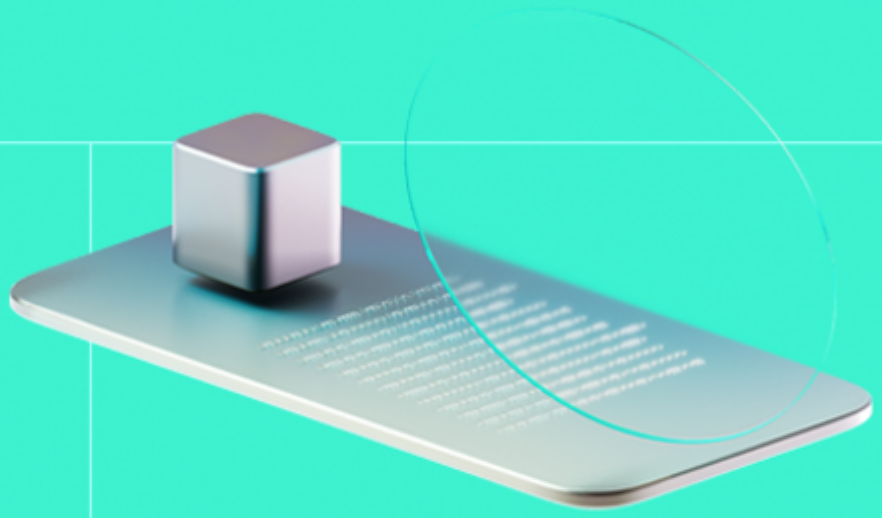
HACKEN

Smart Contract Code Review And Security Analysis Report



Customer: Parallax

Date: 15/04/2024

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

CLMM (Concentrated Liquidity Management Mechanism) is an asset manager that allows users to provide liquidity to Uniswap V3 pools on behalf of Parallax. Parallax performs auto asset management of users' positions on CLMM with compounding and rebalancing functionality.

Platform: EVM

Language: Solidity

Tags: DEX, Vesting

Timeline: 12/02/2024 - 12/04/2024

Methodology: https://hackenio.cc/sc_methodology

Review Scope

Repository	https://bitbucket.ideasoft.io/projects/PAR/repos/clmm
Commit	5f9211e

Audit Summary

10/10	10/10	93%	9/10
Security Score	Code quality score	Test coverage	Documentation quality score

Total 9.7/10

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

2	2	0	0
Total Findings	Resolved	Accepted	Mitigated

Findings by severity

Critical	0
High	1
Medium	0
Low	1

Vulnerability	Status
F-2024-0981 - Missing Validation For The Pool Configuration	Fixed
F-2024-0986 - Missing Refund In The withdrawERC721Token() Function Leads To Stuck Tokens	Fixed

This report may contain confidential information about IT systems and the intellectual property of the Customer, as well as information about potential vulnerabilities and methods of their exploitation.

The report can be disclosed publicly after prior consent by another Party. Any subsequent publication of this report shall be without mandatory consent.

Document

Name	Smart Contract Code Review and Security Analysis Report for Parallax
Audited By	David Camps Novi, Viktor Lavrenenko
Approved By	Przemyslaw Swiatowiec
Website	https://parallaxfinance.org/
Changelog	12/04/2024 - Final Report

