QuillAudits

AUDIT REPORT

March, 2025

For

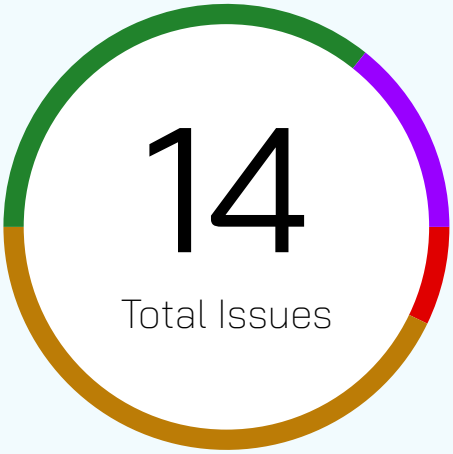
Table of Content

Executive Summary

Project name	Nex Labs
Project URL	https://www.nexlabs.io/
Overview	Nex Labs Defi Indices 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 Defi Indices Model Smart Contracts for quality, security, and correctness.
Source code	https://github.com/nexlabs22/Defi-Indices-Model-Contracts/tree/main/contracts
Commit Hash	bbd54b50b294ee2ff8f2a7a1541956017b93475f
Language	Solidity
Blockchain	Arbitrum
Method	Manual Analysis, Functional Testing, Automated Testing

Contracts in Scope	-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/SwapHelpers.sol -proposable/ProposableOwnable.sol -proposable/ProposableOwnableUpgradeable.sol -token/IndexToken.sol -vault/Vault.sol
Review 1	31st January 2025 - 10th February 2025
Review 2	12th March 2025 to 20th March 2025
Fixed In	Branch: Audit Changes 5d765c2e0bcfdbbc7ad6303d85c6bac5f6789f4d

Number of Issues per Severity



High	1 (7.14%)
Medium	6 (42.86%)
Low	5 (35.71%)
Informational	2 (14.29%)

Severity

Issues	High	Medium	Low	Informational
Open	0	0	0	0
Resolved	**1**	**6**	**5**	**2**
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

_swapFee/poolFee variable does not apply for uniswapV2

Resolved

Path

SwapHelpers.sol#

Function

swap()

Description

IndexFactory::swap takes in a _swapFee parameter which calls SwapHelpers::swap as poolFee. Now poolFee is checked to be greater than 0 to run the swapVersion3 code block and if it is 0 it will run the swapTokensv2 (else block) instead because as per the NATSPEC comment on the swap function, 2 or 3 is expected to signify v2 or v3 respectively.

Only the if block will run in any scenario because poolFee is set to 2 or 3 which is always greater than 0.

```
/**
 * @dev Internal function to swap tokens.
 * @param path The path of the tokens to swap.
 * @param fees The fees of the tokens to swap.
 * @param amountIn The amount of input token.
 * @param _recipient The address of the recipient.
 * @param _swapFee The swap version (2 for Uniswap V2, 3 for Uniswap V3). // @audit-info
 * @return outputAmount The amount of output token.
 */
ftrace | funcSig
function swap(
    address[] memory path↑,
    uint24[] memory fees↑,
```

```
ftrace | funcSig
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

Properly implement the conditional checks to avoid having a code execution path that will never run.

Medium Severity Issues

Validate that incoming Chainlink price data is not stale

Resolved

Path

IndexFactoryStorage.sol#L251

Function

priceInWei

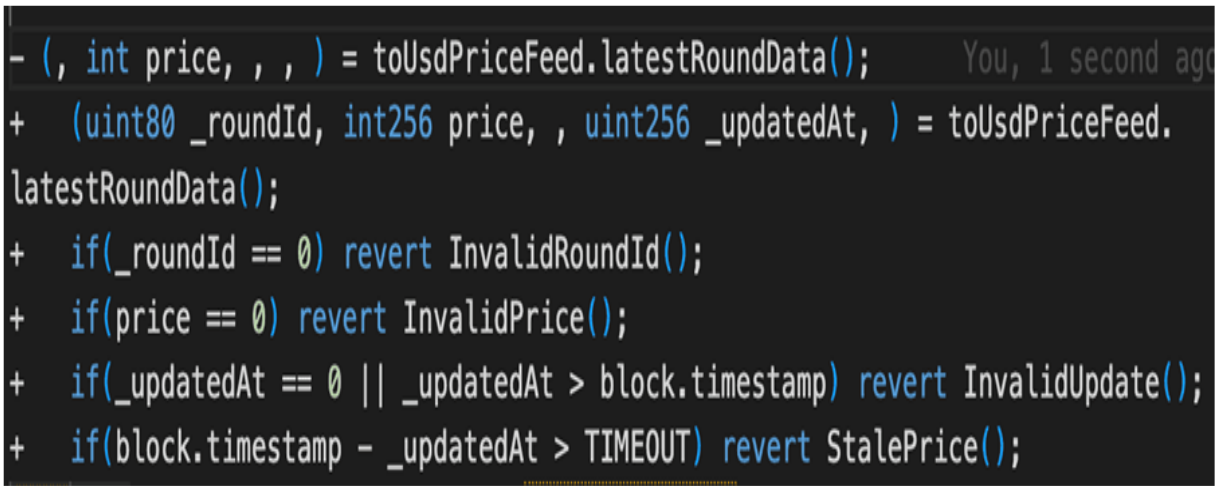
Description

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#L255

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.

```
4
5  // @custom:oz-upgrades-unsafe-allow constructor
6  constructor() {
7      _disableInitializers();
8  }
9
```

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.
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.

```
require(totalSupply() + amount <= supplyCeiling, "will exceed supply ceiling");
```

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.

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#swapexacttokens-fortokenssupportingfeeontransfertokens

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

setFeeRate requires more time than 12 hours to reset fee rate

Resolved

Path

IndexfactoryStorage.sol#L4

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.

Use safeTransferLib for safety transfer of tokens

Resolved

Path

IndexFactory.sol#L222

Function

transferFrom()

Description

With the use of safeTransferLib, there is assurance of safety of transfer of tokens.

Recommendation

Use safeTransferLib and replace transferFrom() with safeTransferFrom().

Owner is allowed to mint tokens to the feeReceiver even when the contract is paused

Resolved

Path

IndexToken.sol#L190

Function

mintToFeeReceiver()

Description

The functionality of the pausable contract does not apply to the mintToFeeReceiver function and allows for the owner to continue token minting to the fee receiver while the contract is paused.

Recommendation

Add whenNotPaused to the mintToFeeReceiver().

Excess ETH will be stuck in the IndexFactory

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

Path

IndexFactory.sol, IndexFactoryBalancer.sol

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();
}
```

```
/**
 * @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");
}
```

Recommendation

Ensure code-documentation consistency.

Informational Severity Issues

Unused state variables and functions

Resolved

Path

factory/IndexFactoryStorage.sol

Function

concatenation(), toWei(), mockFillAssetsList()

Description

There are mock functions that appear to be used only for testing functionality within the contract because they fill storage variables with dummy data, and can be used as view/pure functions that do not directly affect any other functions

Recommendation

Use unused state variables and functions.

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.

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