BAILSEC.IO

OFFICE@BAILSEC.IO

X: @BAILSECURITY

TG: @HELLOATBAILSEC

FINAL REPORT:

Moebius Finance

May 2024

Disclaimer:

Security assessment projects are time-boxed and often reliant on information that may be provided by a client, its affiliates, or its partners. As a result, the findings documented in this report should not be considered a comprehensive list of security issues, flaws, or defects in the target system or codebase.

The content of this assessment is not an investment. The information provided in this report is for general informational purposes only and is not intended as investment, legal, financial, regulatory, or tax advice. The report is based on a limited review of the materials and documentation provided at the time of the audit, and the audit results may not be complete or identify all possible vulnerabilities or issues. The audit is provided on an "as-is," "where-is," and "as-available" basis, and the use of blockchain technology is subject to unknown risks and flaws.

The audit does not constitute an endorsement of any particular project or team, and we make no warranties, expressed or implied, regarding the accuracy, reliability, completeness, or availability of the report, its content, or any associated services or products. We disclaim all warranties, including the implied warranties of merchantability, fitness for a particular purpose, and non-infringement.

We assume no responsibility for any product or service advertised or offered by a third party through the report, any open-source or third-party software, code, libraries, materials, or information linked to, called by, referenced by, or accessible through the report, its content, and the related services and products. We will not be liable for any loss or damages incurred as a result of the use or reliance on the audit report or the smart contract.

The contract owner is responsible for making their own decisions based on the audit report and should seek additional professional advice if needed. The audit firm or individual assumes no liability for any loss or damages incurred as a result of the use or reliance on the audit report or the smart contract. The contract owner agrees to indemnify and hold harmless the audit firm or individual from any and all claims, damages, expenses, or liabilities arising from the use or reliance on the audit report or the smart contract.

By engaging in a smart contract audit, the contract owner acknowledges and agrees to the terms of this disclaimer.

1. Project Details

Important:
Please ensure that the deployed contract matches the source-code of the last commit hash.

Project	Moebius
Website	moebiusfinance.com
Type	LRT
Language	Solidity
Methods	Manual Analysis
Github repository	https://github.com/Moebius-Finance/contracts-audit/tree/5631ede20f46d69492b00726fc69c3ce0afc6e48
Resolution 1	https://github.com/Moebius-Finance/contracts-audit/tree/10d11ca6482ae78d16c44ad2c656395958f27ed5

2. Detection Overview

Severity	Found	Resolved	Partially Resolved	Acknowledged (no change made)
High	1	1		
Medium	6	4		2
Low	7	3		4
Informational	7	4		3
Governance	4			4
Total	25	12		13

2.1 Detection Definitions

Severity	Description
High	The problem poses a significant threat to the confidentiality of a considerable number of users' sensitive data. It also has the potential to cause severe damage to the client's reputation or result in substantial financial losses for both the client and the affected users.
Medium	While medium level vulnerabilities may not be easy to exploit, they can still have a major impact on the execution of a smart contract. For instance, they may allow public access to critical functions, which could lead to serious consequences.
Low	Poses a very low-level risk to the project or users. Nevertheless the issue should be fixed immediately
Informational	Effects are small and do not post an immediate danger to the project or users
Governance	Governance privileges which can directly result in a loss of funds or other potential undesired behavior

3. Detection

DepositManager

KarakDepositManager/Storage

The KarakDepositManager forms the entry contract for users to stake their tokens while receiving the LRT token as a staking receipt, this is done using a pro-rata approach comparing the USD value of the deposit amount to the overall USD value in the system.