Table of Contents

System Overview **6**

Privileged Roles 6

Executive Summary **7**

Documentation Quality 7

Code Quality 7

Test Coverage 7

Security Score 7

Summary 7

Risks **8**

Findings **10**

Vulnerability Details 10

Observation Details 16

Disclaimers 23

Appendix 1. Severity Definitions **24**

Appendix 2. Scope **25**

System Overview

- **CLMMBase.sol** - the first implementation in the list that includes admin methods, depositing, and withdrawal functionality.
- **CLMMCore.sol** - consists of all the storage slots utilized by the protocol to ensure proper storage scheme integrity during upgrades or delegate calls. CLMMCore also extends all the necessary *@openzeppelin* libraries and SwapRouter.
- **CLMMRouter.sol** - provides interactive functions such as depositing, withdrawal, compounding, claiming, rebalancing, and admin methods. CLMMRouter is the main proxy contract of the CLMM protocol and holds all the storage set. The functionality of Router is extended with implementations that do not hold any storage. All the storage is delegated by the router contract.
- **CLMMUtils.sol** - extends CLMMCore and provides functionality that is utilized across all the entities and prevents functionality replication. Each implementation and Router extend CLMMUtils to ensure single scheme integrity.
- **CLMMVault.sol** - the second implementation that includes methods for claiming, compounding, and rebalancing.
- **CLMMVesting.sol** - the third implementation that includes all the functionality for the rewards vesting that could be activated if the user claims rewards in a specified token.
- **SwapRouter.sol** - provides optimal underlying allocation functionality, calculating the actual ratio to deposit into the liquidity pool.
- **UniswapWrapper.sol** - the last implementation that consists of several methods to interact with Uniswap V3 protocol. NOTE: UniswapWrapper does not extend CLMMUtils because it does not utilize any static memory.

Privileged roles

- **Owner:** able to upgrade contracts, add new or to rebalance pools, pause deposits/vesting, set various properties such as compound minimal amounts or implementation addresses. Has also all the rights of User role.
- **User:** able to interact with deposit, withdraw, claim functionality.

Executive Summary

This report presents an in-depth analysis and scoring of the customer's smart contract project. Detailed scoring criteria can be referenced in the scoring methodology.

Documentation quality

The total Documentation Quality score is **9** out of **10**.

- Functional requirements are limited.
  - Business logic is limited. Main ideas are provided but deeper description is missing.
    - How does each flow work, and what are the expected outputs.
    - What is the upgradeability and project architecture.
    - Missing roles description
- The technical description is complete.

Code quality

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

- The development environment is configured.
- Best practices are followed.

Test coverage

Code coverage of the project is **93%** (branch coverage).

- Deployment and basic user interactions are covered with tests.
- Negative cases coverage is provided.

Security score

Upon auditing, the code was found to contain **0** critical, **1** high, **0** medium, and **1** low severity issues, leading to a security score of **10** out of **10**.

All identified issues are detailed in the “Findings” section of this report.

Summary

The comprehensive audit of the customer's smart contract yields an overall score of **9.7.** This score reflects the combined evaluation of documentation, code quality, test coverage, and security aspects of the project.

Risks

- Unconventional ERC20 tokens, such as those with fee-on-transfer or deflationary ones, are not planned for use. However, if they are utilized, it could potentially cause the system's improper functioning.
- The system possesses an upgradeable feature, implying that the owner can upgrade the system by substituting the implementation contract of the CLMMRouter. This introduces the potential risk that subsequent implementations may harbor unforeseen vulnerabilities.
- The system permits the owner to retrieve any trapped ERC20 tokens using the `CLMMCore::rescueERC20Token()` function. However, there are no constraints on the specific ERC20 tokens that can be withdrawn. This presents a risk, as the owner could potentially withdraw the ERC20 reward tokens intended for vesting, which are sent to the `CLMMRouter` contract via `CLMMRouter::fill()` function. This issue is raised in components that haven't been included in the scope and therefore, only shallowly validated.
- The system relies on the secureness of the Owner's private keys, which can impact the execution flow and secureness of the funds. We recommend this account to be at least ⅗ multi-sig.
- The owner can change the pool configuration using `CLMMRouter::setCompoundConfig()` at any time.
- The owner can pause the `CLMMRouter` contract.
- Several interactions of the system are with out-of-scope contracts (see Risk Statement below).

Risk Statement

This audit report focuses exclusively on the security assessment of the contracts within the specified review scope. Interactions with out-of-scope contracts are presumed to be correct and are not examined in this audit. We want to highlight that Interactions with contracts outside the specified scope, such as:

- ./contracts/extensions/UniswapWrapper.sol: IUniswapV3Factory(uniswapV3Pool), ISwapRouter(router), IQuoterV2(quoter).
- ./contracts/core/SwapRouter.sol: CallbackValidation.verifyCallbackCalldata, V3PoolCallee.swap(), V3PoolCallee.wrap(), OptimalSwap.getOptimalSwap().
- ./contracts/core/CLMMVault.sol: positionManager.positions(), positionManager.increaseLiquidity(), positionManager.burn(), positionManager.mint(). IUniswapWrapper(wrapper).getActualRange().
- ./contracts/core/CLMMUtils.sol: positionManager.collect(), positionManager.positions(), positionManager.decreaseLiquidity().
- ./contracts/core/CLMMRouter.sol: positionManager.transferFrom(), positionManager.positions().
- ./contracts/core/CLMMCore.sol: _rescueNativeToken(), _rescueERC20Token().
- ./contracts/core/CLMMBase.sol: positionManager.positions(), positionManager.transferFrom(), positionManager.mint().

have not been verified or assessed as part of this report.

