QuillAudits

AUDIT REPORT

March, 2025

For

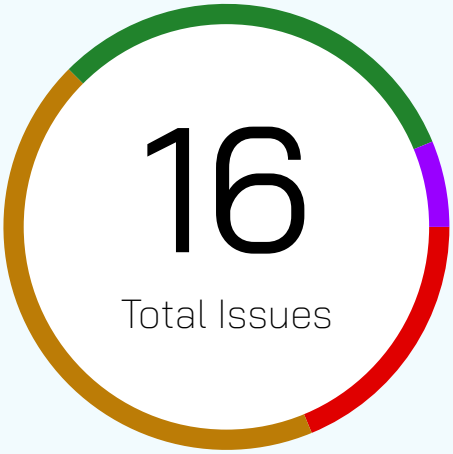
Table of Content

Executive Summary

Project name	Nex Labs
Overview	Nex Labs Cross Chain Model Smart Contracts provide an on-chain infrastructure for managing tokenized stock index portfolios. At its core, the IndexFactory facilitates buying and selling underlying assets, minting and burning IndexTokens that represent a holder's share in the portfolio. FactoryBalancer exists for the purpose of reweighting and rebalancing of assets, leveraging price data from FactoryStorage, which communicates with oracles. The Vault securely holds underlying assets, ensuring transparency and liquidity. Ownership and governance are managed through ProposableOwnable, enabling decentralized control.
Audit Scope	The scope of this Audit was to analyze the Nex Labs Cross Chain Model Smart Contracts for quality, security, and correctness.
Contracts	ccip/CCIPReceiver.sol chainlink/ConfirmedOwner.sol chainlink/ConfirmedOwnerWithProposal.sol chainlink/FunctionsClient.sol factory/IndexFactory.sol factory/IndexFactoryBalancer.sol factory/IndexFactoryStorage.sol factory/PriceOracle.sol factory/IPriceOracle.sol factory/OrderManager.sol libraries/FeeCalculation.sol librares/MessageSender.sol libraries/SwapHelpers.sol proposable/ProposableOwnable.sol proposable/ProposableOwnableUpgradeable.sol token/IndexToken.sol vault/CrossChainFactory.sol vault/Vault.sol
Commit Hash	bdf96dae49d913239329dc85da75bbe24d2e664d

Language	Solidity
Blockchain	Arbitrium
Method	Manual Review, Functional Testing, Automated Testing, etc. All the raised flags were manually reviewed and re-tested to identify any false positives.
review 1	12th February 2025 - 28th February 2025
Review 2	12th March 2025 to 20th March 2025
Fixed In	Branch: Audit changes c0130979eb9da82cd4e7dba2febe80d7a56ad622

Number of Issues per Severity



High	3 (18.75%)
Medium	7 (43.75%)
Low	5 (31.25%)
Informational	1 (6.25%)

Severity

Issues

	High	Medium	Low	Informational
Open	0	0	0	0
Resolved	**3**	**7**	**5**	**1**
Acknowledged	0	0	0	0
Partially Resolved	0	0	0	0

Checked Vulnerabilities

- ✓ Access Management
- ✓ Arbitrary write to storage
- ✓ Centralization of control
- ✓ Ether theft
- ✓ Improper or missing events
- ✓ Logical issues and flaws
- ✓ Arithmetic Computations Correctness
- ✓ Race conditions/front running
- ✓ SWC Registry
- ✓ Re-entrancy
- ✓ Timestamp Dependence
- ✓ Gas Limit and Loops
- ✓ Exception Disorder
- ✓ Gasless Send
- ✓ Use of tx.origin
- ✓ Malicious libraries
- ✓ Compiler version not fixed
- ✓ Address hardcoded
- ✓ Divide before multiply
- ✓ Integer overflow/underflow
- ✓ ERC's conformance
- ✓ Dangerous strict equalities
- ✓ Tautology or contradiction
- ✓ Return values of low-level calls
- ✓ Missing Zero Address Validation
- ✓ Private modifier
- ✓ Revert/require functions
- ✓ Multiple Sends
- ✓ Using suicide
- ✓ Using delegatecall
- ✓ Upgradeable safety
- ✓ Using throw

- [x] Using inline assembly
- [x] Unsafe type inference
- [x] Style guide violation
- [x] Implicit visibility level.