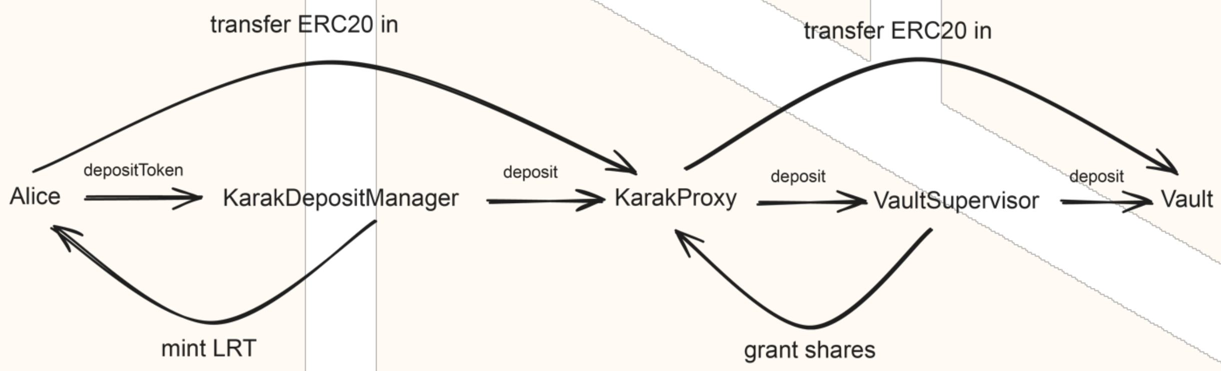
Whenever a deposit happens, the contract chooses from a pool of KarakProxy contracts which serve as intermediary contracts and then stakes into Karak's VaultSupervisor contract. The methodology which determines which proxy contract is used for the next deposit is trivial: Use the one with the lowest USD balance.

There are several configurational settings such as:

a) The possibility of multiple KarakProxy contracts
b) The possibility of enabling up to 31 staking assets
c) The possibility of enabling a minimum deposit amount
d) The possibility of having an upper limit on how much of one token can be staked (overall)

Each enabled asset should have its corresponding Karak vault setted.

The deposit flow can be illustrated as follows:



Appendix: Balance Fetching Mechanism

Upon each deposit, the overall USD balance from the whole system is fetched and then used for pro-rata calculation to determine the shares users will receive for their corresponding token deposit.

It is done as follows:

a) A loop over all proxy contracts is executed
   i) The balance of each proxy for each token (sorted after indexes) is returned
   ii) A loop is being executed over all tokens
      1) **totalTokenDeposits[token]** keeps track of the token amount for one specific token and is aggregated through all proxies. This returns the amount of each token in the system
      2) The USD balance of the proxy is aggregated by converting all tokens in the proxy to the corresponding USD value, the result is stored in **proxiesBalancesInBaseCurreny[proxy]**
      3) This is done for all proxies
   iii) A loop is executed over all tokens
      1) **totalTokenDeposits[token]** is converted to USD, aggregated within each loop and then stored in **_totalValueInBaseCurrency**

Issue_01	Governance Privilege
Severity	**Governance**
Description	Currently, governance of this contract has several privileges for invoking certain functions that can drastically alter the contracts behavior. Trivial Example: The contract owner can swap the vault contract, which then means tokens will reside in the proxy contract (after deposits) and can then be taken out by the proxy owner via calling deposit with a malicious vault as parameter (if previously granted approval). This also means that a malicious owner can mint any amount of LRT by manipulating the vault address and repetitively withdrawing the initial token after LRT has been minted. The governance privilege throughout the architecture is however clear all along because all KarakProxy contracts are upgradeable and trust is expected due to the lack of withdrawal functionality and the possibility to arbitrarily mint LRT.
Recommendations	Consider incorporating a Gnosis Multisignature contract as owner and ensuring that the Gnosis participants are trusted entities.
Comments / Resolution	Acknowledged, the client will incorporate a multisig.

Issue_02	Lack of approval after setVault will result in DoS
Severity	**High**
Description	The setVault function allows to assign a new vault to an already added token. This means that the vault for an asset is “swapped out”. This exposes two problems: a) The old approval is not revoked b) The new approval is not granted This has the effect that all existing proxies will never be able to deposit this token into the new vault, effectively DoS’ing the whole flow. Especially if the vault within the Karak architecture is changed or upon the addToken function the wrong vault is selected, this means that the contract is inherently broken for this specific asset.
Recommendations	Consider looping over all existing proxies and a) revoking the old approval from the old vault b) granting the new approval to the new vault.
Comments / Resolution	Resolved, the setVault function was removed.