While we have diligently identified and mitigated potential security risks within the defined scope, it is important to note that our assessment is confined to the isolated contracts within this scope. The overall security of the entire system, including external contracts and integrations beyond our audit scope, cannot be guaranteed.

Users and stakeholders are urged to exercise caution when assessing the security of the broader ecosystem and interactions with external contracts. For a comprehensive evaluation of the entire system, additional audits and assessments outside the scope of this report are necessary.

This report serves as a snapshot of the security status of the audited contracts within the specified scope at the time of the audit. We strongly recommend ongoing security evaluations and continuous monitoring to maintain and enhance the overall system's security.

Findings

Vulnerability Details

F-2024-0986 - Missing Refund In The withdrawERC721Token() Function Leads To Stuck Tokens - High

Description:

The functions `CLMMRouter::withdrawERC721Token()` and `CLMMBase::withdrawERC721Token()` enable a user to withdraw their liquidity from the CLMM protocol.

As can be seen from the code snippets below, the function `CLMMBase::withdrawERC721Token()` removes the user's shares, withdraws liquidity from the pool with fees and mints a new position on UniswapV3. However, it is not always the case that the new position will have the token values specified in the `INPM.MintParams`. Uniswap needs to ensure that the pool keeps its normal state after the deposit, hence it recalculates the numbers and might leave the remaining values unused.

The function `positionManager.mint()` is responsible for creating a position with a pair of tokens and minting the LP `ERC721` token to the recipient. However, at the end of the withdrawal, there are some unused tokens left on the balance of the `CLMMRouter` contract, which belong to the holder and should be sent back.

```
function withdrawERC721Token(
Pair memory pair,
uint128 shares,
uint256[4] memory amountsOutMinInner,
uint256[3] memory claimAmountsOutMin
) external onlyValidPool(pair) nonReentrant returns (Amounts memory)
{
// Compound and update user's accumulated rewards on the pool
_updateAccumulatedRewards(pair, msg.sender);
// Claim rewards
Amounts memory claimed = _claim(pair, claimAmountsOutMin, false, true
);
//Repay claimed rewards
if (claimed.amount0 > 0) {
pay(pair.token0, address(this), msg.sender, claimed.amount0);
}
if (claimed.amount1 > 0) {
pay(pair.token1, address(this), msg.sender, claimed.amount1);
}
// Perform delegate call on the base implementation contract to withd
raw ERC721 and transfer to sender
return
abi.decode(
_delegateCall(
base,
abi.encodeWithSelector(
withdrawERC721Selector,
pair,
msg.sender,
shares,
amountsOutMinInner
)
),
(Amounts)
```

```
);
}
```

```
function withdrawERC721Token(
Pair memory pair,
address holder,
uint128 shares,
uint256[4] memory amountsOutMin
) external returns (Amounts memory) {
// Remove shares
(
bytes32 poolId,
INPM.Position memory position,
uint256 liquidity
) = _withdraw(pair, holder, shares);
// Decrease liquidity from pool's ERC721 token
Amounts memory withdrawAmounts = _decreaseLiquidity(
poolInfo[poolId].commonTokenId,
uint128(liquidity),
amountsOutMin[0],
amountsOutMin[1]
);
// Perform swap to the optimal ratio
withdrawAmounts = _optimalSwap(
withdrawAmounts,
poolId,
position,
position.tickLower,
position.tickUpper
);
// Mint ERC721 token
INPM.MintParams memory mintParams = INPM.MintParams({
token0: position.token0,
token1: position.token1,
fee: position.fee,
tickLower: position.tickLower,
tickUpper: position.tickUpper,
amount0Desired: withdrawAmounts.amount0,
amount1Desired: withdrawAmounts.amount1,
amount0Min: amountsOutMin[2],
amount1Min: amountsOutMin[3],
recipient: holder,
deadline: type(uint).max
});
(, , uint256 amount0, uint256 amount1) = positionManager.mint(
mintParams
);
emit Withdraw(
holder,
poolId,
withdrawAmounts.amount0,
withdrawAmounts.amount1,
shares
);
return Amounts({ amount0: amount0, amount1: amount1 });
}
```

The Proof of Concept can be found below.