Techniques and Methods

Throughout the audit of smart contracts, care was taken to ensure:

- The overall quality of code.
- Use of best practices.
- Code documentation and comments, match logic and expected behavior.
- Token distribution and calculations are as per the intended behavior mentioned in the whitepaper.
- Implementation of ERC standards.
- Efficient use of gas.
- Code is safe from re-entrancy and other vulnerabilities.

The following techniques, methods, and tools were used to review all the smart contracts.

Structural Analysis

In this step, we have analyzed the design patterns and structure of smart contracts. A thorough check was done to ensure the smart contract is structured in a way that will not result in future problems.

Static Analysis

A static Analysis of Smart Contracts was done to identify contract vulnerabilities. In this step, a series of automated tools are used to test the security of smart contracts.

Code Review / Manual Analysis

Manual Analysis or review of code was done to identify new vulnerabilities or verify the vulnerabilities found during the static analysis. Contracts were completely manually analyzed, their logic was checked and compared with the one described in the whitepaper. Besides, the results of the automated analysis were manually verified.

Gas Consumption

In this step, we have checked the behavior of smart contracts in production. Checks were done to know how much gas gets consumed and the possibilities of optimization of code to reduce gas consumption.

Tools And Platforms Used For Audit

Remix IDE, Foundry, Solhint, Mythril, Slither, Solidity statistical analysis.

Types of Severity

Every issue in this report has been assigned to a severity level. There are four levels of severity, and each of them has been explained below

High Severity Issues

A high severity issue or vulnerability means that your smart contract can be exploited. Issues on this level are critical to the smart contract's performance or functionality, and we recommend these issues be fixed before moving to a live environment.

Medium Severity Issues

The issues marked as medium severity usually arise because of errors and deficiencies in the smart contract code. Issues on this level could potentially bring problems, and they should still be fixed.

Low Severity Issues

Low-level severity issues can cause minor impact and are just warnings that can remain unfixed for now. It would be better to fix these issues at some point in the future.

Informational Issues

These are four severity issues that indicate an improvement request, a general question, a cosmetic or documentation error, or a request for information. There is low-to-no impact.

Types of Issues

Open

Security vulnerabilities identified that must be resolved and are currently unresolved.

Resolved

Security vulnerabilities identified that must be resolved and are currently unresolved.

Acknowledged

Vulnerabilities which have been acknowledged but are yet to be resolved.

Partially Resolved

Considerable efforts have been invested to reduce the risk/ impact of the security issue, but are not completely resolved.

High Severity Issues

SwapHelpers::swap() always uses UniswapV3 because of incorrectly checked _swapFee parameter

Resolved

Path

SwapHelpers.sol#

Function

swap()

Description

he swap() function in IndexFactory.sol is intended to support both Uniswap V3 and V2 swaps based on the _swapFee parameter. However, due to input validation requirements in multiple functions, the code always executes the V3 swap path, making the V2 swap functionality unreachable.

The issue stems from several functions requiring _swapFee > 0 in their validation:
1. issuanceIndexTokens() requires _tokenInSwapFee > 0
2. redemption() requires _tokenOutSwapFee > 0
3. _handleReceivedRedemption() hardcodes fee as 3000

These requirements force the swap() function to always take the V3 path since the SwapHelpers.swap() implementation using logic in the image
This means the V2 fallback path is never taken, potentially leading to failed swaps if:
1. The token pair doesn't exist on Uniswap V3
2. The V3 pool has insufficient liquidity
3. The specified fee tier doesn't exist for the pair

```
function swap(
    ISwapRouter uniswapRouter⬆,
    IUniswapV2Router02 uniswapRouterV2⬆,
    uint24 poolFee⬆,
    address[] memory path⬆,
    uint24[] memory fees⬆,
    uint256 amountIn⬆,
    address recipient⬆
) internal returns (uint256 amountOut) {
    if (poolFee⬆ > 0) {
        amountOut = swapVersion3(uniswapRouter⬆, poolFee⬆, path⬆, fees⬆, amountIn⬆, recipient⬆);
    } else {
        amountOut = swapTokensV2(uniswapRouterV2⬆, path⬆, amountIn⬆, recipient⬆);
    }
}
```

Recommendation

1. Remove the _swapFee > 0 requirements from input validation and instead validate that either V2 or V3 path is viable
2. Update the swap routing logic to check pool existence/liquidity before deciding on V2/V3:

POC

1. User calls redemption() with a token pair that only exists on Uniswap V2
2. Function requires _tokenOutSwapFee > 0
3. swap() is called with positive fee, forcing V3 path
4. Transaction reverts due to non-existent V3 pool

Incorrect BPS fee calculation in redemption leads to 30% fee Instead of 0.3%

Resolved

Path

SwapHelpers.sol#

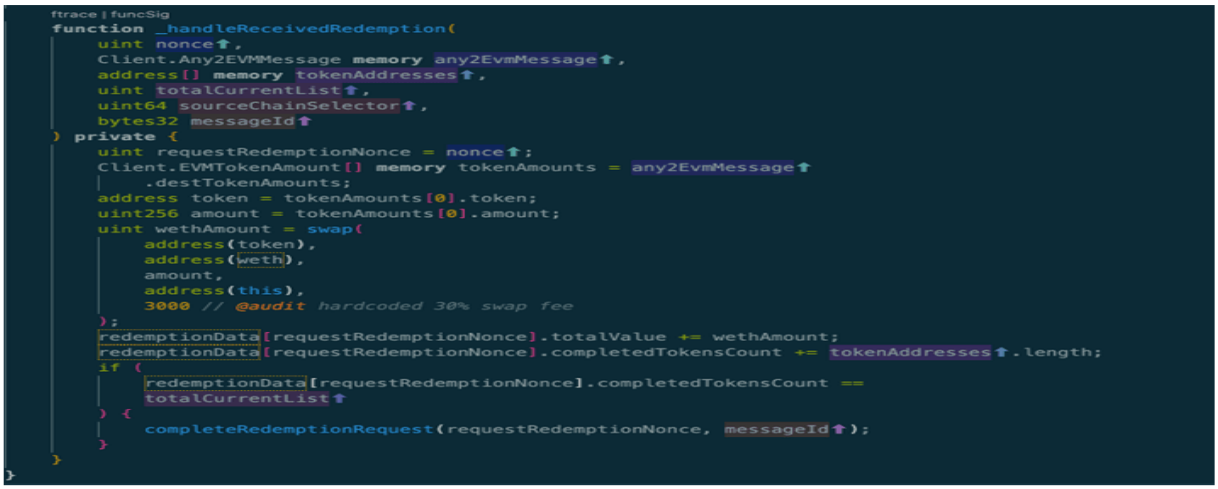
Function

swap()

Description

The IndexFactory contract implements a fee system for token redemptions using basis points (BPS). The fee calculation is performed in the _handleReceivedRedemption() function when processing cross-chain redemption requests. However, there is a critical error in how the swap fee is handled. In the contract, the fee rate is documented and used elsewhere as:
uint8 public feeRate; // 10/10000 = 0.1%
However, in _handleReceivedRedemption(), the swap fee is hardcoded to 3000

This leads to a fee of 3000 basis points (30%) being charged instead of the expected 0.3% (30 basis points), representing a 100x increase in the fee rate compared to what users would expect based on the documentation and other parts of the contract.



```
ftrace | funcSig
function _handleReceivedRedemption(
    uint nonce⬆,
    Client.Any2EVMMessage memory any2EvmMessage⬆,
    address[] memory tokenAddresses⬆,
    uint totalCurrentList⬆,
    uint64 sourceChainSelector⬆,
    bytes32 messageId⬆
) private {
    uint requestRedemptionNonce = nonce⬆;
    Client.EVMTokenAmount[] memory tokenAmounts = any2EvmMessage⬆
        .destTokenAmounts;
    address token = tokenAmounts[0].token;
    uint256 amount = tokenAmounts[0].amount;
    uint wethAmount = swap(
        address(token),
        address(weth),
        amount,
        address(this),
        3000 // @audit hardcoded 30% swap fee
    );
    redemptionData[requestRedemptionNonce].totalValue += wethAmount;
    redemptionData[requestRedemptionNonce].completedTokensCount += tokenAddresses⬆.length;
    if (
        redemptionData[requestRedemptionNonce].completedTokensCount ==
        totalCurrentList⬆
    ) {
        completeRedemptionRequest(requestRedemptionNonce, messageId⬆);
    }
}
}
```

Recommendation

The hardcoded fee value should be replaced with the protocol's standard fee rate

POC

1. User initiates a cross-chain redemption for 1000 tokens.
2. The redemption request is processed on the destination chain via _handleReceivedRedemption().
3. The swap is executed with a fee of 3000 BPS (30%).
4. For a transaction worth 1000 tokens, the user loses 300 tokens to fees instead of the expected 3 tokens (0.3%)

Excess tokens will be lost to CCIP due to high gas limit set

Resolved

Path

.sol#

Function

sendMessage()

Description

The gasLimit specifies the maximum amount of gas CCIP can consume to execute ccipReceive() on the contract located on the destination blockchain. It is the main factor in determining the fee to send a message. Unspent gas is not refunded.