Issue_03	First depositor can bypass deposit limit
Severity	**Medium**
Description	Whenever a token is added, the owner can determine a limit, which determines how much of a token can be deposited by all users in total and is enforced as follows: https://github.com/Moebius-Finance/contracts-audit/blob/5631ede20f46d69492b00726fc69c3ce0afc6e48/contracts/KarakDepositManager.sol#L265 The following Poc demonstrates how the first depositor can bypass this limit: a) Contract is newly deployed and the owner creates a pool with wSTEth and the corresponding Karak vault. The Karak vault is at this point still empty. The limit is 10_000e18. b) Charles transfers 10_000e18 tokens manually to the Karak vault c) Charles invokes depositToken with 10_000e18, which triggers the deposit flow. Since this is the first deposit on the Karak vault, the Supervisor will receive 10_000e18 shares which are then allocated to the proxy. However, 10_000e18 shares do represent an amount of 20_000e18 d) Charles will receive 10_000e18 LRT, which now represent 20_000e18 wSTEth, respectively the corresponding USD amount e) Charles successfully circumvented the deposit limit.
Recommendations	Consider ensuring that newly added vaults are never empty.
Comments / Resolution	Acknowledged.

Issue_04	Advanced exploit allows for breaking balance fetching mechanism in case of temporary vault misconfiguration
Severity	**Medium**
Description	The owner can assign a vault to a token via the addToken function and later change the assigned vault via the setVault function, In both scenarios it is not validated that the vault is in fact compatible with the token, which will then result in undesired side-effects. It is notable that this usually does not expose any harm because in such a scenario a deposit would simply revert because the proxy normally does not store any tokens, nor has granted approval to the vault. However, we determined a scenario where this in fact can be abused by a malicious party to break the whole system if the configuration is incorrect: a) Owner adds wSTEth with vault for cbETH (flawed) -> index 0 b) Owner adds cbETH with vault for cbETH -> index 1 -> Proxy is approved to spend cbETH c) Malicious User manually transfers 0.1 ETH of cbETH to the Proxy and invokes depositToken with token = wSTEth and amount = 0.1 ETH -> this will invoke the deposit function on the KarakProxy contract with the incorrect vault (cbETH): https://github.com/Moebius-Finance/contracts-audit/blob/5631ede20f46d69492b00726fc69c3ce0afc6e48/contracts/KarakDepositManager.sol#L294

-> this will set the tokenMapping on the proxy with wSTEth as first index:

https://github.com/Moebius-Finance/contracts-audit/blob/5631ede20f46d69492b00726fc69c3ce0afc6e48/contracts/KarakProxy.sol#L66

-> this will call deposit on the Supervisor with the cbETH vault (due to the misconfiguration):

https://github.com/Moebius-Finance/contracts-audit/blob/5631ede20f46d69492b00726fc69c3ce0afc6e48/contracts/KarakProxy.sol#L67

-> due to the fact that the provided vault is the cbETH vault, this will push the cbETH vault as the first index in the stakersVault[proxyAddr] array:

https://github.com/Andalusia-Labs/karak-restaking/blob/main/src/VaultSupervisor.sol#L290

Note that this exploit is only possible due to step b) which has approved the vault to spend the cbETH from the proxy contract.

Summarized, this will result in:

a) tokenMapping containing wSTEth as first index
b) stakersVault[proxyAddr] containing cbETH as first index

This means upon the _getSystemBalances function and the further balance fetching flow, the returned sequence of the proxy's deposit will return cbETH as first index, which is then assigned to the first index in the tokenMapping, which is wSTEth.

The balance fetching mechanism is therefore permanently flawed.

Recommendations	Consider ensuring that the vault is compatible with the token: https://github.com/Vectorized/solady/blob/main/src/tokens/ERC4626.sol#L71 require(Vault.asset() == token, “No match”);
Comments / Resolution	Resolved, setVault is removed. Furthermore the team indicated that the required check will be added within the next iteration.

