidityGaugePool.sol SHA3: 15bff7f623e65fedc4559a838145e9288fd5d816fe7e152afa4cce87fbb0d364 File: src/gauge-pool/LiquidityGaugePoolController.sol SHA3: 3a5170d62e7cf20bd7169bf3a97eba411fd0df61e213bd47ab56db8c7746dda8 File: src/gauge-pool/LiquidityGaugePoolReward.sol SHA3: a3befa1db28996a6ff0fd58577a1683c257eb28eed3134fe2594dd232b268c44 File: src/gauge-pool/LiquidityGaugePoolState.sol SHA3: 51f75b7d2ff7c7a8806a89ab47b5fc57521ad4c068bcf1852709a7af20d4be67 File: src/gauge-pool/interfaces/ILiquidityGaugePool.sol SHA3: 0ea79859e0d784348e6458924ca75fe5a5f2bf809b6c7b60b62a2948fb0b8436

Second review scope

Repository	https://github.com/neptune-mutual-blue/periphery/
Commit	5ff90a06745f2db81ac3e44da432439810f86fcb
Requirements	Link Link
Technical Requirements	Link Link
Contracts	File: src/gauge-pool/LiquidityGaugePool.sol SHA3: 2a8d07673d2ca67bd51336a041417009b18f49075c7e1e3db8fdecc53c021190 File: src/gauge-pool/LiquidityGaugePoolController.sol SHA3: 46a05dd9e539b56baf8efcc0139b70cfc065a1f013871abc054995e96204cc98 File: src/gauge-pool/LiquidityGaugePoolReward.sol SHA3: f1a63302a0f201cb3a8d0f734072c56287b9c08be1257be3ecd9ae8e429d3e79 File: src/gauge-pool/LiquidityGaugePoolState.sol SHA3: 233b5cf2451e41e5523fac0df43311397636e1e47b456140f7c197f496a6f0c9 File: src/gauge-pool/interfaces/ILiquidityGaugePool.sol SHA3: 275018fb3f0a82c2ef833afed3815d82f310b8919c97dc6e00d7f695e3a970fb

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

Third review scope

Repository	https://github.com/neptune-mutual-blue/periphery/
Commit	e90464d07b425f5f1a85959e1c196a0a8ae43282
Requirements	Link Link
Technical Requirements	Link Link
Contracts	File: LiquidityGaugePool.sol SHA3: 0db3d7a55c6117c5a5eaa2e4453a406a7c4ecbe80112d299d34a103fe3833afb File: LiquidityGaugePoolController.sol SHA3: 46a05dd9e539b56baf8efcc0139b70cfc065a1f013871abc054995e96204cc98 File: LiquidityGaugePoolReward.sol SHA3: f1a63302a0f201cb3a8d0f734072c56287b9c08be1257be3ecd9ae8e429d3e79 File: LiquidityGaugePoolState.sol SHA3: a7c47423059a7329e0979ef9a8e5d84d2c3e785e957b8ee3451877759b2dce04 File: interfaces/ILiquidityGaugePool.sol SHA3: 275018fb3f0a82c2ef833afed3815d82f310b8919c97dc6e00d7f695e3a970fb