Assets:

- contracts/core/CLMMBase.sol [https://bitbucket.ideasoft.io/projects/PAR/repos/clmm]
- contracts/core/CLMMRouter.sol [https://bitbucket.ideasoft.io/projects/PAR/repos/clmm]

Status: Fixed

Classification

Severity: High

Impact: Likelihood [1-5]: 5

Impact [1-5]: 3

Exploitability [0-2]: 0

Complexity [0-2]: 2

Final Score: 3.6 (High)

Hacken Calculator Version: 0.6

Recommendations

Remediation: Utilize a refund functionality to transfer the remaining values to the depositor.

Remediation (revised commit: ab15594): a call to `_refund` was added into `CLMMBase::withdrawERC721Token`.

Evidences

Proof Of Concept

Reproduce:

- Alice mints a position on UniswapV3
- Alice deposits her liquidity to the protocol using the position NFT and the function `CLMMRouter::depositERC721Token()`
- Alice withdraws her liquidity in a time via `CLMMRouter::withdrawERC721Token()`
- What can be seen from the calculations is that the balance of the router increased and now stores some remains after the withdrawal process

Results: The developed test:

```
const { users, calm, fee, token0, poolId, service } = await loadFixtu
re(preparing);
const alice = users[0];
let amounts = [1000000000000, 10000000];
const userTokenId = await mintUniPosition(alice.address, amounts[0],
amounts[1], fee);

console.log("Token0 on Router's Balance Before: ", await token0.balan
ceOf(calm.address));
console.log("Token0 on Router's Balance Before: ", await token1.balan
ceOf(calm.address));
await positionManager.connect(alice).approve(calm.address, userTokenI
d);
await calm
.connect(alice)
.depositERC721Token(userTokenId, { token0: token0.address, token1: to
ken1.address }, [0, 0]);
```

```
const shares = (await calm.userInfo(alice.address, poolId)).shares;
const pair = { token0: token0.address, token1: token1.address };
console.log("Token0 on Router's Balance Before Withdrawal: ", await t
oken0.balanceOf(calm.address));
console.log("Token1 on Router's Balance Before Withdrawal:", await to
ken1.balanceOf(calm.address));
await calm.connect(alice).withdrawERC721Token(pair, shares, [0, 0, 0,
0], [0, 0, 0]);
console.log("Token0 on Router's Balance After Withdrawal: ", await to
ken0.balanceOf(calm.address));
console.log("Token1 on Router's Balance After Withdrawal:", await tok
en1.balanceOf(calm.address));
```

Output:

```
Token0 on Router's Balance Before: BigNumber { value: "0" }
Token0 on Router's Balance Before: BigNumber { value: "0" }
Token0 on Router's Balance Before Withdrawal: BigNumber { value: "0"
}
Token1 on Router's Balance Before Withdrawal: BigNumber { value: "0"
}
Token0 on Router's Balance After Withdrawal: BigNumber { value: "2162
5069699" }
Token1 on Router's Balance After Withdrawal: BigNumber { value: "0" }
```

F-2024-0981 - Missing Validation For The Pool Configuration - Low

Description:

Functions `CLMMRouter::addPool()` and `CLMMBase::addPool()` do not validate that values `compoundMins.amount0` `compoundMins.amount1`, and `compoundFee` meet the following condition:

`(compoundMins.amount0 * compoundFee) / PRECISION == 0 || (compoundMins.amount1 * compoundFee) / PRECISION == 0,` which allows the owner to add a pool with any `compoundMins` and `compoundFee.`

Any `compoundMins` and `compoundFee` values will be used in the `CLMMVault::compound()` function. The expected behavior of the `CLMMVault::compound()` function is that the fees that are smaller than `pool.CompoundMins` won't be used and transferred to the `owner`. In reality, with any `compoundMins`, small collected rewards can be compounded.

Assets:

- contracts/core/CLMMBase.sol [https://bitbucket.ideasoft.io/projects/PAR/repos/clmm]

Status: Fixed

Classification

Severity: Low

Impact:

Likelihood [1-5]: 3

Impact [1-5]: 2

Exploitability [0-2]: 2

Complexity [0-2]: 1

Final Score: 1.7 (Low)

Hacken Calculator Version: 0.6

Recommendations

Remediation: Consider following and implementing the same validation mechanism for `compoundMins.amount0`, `compoundMins.amount1` and `compoundFee` as in the `CLMMRouter::setCompoundConfig()`.