Chainlink documentation

There are instances where different gas limits were set for this kind of call - 900,000 and 3 million. When calls are made to the CCIP with a high gas limit, unspent gas will not be refunded.

https://github.com/nexlabs22/Nex-Cross-Chain-Model/blob/d2839fc6800054db82d3136101fb275f8c35cc31/contracts/vault/CrossChainFactory.sol#L385
https://github.com/nexlabs22/Nex-Cross-Chain-Model/blob/d2839fc6800054db82d3136101fb275f8c35cc31/contracts/vault/CrossChainFactory.sol#L998
https://github.com/nexlabs22/Nex-Cross-Chain-Model/blob/d2839fc6800054db82d3136101fb275f8c35cc31/contracts/libraries/MessageSender.sol#L49

Recommendation

Thoroughly test on local nodes and testnet to help estimate an optimal figure to set for gas limit on every contract interaction.

References

https://docs.chain.link/ccip/best-practices#:~:text=will%20be%20set.-,Setting%20gasLimit,Unspent%20gas%20is%20not%20refunded

https://docs.chain.link/ccip/tutorials/ccipreceive-gaslimit

Medium Severity Issues

Validate that incoming Chainlink price data is not stale

Resolved

Path

IndexFactoryStorage

Function

priceInWei

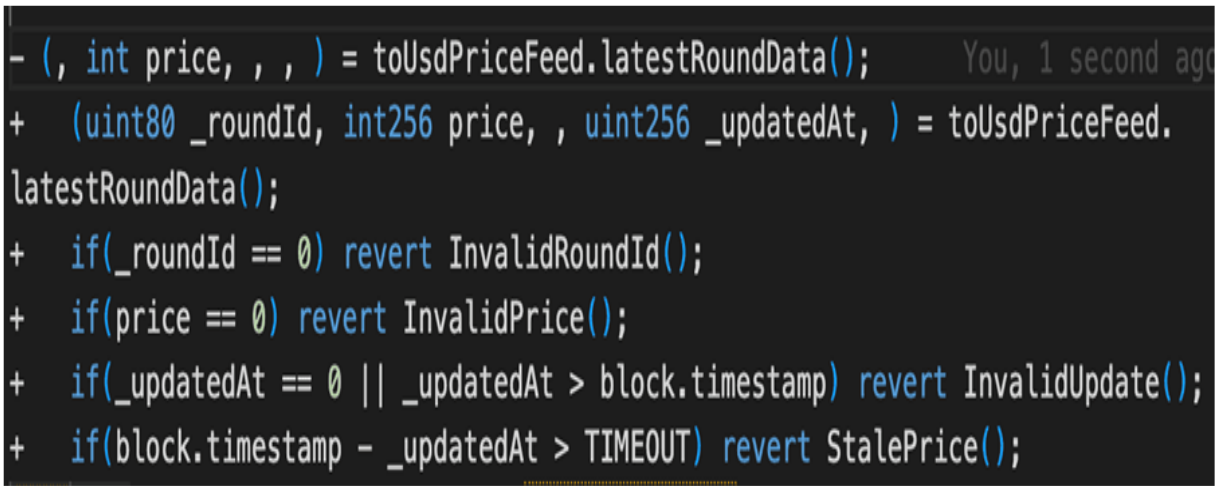
Description

The Index smart contract integrates Chainlink oracle for price derivation. However, when it fetches the price of a token, it fails to check that the price value is not stale as it is possible for an attacker to manipulate this.

```
/**
 * @dev Returns the price in Wei.
 * @return The price in Wei.
 */
ftrace | funcSig
function priceInWei() public view returns (uint256) {
    (, int price, , , ) = toUsdPriceFeed.latestRoundData();
    uint8 priceFeedDecimals = toUsdPriceFeed.decimals();
    price = _toWei(price, priceFeedDecimals, 18);
    return uint256(price);
}
```