Issue_05	Lack of upper limit for amount of proxies can result in DoS
Severity	**Medium**
Description	Currently, there is no upper limit on how many proxies can be added to the system. This means the owner can arbitrarily often invoke the addProxy function to add a new proxy to the system. Within the next deposit, this new proxy is then presumably the one which is used for the next deposit, as the _selectProxy function always fetches the proxy with the lowest USD balance. Summarized, this means that all proxies in the system will be used for a deposit, sooner or later. In reverse this means that upon the _getSystemBalances a loop over all existing proxies is executed to fetch the overall USD balance. This step is mandatory for any depositToken call. This now means if there are too many proxies, the depositToken function will run out of gas and no further deposits are allowed but at the same time it is not possible to remove a proxy because all proxies will have a deposit allocation, therefore not being removable:

	https://github.com/Moebius-Finance/contracts-audit/blob/5631ede20f46d69492b00726fc69c3ce0afc6e48/contracts/KarakDepositManager.sol#L160
Recommendations	Consider setting a reasonable hardcoded limit on how many proxies can be added to the system.
Comments / Resolution	Resolved.

Issue_06	Vault change will result in temporarily inaccessible shares
Severity	**Low**
Description	The VaultSupervisor allocates shares not based on the provided token, but on the "to deposited" vault: https://github.com/Andalusia-Labs/karak-restaking/blob/main/src/VaultSupervisor.sol#L294 If now the vault for a token is changed via the setVault function, these shares are no longer accessible (until it is set back to the correct vault) This issue is corresponding to the future implementation where withdrawals are executed via redeemShares.
Recommendations	Consider ensuring that the vault address remains constant to where tokens were deposited in.
Comments / Resolution	Resolved.

Issue_07	Rebase mechanism of underlying value can be used to gain advantage during deposit
Severity	**Low**
Description	Whenever a user deposits tokens, the amount of LRT to be minted is smoothly calculated using a pro-rata approach with the to be deposited USD value and the total underlying USD value. Due to the fact that there will be a basket of LRTs staked, their rebase epochs may differ. For example a user can stake LRT 1 which rebases every monday while there is LRT 2 staked which rebases every friday. This difference can then be abused by the user to frontrun the rebase mechanism by depositing 1 block before LRT 2 rebase, which then immediately increases the value of the user's LRT receipt token. This issue is only marked as low because there is no real way to prevent it and it can not currently be exploited because withdrawals aren't enabled. If there would be withdrawals without wait time enabled, this could be abused to flash-theft rebase yield from the vault.
Recommendations	Consider being careful when adding withdrawals to make sure that this behaviour can't be exploited by malicious users.
Comments / Resolution	Resolved, there will always be a delay.

Issue_08	Proxy removal does not work in the current implementation
Severity	**Informational**
Description	Proxies can be only removed if they have a zero balance for all tokens. This is in the current state impossible to achieve because withdrawals are not enabled. Therefore, no proxies can be removed. **For the future:** Additionally, users can simply grief this call by depositing into the to be removed proxy. We recommend for the future to implement functionality that prevents depositing from said vaults before a removal attempt
Recommendations	Consider implementing necessary validation for the future upgrade to ensure this griefing vector is mitigated
Comments / Resolution	Acknowledged, this will be addressed in the next iteration.

Issue_09	Contract is not compatible with transfer-tax tokens
Severity	**Informational**
Description	This contract is not compatible with transfer-tax tokens. If these token types are used for any purpose within the contract, this will result in down-stream issues and inherently break the accounting.
Recommendations	Consider not using these tokens.
Comments / Resolution	Acknowledged, these tokens will never be used.

Token

LRT

The LRT contract serves as a receipt token for deposits in the Moebius protocol. It represents a basket of underlying LSTs which are deposited into Karak and steadily increases in value due to fee accruals.

Issue_10	Governance Privilege: Key address can arbitrarily mint and burn
Severity	**Governance**
Description	The MoebiusCNC contract handles to which address the depositManagerKey is assigned. This address can mint and burn LRT tokens and is meant to be the KarakDepositManager but can also be any other address. In any scenario it is possible for a total loss of funds if this role is assigned to the wrong address.
Recommendations	Consider incorporating a Gnosis Multisignature contract as owner and ensuring that the Gnosis participants are trusted entities.
Comments / Resolution	Acknowledged, the client will incorporate a multisig.

Proxy

KarakProxy/Storage

The KarakProxy contract is the intermediate contract between the Moebius Protocol and the Karak Protocol. It directly interacts with Karak’s VaultSupervisor via the deposit function and is the recipient of the corresponding shares. In the current implementation, withdrawals are not possible but will be implemented in the future.