Remediation (revised commit: ab15594): the recommended check was added into `CLMMBase::addPool()`.

Observation Details

F-2024-0953 - Missing Storage Gaps - Info

Description:

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.

Note: OpenZeppelin Upgrades checks the correct usage of storage gaps.

Assets:

- contracts/core/CLMMCore.sol [https://bitbucket.ideasoft.io/projects/PAR/repos/clmm]
- contracts/core/CLMMUtils.sol [https://bitbucket.ideasoft.io/projects/PAR/repos/clmm]

Status: Fixed

Recommendations

Remediation:

It is recommended to introduce the storage gaps in the affected contracts.

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 or Truffle). If a storage gap is not being reduced properly, you will see an error message indicating the expected size of the storage gap.

Remediation (revised commit: ab15594): storage gaps were introduced in the base contract `CLMMCore`.

F-2024-0973 - Missing Events Emitting For Critical Functions - Info

Description:

Events allow capturing the changed parameters so that off-chain tools/interfaces can register such changes with timelocks that allow users to evaluate them and consider if they would like to engage/exit based on how they perceive the changes as affecting the trustworthiness of the protocol or profitability of the implemented financial services. The alternative of directly querying the on-chain contract state for such changes is not considered practical for most users/usages.

Missing events do not promote transparency and if such changes immediately affect users' perception of fairness or trustworthiness, they could exit the protocol causing a reduction in liquidity which could negatively impact protocol TVL and reputation.

The following functions that do not emit any events in the contracts:

- `./contracts/core/CLMMRouter::resetSelectors()`
- `./contracts/core/CLMMRouter::setVestingStatus()`
- `./contracts/core/CLMMRouter::setCap()`
- `./contracts/core/CLMMRouter::setBase()`
- `./contracts/core/CLMMRouter::setVault()`
- `./contracts/core/CLMMRouter::setVesting()`
- `./contracts/core/CLMMRouter::resetVesting()`
- `./contracts/core/CLMMRouter::fill()`
- `./contracts/core/CLMMRouter::setWhiteList()`
- `./contracts/core/CLMMRouter::switchPoolStatus()`
- `./contracts/core/CLMMRouter::setCompoundConfig()`
- `./contracts/core/CLMMRouter::depositTokens()`
- `./contracts/core/CLMMRouter::depositERC20Token()`
- `./contracts/core/CLMMRouter::depositERC721Token()`
- `./contracts/core/CLMMRouter::withdrawERC20Token()`
- `./contracts/core/CLMMRouter::withdrawERC721Token()`
- `./contracts/core/CLMMRouter::withdrawTokens()`
- `./contracts/core/CLMMRouter::emergencyWithdraw()`

Assets:

- contracts/core/CLMMBase.sol [https://bitbucket.ideasoft.io/projects/PAR/repos/clmm]
- contracts/core/CLMMRouter.sol [https://bitbucket.ideasoft.io/projects/PAR/repos/clmm]

Status: Fixed

Recommendations

Remediation: Consider emitting the corresponding events in the critical functions.

Remediation (revised commit: ab15594): Missing events have been added to all the functions

F-2024-0977 - Missing Two-Step Transfer Of Ownership Introduces Risks - Info

Description: Ownable2Step and Ownable2StepUpgradeable prevent the contract ownership from mistakenly being transferred to an address that cannot handle it (e.g. due to a typo in the address), by requiring that the recipient of the owner permissions actively accept via a contract call of its own.

Assets:

- contracts/core/CLMMCore.sol [https://bitbucket.ideasoft.io/projects/PAR/repos/clmm]

Status: Fixed

Recommendations

Remediation: Consider using `Ownable2Step` or `Ownable2StepUpgradeable` instead of `Ownable` or `OwnableUpgradeable` from OpenZeppelin Contracts to enhance the security of your contract ownership management. These contracts prevent the accidental transfer of ownership to an address that cannot handle it, such as due to a typo, by requiring the recipient of owner permissions to actively accept ownership via a contract call. This two-step ownership transfer process adds an additional layer of security to your contract's ownership management.