Recommendation

Add check for stale price



```
- (, int price, , , ) = toUsdPriceFeed.latestRoundData();      You, 1 second ago
+   (uint80 _roundId, int256 price, , uint256 _updatedAt, ) = toUsdPriceFeed.
latestRoundData();
+   if(_roundId == 0) revert InvalidRoundId();
+   if(price == 0) revert InvalidPrice();
+   if(_updatedAt == 0 || _updatedAt > block.timestamp) revert InvalidUpdate();
+   if(block.timestamp - _updatedAt > TIMEOUT) revert StalePrice();
```

Reference

https://forum.openzeppelin.com/t/what-does-disableinitializers-function-mean/28730

Missing whenNotPaused Modifier on critical entry points

Resolved

Path

IndexFactory.sol#L206, IndexFactoryBalancer.sol

Function

whenNotPaused

Description

These contracts implement the pausable functionality that prevents users from calling important entry points in time of emergency. In the IndexFactory, the internal functions (_pause and _unpause) of the Pausable contract were appropriately invoked and the external functions had onlyOwner modifier. However, the implementation is incomplete as the “whenNotPaused” modifier was not attached to any critical functions. This implies that in time of emergency when the admin calls the pause function, the activity of the contract does not exactly pause. Users will proceed with their activity.

Recommendation

Add the whenNotPaused modifier to critical functions that are meant to halt when the protocol faces an emergency situation. If not required for contracts where it was inherited but unimplemented, remove.

Reference

https://forum.openzeppelin.com/t/what-does-disableinitializers-function-mean/28730

Missing 'disableInitializers` in Proxy Upgradable Contract Constructors

Resolved

Path

IndexFactory.sol

Description

The failure to disable initializers at the constructor level of the proxy upgradeable contract raises concern as it is susceptible to exploit. This oversight presents the potential of attackers initializing the implementation contract itself.

Recommendation

Invoke the _disableInitializers() at the constructor.

```
// @custom:oz-upgrades-unsafe-allow constructor
constructor() {
    _disableInitializers();
}
```

Reference

https://forum.openzeppelin.com/t/what-does-disableinitializers-function-mean/28730

Owner can mint tokens above the supply ceiling as this check does not exist in mintToFeeReceiver()

Resolved

Path

IndexToken.sol#L190

Function

mintToFeeReceiver()

Description

The mint function of the IndexToken contract has 5 require statements to check for address validity and the supply cap being less or equal to the supplyCeiling, however mintToFeeReceiver() function does not have a similar check for token supply minted. This would make the owner able to mint new tokens above the supplyCeiling set and also cause a DOS the next time an address with MINTER role calls mint() because totalSupply() + amount would already have exceeded the supplyCeiling.
require(totalSupply() + amount <= supplyCeiling, "will exceed supply ceiling");

If the owner address gets hijacked at any point, the compromised account would be able to mint new tokens without any restrictions leading to economic loss for legitimate users of the protocol.

Recommendation

Include the same require check in the mintToFeeReceiver function.

Sandwich attack possibility

Resolved

Path

SwapHelpers

Function

swap()

Description

It was noticed that the minAmountOut is set to zero during interactions with the AMM router (Uniswap) for swaps. These parameters are intended to protect against slippage; setting them to zero exposes transactions to slippage, potentially resulting in receiving or adding smaller amounts than intended.

Recommendation

Instead of hard-coding the minimum amounts to zero, introduce a state variable to control the slippage percentage (e.g., 80% of the swapped or added amounts, with a setter to update the slippage during high market volatility) and utilize it when interacting with the AMM.

IndexFactoryStorage truncates uint256 to uint128 in getAmountOut() leading to potential value loss

Resolved

Description

getAmountOut() accepts a uint256 parameter amountIn but silently truncates it to uint128 when calling estimateAmountOut(), which could lead to incorrect calculations for large token amounts. In IndexFactoryStorage.sol, the getAmountOut() function is used to calculate expected output amounts for token swaps and is a critical component for determining portfolio balances.