The contract furthermore implements logic to sort amounts of deposited assets in the Karak protocol in such a manner that it fits into the supportedTokens sequence:

Upon each deposit, the moebiusIndex of the deposit token is pushed into the tokenMapping, which then represents the sequence which is stored in the VaultSupervisor contract. Upon balance fetching, a loop is executed over all supported tokens and eventual deposited balances will be assigned to the correct moebiusIndex.

Issue_11	Governance Privilege
Severity	**Governance**
Description	All proxies which are deployed can be upgraded at once by changing the implementation. This can result in a full loss of funds if governance keys are compromised.
Recommendations	Consider incorporating a Gnosis Multisignature contract as owner and ensuring that the Gnosis participants are trusted entities.
Comments / Resolution	Acknowledged, the client will incorporate a multisig.

Issue_12	Advanced exploit allows attacker to drain majority of the system value under rare circumstances
Severity	**Medium**
Description	First of all, we need to acknowledge that the Karak Vault implementation is potentially vulnerable to the share inflation attack. **This is highly dependent on the configuration of the ERC4626 vault (offset/useVirtualShares), which is out of scope, but it is technically possible.** In itself, this is usually no big deal because Karaks VaultSupervisor incorporates a minSharesOut parameter that exactly **prevents this scenario**: https://github.com/Andalusia-Labs/karak-restaking/blob/v1.1/src/VaultSupervisor.sol#L59 This parameter however remains unused by the Moebius protocol, simply because it is hard to incorporate in the deposit flow. It would essentially need to be externally calculated by the user that initiates the deposit. If it would be calculated by the frontend solely, it would still use the (potentially) manipulated exchange ratio. Therefore, it seems simply not feasible to implement. Now we need to understand that the share inflation attack is economically only successful if users subsequently deposit after the exchange ratio is manipulated. While, as mentioned, it is theoretically possible that users will execute subsequent deposits in a flawed vault, it is highly unlikely given the overall deposit flow. This is what makes this exploit so sophisticated: It is not necessary for victims to deposit, the attacker can just abuse the design choice. Our advanced crafted exploit scenario allows an exploiter to steal "shares" from the system using the share inflation attack as root-cause.

How is that possible?

a) The whole system is running for some time and everything works perfectly. Karak suddenly deploys a new vault with a corresponding token, let's say wSTEth. This is detected by the Moebius owner and the same vault with the token is added to the system with a large limit (the higher the limit, the higher the damage).

b) Attack unfolds:

First: Malicious user transfers a 100e18-1 wSTEth directly to the Karak vault and then deposits 1 wei in the VaultSupervisor. This step is done outside of the Moebius System, by directly interacting with Karak.
This will manipulate the exchange ratio in such a way that the share value for future deposits does not necessarily become zero but rounds down from 2 WEI to 1 WEI.

Example:

assets * totalSupply / totalAssets

99e18 * 1 / 100e18 = 1.99 WEI -> 1 WEI

This exchange ratio is specifically important because the VaultSupervisor won't allow for receiving zero shares:

https://github.com/Andalusia-Labs/karak-restaking/blob/main/src/VaultSupervisor.sol#L284

The crux: Usually, users would need to fall for this manipulation and at least after a few deposits, this exploit would have been uncovered.

	Second: The attacker can now deposit via the KarakDepositManager, which grants the attacker a pro-rata share of his deposit value based on the overall system value (Independent of how much the system actually receives as shares). The problem: The Karak system will *not* receive the shares that are expected because of the manipulated exchange ratio, the Karak system will essentially receive the rounded down amount of shares. Following our example, this means that the attacker can deposit 99e18 wSTEth, which will mint him the corresponding amount of LRT. However, due to the flaw in the exchange ratio, the Moebius system will effectively receive shares worth 49.5e18 wSTEth and the attacker will receive 49.5e18 wSTEth on his external vault position. The attacker can repeat this exploit until he owns the majority of shares with only 50% of the deposit amount. Due to the fact that this is dependent on the ERC4626 configuration, we have marked this issue only as **medium severity**. However, we are still of the opinion that it is mandatory to follow our recommendation.
Recommendations	Consider ensuring to only add vaults which have a valid exchange ratio and are not empty. This can be done with a simple check during the addToken and setVault functions within the KarakDepositManager: require(Vault.totalSupply() >= 1e10, “Vault can still be manipulated”) (This check must be adjusted depending on the underlying token’s decimals) Additionally to this check, it is important to manually confirm the exchange ratio correctness before adding the vault.
Comments / Resolution	Resolved, that check has been implemented in the KarakDepositManager.