Remediation (revised commit: ab15594): The `OwnableUpgradeable.sol` has been replaced with `Ownable2StepUpgradeable.sol` in the `CLMMCore.sol`

F-2024-0980 - Missing Condition Allows Users To Claim Their Vested Rewards When The Pool is Not Active - Info

Description: Missing `onlyValidPool` condition in the `CLMMRouter::claimVested()` allows users to claim their vested rewards when the pool is not active. Users should be allowed to claim their vested rewards only from active pools.

Assets:

- contracts/core/CLMMRouter.sol [https://bitbucket.ideasoft.io/projects/PAR/repos/clmm]

Status: Fixed

Recommendations

Remediation: It is recommended to add `onlyValidPool` modifier to the `CLMMRouter::claimVested()` function.

Remediation (revised commit: ab15594): the modifier `onlyValidPool` was added into the `CLMMRouter::claimVested()` function as recommended.

F-2024-0982 - Duplicate Code Increases Gas Usage - Info

Description:

The `CLMMRouter::setStatus()` function contains a duplicate piece of code that utilizes additional gas.

The internal functions `_requirePaused()` and `_requireNotPaused()` which have been inherited from the OpenZeppelin `PausableUpgradeable`, are also called inside of the modifiers: `whenPaused()` and `whenNotPaused()` respectively.

These modifiers are already used by the internal functions `_unpause()` and `_pause()`, hence there is no need to utilize them again.

```
function setStatus(bool on) external onlyOwner {
if (on) {
_requirePaused();
_unpause();
} else {
_requireNotPaused();
_pause();
}
}
```

Assets:

- contracts/core/CLMMRouter.sol [https://bitbucket.ideasoft.io/projects/PAR/repos/clmm]

Status: Fixed

Recommendations

Remediation:

Consider removing the unnecessary functions to save additional gas.

Remediation (revised commit: ab15594): the redundant calls were removed from the code.

F-2024-2073 - Missing onERC721Received callback violates best practices - Info

Description: To deal with ERC721 tokens securely, contracts or recipients should implement the IERC721Receiver interface by implementing the `onERC721Received` callback function. The callback is called every time the ERC721 token is transferred via the `safeTransferFrom()`. It is a way of signaling back to the `safeTransferFrom()`, that the recipient understands that it can deal with ERC721 and should implement the necessary functionality to handle such tokens. Otherwise, the NFTs can be sent to the contract which cannot work with ERC721 tokens and eventually stuck. Even though the previously mentioned callback function cannot guarantee the safety of NFTs, it is recommended to be followed.

Assets:

- contracts/core/CLMMRouter.sol [https://bitbucket.ideasoft.io/projects/PAR/repos/clmm]

Status: Fixed

Recommendations

Remediation: It is recommended to implement `onERC721Received` in the `CLMMRouter.sol`

Remediation: (revised commit: fd372f1): The `onERC721Received` callback was implemented in the `CLMMRouter.sol`

Disclaimers

Hacken Disclaimer

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

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

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

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

Technical Disclaimer

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

Appendix 1. Severity Definitions

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

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

hknio/severity-formula

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

Appendix 2. Scope

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

Scope Details

Repository	https://bitbucket.ideasoft.io/projects/PAR/repos/clmm
Commit	5f9211e
Whitepaper	Whitepaper
Requirements	Requirements
Technical Requirements	Technical documentation

Contracts in Scope

contracts/core/CLMMBase.sol

contracts/core/CLMMCore.sol

contracts/core/CLMMRouter.sol

contracts/core/CLMMUtils.sol

contracts/core/CLMMVault.sol

contracts/core/CLMMVesting.sol

contracts/core/SwapRouter.sol

contracts/extensions/UniswapWrapper.sol

contracts/interfaces/ICLMMBase.sol

contracts/interfaces/ICLMMCore.sol

contracts/interfaces/ICLMMMinter.sol

contracts/interfaces/ICLMMRouter.sol

contracts/interfaces/ICLMMUtils.sol

contracts/interfaces/ICLMMVault.sol

contracts/interfaces/ICLMMVesting.sol

contracts/interfaces/IDecimals.sol

contracts/interfaces/INonfungiblePositionManager.sol

contracts/interfaces/ITokensRescuer.sol

Contracts in Scope

contracts/interfaces/IUniswapWrapper.sol

contracts/interfaces/IWETH.sol