This truncation could lead to incorrect calculations if the input amount exceeds type(uint128).max, particularly when calculating large portfolio balances through getPortfolioBalance().

```
function getAmountOut(
    address tokenIn⬆,
    address tokenOut⬆,
    uint amountIn⬆, // @audit accepts a uint256 but gets truncated to u128
    uint24 _swapFee⬆
) public view returns (uint finalAmountOutValue) {
    uint finalAmountOut;
    if (amountIn⬆ > 0) {
        if (_swapFee⬆ > 0) {
            finalAmountOut = estimateAmountOut(
                tokenIn⬆,
                tokenOut⬆,
                uint128(amountIn⬆),
                _swapFee⬆
            );
        } else {
```

Recommendation

1. Add a require check to ensure the input amount doesn't exceed type(uint128).max
2. Update getAmountOut() to take uint128 values

swapV2 will always revert for ordinary tokens that have no fee-on-transfer.

Resolved

Path

SwapHelpers.sol#

Function

swapV2()

Description

Identical to swapExactTokensForTokens, but succeeds for tokens that take a fee on transfer https://docs.uniswap.org/contracts/v2/reference/smart-contracts/router-02#swapexacttokensfor-tokenssupportingfeeontransfertokens

Recommendation

Use the swapExactTokensForTokens to allow only ordinary tokens or design the implementation for users to identify if tokenIn during issuance of index tokens or tokenOut during redemption, has fee on transfer.

Low Severity Issues

Crosschain fee parameter is unused in issuance and redemption functions

Resolved

Path

IndexFactory.sol

Function

issuance(), redemption()

Description

The _crossChainFee parameter in issuanceIndexTokens() and redemption() functions is accepted but never used in the actual fee calculations or token transfers, making it a redundant parameter that could mislead users and integrators.

Although it doesn't affect the security or functionality of the contract, it creates confusion for users who may expect cross-chain fees to be charged when they are not. This could lead to integration issues or user confusion.

```
/**
 * @dev Issues index tokens.
 * @param _tokenIn The address of the input token.
 * @param _inputAmount The amount of input token.
 * @param _crossChainFee The cross-chain fee.
 * @param _tokenInSwapFee The swap version of the input token.
 */
ftrace | funcSig
function issuanceIndexTokens(
    address _tokenIn⬆,
    uint _inputAmount⬆,
    uint _crossChainFee⬆,
    uint24 _tokenInSwapFee⬆
) public {
    // Validate input parameters
    require(_tokenIn⬆ != address(0), "Invalid input token address");
    require(_inputAmount⬆ > 0, "Input amount must be greater than zero");
    require(_crossChainFee⬆ >= 0, "Cross-chain fee must be non-negative");
    require(_tokenInSwapFee⬆ > 0, "Invalid input token swap fee");
```

```
/**
 * @dev Redeems tokens.
 * @param amountIn The amount of input tokens.
 * @param _crossChainFee The cross-chain fee.
 * @param _tokenOut The address of the output token.
 * @param _tokenOutSwapFee The swap version of the output token.
 */
ftrace | funcSig
function redemption(
    uint amountIn⬆,
    uint _crossChainFee⬆,
    address _tokenOut⬆,
    uint24 _tokenOutSwapFee⬆
) public {
    // Validate input parameters
    require(amountIn⬆ > 0, "Amount must be greater than zero");
    require(_crossChainFee⬆ >= 0, "Cross-chain fee must be non-negative");
    require(_tokenOut⬆ != address(0), "Invalid output token address");
    require(_tokenOutSwapFee⬆ > 0, "Invalid output token swap fee");
    uint burnPercent = (amountIn⬆ * 1e18) / indexToken.totalSupply();
```

Recommendation

Either implement the cross-chain fee functionality or remove the unused _crossChainFee parameter from both functions to improve code clarity and prevent confusion. If cross-chain fees are planned for future implementation, this should be clearly documented.

setFeeRate requires more time than 12 hours to reset fee rate

Resolved

Path

IndexfactoryStorage.sol

Function

setFeeRate()

Description

The setFeeRate is conditioned to be called after 12 hours from the last time of update. After 12 hours and 59 minutes, it's impossible to reset the fee rate until 13 hours.

Recommendation

Use >= notation rather than just > in order to reopen the ability to reset fee at the 12th hour.