Issue_13	Usage of non-upgradeable dependencies can result in future storage collision
Severity	**Low**
Description	The KarakProxy contract inherits the KarakProxy storage and MoebiusBase contract. Both contracts do not expose a gap[] variable which can lead to storage collisions in future upgrades. Read more here: https://bailsec.io/tpost/8cts0tk4g1-why-you-need-to-be-careful-when-upgradin
Recommendations	Consider implementing a separate MoebiusBaseUpgradeable contract and incorporate a storage gap[] into the KarakProxy.
Comments / Resolution	Resolved, the MoebiusBase contract is supposed to be stateless. A comment has been added.

Issue_14	The mapped token index might be wrong after withdrawals are enabled
Severity	**Informational**
Description	The Karak proxy stores a tokenMapping to allow for easier and cheaper calculation of the total value locked in the different proxies. When withdrawals are allowed, Karak might return the tokens in a different order than before, due to the tokenMapping variable not being adjusted whenever a token is completely withdrawn. This could lead to severe issues as the total value locked would be wrong. This issue is only raised as informational as the withdrawals are not yet added.

Recommendations	The issue is pretty complicated to really fix. However, there is an easy fix: Never withdraw the full amount of an asset and always keep at least 1 share in the Karak vault. That way, the array returned will never be reordered.
Comments / Resolution	Acknowledged, the team will take this into consideration for the next iteration.

ProxyFactory

The ProxyFactory contract is a simple factory contract for Beacon proxies. The owner can deploy a new BeaconProxy which then sets the ProxyFactory as the beacon and fetches the corresponding implementation pointer from it.

This allows for upgrading all proxies at once by calling the setImplementation function. It is particularly useful in the context of future withdrawals because this removes the necessity of upgrading each KarakProxy individually.

No issues found.

Oracle

PriceProvider

The PriceProvider contract is a multi-configurational oracle that allows for fetching asset prices in USD or ETH using a valid Chainlink feed. It imposes various different validations such as:

a) Staleness check
b) Deviation check
c) AggregatorRoundId check

To facilitate these checks, historical data is stored in the form of the latest valid oracle response and is then potentially used if new responses are invalid, as long as the historic response is not considered as stale.

Additionally, it facilitates fetching assets without a direct oracle, using their exchange rate, such as ERC4646 tokens or LRT/LST tokens, by executing a static call to the corresponding contract, fetching the exchange rate and applying it on top of the CL oracle.

Issue_15	Governance Issue: Privileged oracle setting
Severity	**Governance**
Description	Currently, governance of this contract has several privileges for invoking certain functions that can drastically alter the contracts behavior. For instance, any oracle can be assigned to any token, which can significantly alter the balance fetching mechanism.
Recommendations	Consider incorporating a Gnosis Multisignature contract as owner and ensuring that the Gnosis participants are trusted entities.
Comments / Resolution	Acknowledged, the client will incorporate a multisig.

Issue_16	setOracle call may keep incorrect historical price
Severity	**Medium**
Description	The setOracle function has two distinct purposes: a) Set an initial oracle b) Set a subsequent oracle The second purpose may be applied if the current active oracle is flawed, which then means it needs to be updated to prevent any (further) damage.