Excess ETH will be stuck

Resolved

Path

IndexFactory.sol

Function

issuanceIndexTokensWithETH

Description

Users are allowed to pass in any amount of ETH via msg.value which could be much greater than the needed input amount, and when this happens there would be no way to process a refund

Recommendation

Restrict msg.value to the exact amount needed to cover for protocol fees and leave no extra inaccessible in the contract.

receive() function causes inconsistency across multiple contracts

Resolved

Description

The in-line documentation of the IndexFactory states that users would not be able to pass in external funds via the receive() function but the code implementation in receive() allows for ETH to be sent into the contract without interacting with a payable function.

Whereas, the IndexFactoryBalancer contract cannot receive ether which renders the withdraw function useless as no other function is marked payable to allow ETH deposits. The withdraw function cannot be called since no ether enters the contract.

```
/**
 * @dev The contract's fallback function that does not allow direct payments to the contract.
 * @notice Prevents users from sending ether directly to the contract by reverting the transaction.
 */
ftrace
receive() external payable {
    revert("DoNotSendFundsDirectlyToTheContract");
}

// Function to withdraw Ether from the contract
ftrace | funcSig
function withdraw(uint256 amount↑) external onlyOwner {
    require(amount↑ <= address(this).balance, "Insufficient balance");
    (bool success, ) = payable(owner()).call{value: amount↑}("");
    require(success, "Transfer failed");
}
```

```
/**
 * @dev The contract's fallback function that does not allow direct payments to the contract.
 * @notice Prevents users from sending ether directly to the contract by reverting the transaction.
 */
ftrace
receive() external payable {
    // revert DoNotSendFundsDirectlyToTheContract();
}
```

Recommendation

Ensure code-documentation consistency.

Minting tokens to the feeReceiver becomes very expensive if not done frequently

Resolved

Path

token/IndexToken.sol

Function

_mintToFeeReceiver()

Description

The current arithmetic implementation for minting tokens to the fee receiver is heavily gas intensive because it performs the arithmetic for compounding with a loop. This would become considerably expensive when the number of days grows to a large number.

With current tests, it costs ~500k gas units to call this function when 300 days have passed from the initial feeTimestamp value. Conversely, using a logarithmic approximation for compounding is just as precise with ~80% gas savings.

Informational Severity Issues

Unused state variables and functions

Resolved

Description

In the codebase, Some Variables,functions and Imports are not used

Recommendation

Use unused state variables and functions. Remove unused import

Functional Tests

Some of the tests performed are mentioned below:

- Logarithmic compounding is more efficient
- Test issuance with ETH twice
- Test issuance with USDC twice
- Test redemption with ETH twice
- Test redemption with USDC twice
- Test should fail with Amount must be greater than zero if redemption is called before issuance
- Test issuance with higher gas fees.

Automated Tests

No major issues were found. Some false positive errors were reported by the tools. All the other issues have been categorized above according to their level of severity.

Closing Summary

In this report, we have considered the security of Nex Labs. We performed our audit according to the procedure described above.

Some issues of High, Medium, Low and informational severity were found. Some suggestions, gas optimizations and best practices are also provided in order to improve the code quality and security posture.

Disclaimer

At QuillAudits, we have spent years helping projects strengthen their smart contract security. However, security is not a one-time event—threats evolve, and so do attack vectors. Our audit provides a security assessment based on the best industry practices at the time of review, identifying known vulnerabilities in the received smart contract source code

This report does not serve as a security guarantee, investment advice, or an endorsement of any platform. It reflects our findings based on the provided code at the time of analysis and may no longer be relevant after any modifications. The presence of an audit does not imply that the contract is free of vulnerabilities or fully secure.

While we have conducted a thorough review, security is an ongoing process. We strongly recommend multiple independent audits, continuous monitoring, and a public bug bounty program to enhance resilience against emerging threats.

Stay proactive. Stay secure.

About QuillAudits

QuillAudits is a secure smart contracts audit platform designed by QuillHash Technologies. We are a team of dedicated blockchain security experts and smart contract auditors determined to ensure that Smart Contract-based Web3 projects can avail the latest and best security solutions to operate in a trustworthy and risk-free ecosystem

7+ Years of Expertise	**1M+** Lines of Code Audited
$30B+ Secured in Digital Assets	**1400+** Projects Secured

Follow Our Journey

AUDIT REPORT

March, 2025

For

QuillAudits

Canada, India, Singapore, UAE, UK

www.quillaudits.com audits@quillaudits.com