	In that scenario, the setOracle function with the new parameters is called, with the intention to immediately change the chainLinkOracle address. There is however an edge-case, in which the price is not updated but rather the old price is kept. Consider the following scenario: a) The old chainLinkAddress is compromised and returns an incorrect price, the latest roundId is stored as 110680464442257326739 b) The contract owner immediately recognizes this flaw and calls setOracle with an updated chainLinkAddress c) Everything is set up and the _processFeedResponses function is called d) The new oracle has the same or lower roundId as the latest one stored in the PriceRecord, therefore the _processFeedResponses function will immediately switch in the else clause: https://github.com/Moebius-Finance/contracts-audit/blob/5631ede20f46d69492b00726fc69c3ce0afc6e48/contracts/PriceProvider.sol#L328 e) The old price is returned and stored, even with the updated oracle now, all executions in the same block will now still use the flawed price.
Recommendations	Consider incorporating this edge-case into the _processFeedResponses function via a 6th parameter (isSetOracleCall) and if this is true, a separate call-flow should be invoked that executes the following branch even if the roundIds are the same:

	https://github.com/Moebius-Finance/contracts-audit/blob/5631ede20f46d69492b00726fc69c3ce0afc6e48/contracts/PriceProvider.sol#L306
Comments / Resolution	Acknowledged, this is accepted risk

Issue_17	Deviation logic incorrectly uses currentPrice as deviation base
Severity	**Medium**
Description	As per comment, and also variable naming: MAX_PRICE_DEVIATION_FROM_PREVIOUS_ROUND “ // Maximum acceptable price deviation between two consecutive Chainlink oracle prices (50%).” There should not be a deviation > 50% between two rounds. The calculation incorrectly uses the currentPrice as deviationBase, which will result in a flawed deviation calculation. Consider the following PoC 1: a) previousPrice was 10 and increased by 50% to currentPrice which is now 15 b) delta = 5 c) 5e18 * 1e18 / 15e18 = 33e17 This is incorrect, as it should be 50e17

	Therefore, the price deviation can be higher than anticipated. Consider the following PoC 2: a) previous price was 10 and decreased by 50% to currentPrice which is now 5 b) delta = 5 c) 5e18 * 1e18 / 5e18 = 1e18 This is incorrect as it should be 50e17 Therefore, the price deviation can be higher than anticipated.
Recommendations	Consider using prevScaledPrice as divisor
Comments / Resolution	Resolved. The client will keep the current price as base for the change determination, as per client, this was always the initial intention.

Issue_18	Typecasting of timestamp to uint32 restricts future contract functionality
Severity	**Low**
Description	Currently, the block.timestamp is casted down to uint32. This will work for the next 82 years but after that it will overflow, rendering the contract unusable.
Recommendations	Consider using a higher uint type for that purpose.
Comments / Resolution	Acknowledged.

Issue_19	Invalid new oracle setting incorrectly extends “non-staleness” period of historical data
Severity	**Low**
Description	Whenever the _setOracle function is invoked a second time, this means that the current oracle record is overridden and points to a different chainlink oracle. In that scenario, it is desired that the new price is reflected properly. If however the previous response is invalid, this means that the historical price is used for the PriceRecord update: https://github.com/Moebius-Finance/contracts-audit/blob/5631ede20f46d69492b00726fc69c3ce0afc6e48/contracts/PriceProvider.sol#L312 which is in itself no problem (if the update is not an emergency update as described in the issue above). A small inconsistency however occurs due to the fact that the historical price is used together with the timestamp of the current response: https://github.com/Moebius-Finance/contracts-audit/blob/5631ede20f46d69492b00726fc69c3ce0afc6e48/contracts/PriceProvider.sol#L143 If for example the historical price was reported at TS 10_000 and the new timestamp is now set to 80_000, this means that the staleness check is manipulated. If the next subsequent fetchPrice call also reports invalid data, a staleness check on the PriceRecord is executed, which will then return true while it in fact may have already reached the threshold.
Recommendations	Due to the fact that the impact is negligible, we do not recommend any further changes as this would alter the function flow itself and will

	also have an impact on the whole context. Potential changes may be subject to re-audit.
Comments / Resolution	Acknowledged.

Issue_20	Lack of heartbeat validation can result in incorrect staleness mode
Severity	**Low**
Description	Chainlink oracles inherently use a heartbeat of 3600 seconds. This means if a price has not reached the deviation threshold, it is latest updated after 3600 seconds. The setOracle function however allows for a heartbeat below 3600 seconds, in fact, there is no lower validation at all. This means if for example a heartbeat of 1800 seconds is used, this will mark an oracle as stale while it is not necessarily stale in reality, just not updated from chainlink yet. Since we are of the opinion that the Chainlink configuration can be seen as best-practice, it might make sense to follow this standard to avoid unnecessarily wrong staleness situations.
Recommendations	Consider ensuring that _heartbeat is >= 3600 seconds.
Comments / Resolution	Acknowledged.

Issue_21	Lack of staleness and historic data check if aggregatorRoundId is 1
Severity	**Low**
Description	The PriceProvider contract implements a lot of validations on the current response, most importantly: a) staleness check b) deviation check (0.5%) This is handled within the _processFeedResponses function whenever the aggregatorRoundId is > 1. If however the aggregatorRoundId is 1, this means that a new aggregator has been set, which will then increase the roundId. In that scenario, there is no staleness nor deviation check.
Recommendations	The root of this issue lies within the complexity of calculating the latest aggregatorRoundId for the previous roundId. The perfect solution would be to calculate this aggregatorRoundId and apply it to the deviation-check. However, we do not recommend this change as it makes the codebase unnecessarily complex. A potential addition should however be the staleness check for the currentResponse.
Comments / Resolution	Acknowledged.

Issue_22	ETH_ADDRESS condition within _fetchPrice may be incorrect if initial oracle is changed
Severity	**Informational**
Description	Upon deployment, the ETH_ADDRESS oracle is set with isUSDIndexed = true. This can be changed later and in such a scenario the following line will be incorrect: https://github.com/Moebius-Finance/contracts-audit/blob/5631ede20f46d69492b00726fc69c3ce0afc6e48/contracts/PriceProvider.sol#L203
Recommendations	Consider not changing the ETH_ADDRESS oracle.
Comments / Resolution	Resolved.

Base

MoebiusBase

The MoebiusBase contract is a simple dependency which is inherited by the following contracts:

a) PriceProvider
b) KarakDepositManager
c) KarakProxy
d) ProxyFactory

It exposes important modifiers and stores the linkage to the MoebiusCNC contract which then handles address and role connections.

No Issues found

Router

LoadBalancingRouter

The LoadBalancingRouter is a simple dependency that receives an array of proxies with their corresponding balances and fetches the proxy with the highest/lowest balance. This is useful for deposit/withdrawal purposes to deposit to the proxy with the lowest balance and withdraw from the proxy with the highest balance

Issue_23	Gas optimisations
Severity	**Informational**
Description	The calculateIngressRoute and calculateEgressRoute functions take the proxiesBalances array and cache it in memory. This practice unnecessarily consumes gas and can be improved.
Recommendations	Consider marking this array as calldata to save some gas, as solidity will copy the entire array to memory which is not useful here.
Comments / Resolution	Resolved.

Issue_24	Typographical errors
Severity	**Informational**
Description	The contract contains one or more typographical issues, in an effort to keep the report size reasonable, we will enumerate these issues below: L 17 * @notice Calculates the ingress route for a deposit by selecting the proxy

	The comment contains two spaces between “the” and “proxy”, consider removing one for better code quality. The calculateIngressRoute and calculateEgressRoute functions will revert with an out-of-bounds error if both arrays have a length of 0. To increase clarity, a custom error could be added.
Recommendations	Consider fixing the aforementioned typographical errors.
Comments / Resolution	Resolved.

Governance

MoebiusCNC

The MoebiusCNC contract is a simple governance and control contract which allows for pausing the system and managing addresses and roles, similar to a registry contract.
The contract owner can assign a specific key or role to an address which is then used to validate privileges throughout the architecture.

Issue_25	Lack of length validation for setAddresses function
Severity	**Informational**
Description	The setAddresses function allows to assign addresses to keys. This assignment is then used throughout the architecture, such as the depositManagerKey, which is assigned to the KarakDepositManager contract. The provided arrays are however not validated to be the same lengths.

Recommendations	Consider validating the array lengths for keys and values.
Comments / Resolution	Resolved.

Libraries

Addresses

The Addresses library is a trivial library which keeps track of the bytes32 value for the corresponding addresses. It is currently used in the following contracts:

a) KarakDepositManager: PRICE_PROVIDER, ROUTER
b) KarakProxy: VAULT_SUPERVISOR

No issues found.

Roles

The Addresses library is a trivial library which keeps track of the bytes32 value for the corresponding addresses. It is currently used in the following contract(s):

a) MoebiusCnc

No issues found.