Nexus Mutual Stacked Risk, On-chain MCR, and Swap Operator Smart Contract Audit

1. Introduction

iosiro was commissioned by Nexus Mutual to conduct a smart contract audit on their Stacked Risk, On-chain MCR and Swap Operator features. The audit was performed between 4 and 11 May 2021, by two auditors, using a total of 10 resource days. A review of vulnerability remediations and the final merged branch was performed between 18 and 19 May 2021 by a single auditor, using a total of two resource days.

This report is organized into the following sections.

- **Section 2 - Executive summary:** A high-level description of the findings of the audit.
- **Section 3 - Audit details:** A description of the scope and methodology of the audit.

- **Section 4 - Design specification:** An outline of the intended functionality of the smart contracts.
- **Section 5 - Detailed findings:** Detailed descriptions of the findings of the audit.

The information in this report should be used to better understand the risk exposure of the smart contracts, and as a guide to improving the security posture of the smart contracts by remediating issues identified. The results of this audit are only a reflection of the source code reviewed at the time of the audit and of the source code that was determined to be in-scope.

The purpose of this audit was to achieve the following:

- Identify potential security flaws.
- Ensure that the smart contracts functioned according to the documentation provided.

Assessing the off-chain functionality associated with the contracts, for example, backend web application code, was out of scope of this audit.

Due to the unregulated nature and ease of transfer of cryptocurrencies, operations that store or interact with these assets are considered very high risk with regards to cyber attacks. As such, the highest level of security should be observed when interacting with these assets. This requires a forward-thinking approach, which takes into account the new and experimental nature of blockchain technologies. Strategies that should be used to encourage secure code development include:

- Security should be integrated into the development lifecycle and the level of perceived security should not be limited to a single code audit.
- Defensive programming should be employed to account for unforeseen circumstances.
- Current best practices should be followed where possible.

2. Executive summary

This report presents the findings of an audit performed by iosiro on Nexus Mutual's Stacked Risk, On-chain MCR and Swap Operator features.

Stacked Risk

Two medium-risk issues, two informational issues and a number of design comments were identified during this audit. The issues found related to access control on sensitive functionality and development best practices. The medium-risk issues and the majority of design comments were resolved. The remaining informational issues relate to development best practices on single-use functionality, and are not considered to pose a risk.

On-chain MCR

A number of design related and code quality issues were identified during the audit. These were resolved.

Swap Operator

A single low risk issue was identified, due to missing authorization validation on a pool's target asset. Suggestions for design improvement were also made, including code refactoring and gas optimization. All issues and design comments were resolved.

3. Audit details

3.1 Scope

The source code considered in-scope for the assessment is described below. Code from all other files was considered to be out-of-scope. Out-of-scope code that interacts with in-scope code was assumed to function as intended and not introduce any functional or security vulnerabilities.

3.1.1 Smart contracts

- **Project name:** Nexus Mutual Stacked Risk, On-Chain MCR and Swap Operator
- **Commits:** Final audit commit: 60efdf6; Final re-audit commit: d38ab41

- **Files:** Claims.sol, ClaimsReward.sol, Incidents.sol, Quotation.sol, TokenFunctions.sol, MCR.sol, Pool.sol, SwapOperator.sol

3.2 Methodology

A variety of techniques, described below, were used to conduct the audit.

3.2.1 Code review

The source code was manually inspected to identify potential security flaws. Code review is a useful approach for detecting security flaws, discrepancies between the specification and implementation, design improvements, and high risk areas of the system.

3.2.2 Dynamic analysis

The contracts were compiled, deployed, and tested in a Hardhat test environment, both manually and through the test suite provided. Manual analysis was used to confirm that the code was functional and to discover whether any potential security issues identified could be exploited.

3.2.3 Automated analysis

Tools were used to automatically detect the presence of several types of security vulnerabilities, including reentrancy, timestamp dependency bugs, and transaction-ordering dependency bugs. The static analysis results were manually reviewed and any false positives were removed from the results. Any true positive results were included in this report.

Static analysis tools commonly used include Slither, Securify, and MythX. Tools such as the Remix IDE, compilation output, and linters could also be used to identify potential areas of concern.

3.3 Risk ratings

Each issue identified during the audit has been assigned a risk rating. The rating is determined based on the criteria outlined below.

- **High risk** - The issue could result in a loss of funds for the contract owner or system users.

- **Medium risk** - The issue resulted in the code specification being implemented incorrectly.
- **Low risk** - A best practice or design issue that could affect the security of the contract.
- **Informational** - A lapse in best practice or a suboptimal design pattern that has a minimal risk of affecting the security of the contract.
- **Closed** - The issue was identified during the audit and has since been addressed to a satisfactory level to remove the risk that it posed.

4. Design specification

The following section outlines the intended functionality of the system at a high level. The specification is based on the specification documents provided by Nexus Mutual as well as the implementation in the codebase. Any perceived points of conflict should be highlighted with the auditing team to determine the source of the discrepancy.

4.1 Stacked Risk

The Stacked Risk feature (also called Yield Token Cover) introduced a new cover product for yield tokens. Users are able to buy cover for supported yield tokens (e.g. yDAI) to insure against the risk of these tokens losing their intended peg with an underlying currency. The implementation simplified the membership and cover purchasing flow, and introduced an `Incidents` contract for handling payouts from incidents. The intended process flows are detailed below.

Cover process

Users buy Yield Token Cover through the same process as they would buy other cover products. Each Yield Token Cover product must be purchased with the corresponding currency (for example, stETH cover is purchased with ETH and yDAI cover is purchased with DAI).

Claim process

1. When an incident occurs that causes a Yield Token to lose its price peg, the Advisory Board creates a governance proposal to add the incident.
2. If members vote to approve the proposal, the new incident is added through the Incidents contract.
3. Users who have active cover at the time of the incident (or whose cover has expired recently, according to Nexus Mutual's defined claim grace period) will be able to call `Incidents.redeemPayout(...)` with the amount of YieldTokens they wish to exchange for the underlying token.
4. The tokens sent will be exchanged for the underlying tokens (e.g. yDAI for DAI) at the pre-incident rate defined when the incident was created, adjusted by the contract's `DEDUCTIBLE_RATIO` (currently 90%).
    - For example, a pre-incident rate of 10 DAI to 1 yDAI would yield the payee 9 DAI for every 1 yDAI sent. The amount exchanged can be less than the full amount covered.
    - Only a single claim per incident is permitted.
5. Following the payout, a claim will be created in `QuotationData` and set to "Claim Accepted Payout Done", simulating the claim process in other products.
6. An amount of NXM tokens will be burned following payout. This will be the payout amount divided by the price of NXM in the product's underlying token, multiplied by the contract's `BURN_RATIO` (currently 20%).

Additionally:

- The Incidents contract burns NXM tokens from Pooled Staking in batches, to save gas.
- The Advisory Board can withdraw a specified amount of a specified asset to a specified destination address, subject to the outcome of a governance vote.
- The `BURN_RATIO` and `DEDUCTIBLE_RATIO` can be altered through successful governance proposals and both have a maximum value of 100%.

4.2 On-chain MCR

The mutual fund has to maintain a Minimum Capital Ratio (MCR) of assets in the Pool in order to be able confidently payout any claims. The MCR is a key component of the pricing formula for NXM tokens and thus is critical to the overall security of the system.

The MCR was previously calculated off-chain and updated daily. Over time the MCR formula has been simplified, making it feasible to implement the calculations on-chain. A key requirement of the MCR is that it should change gradually over time as to prevent large NXM price fluctuations. The implementation also aims to continually increase the absolute minimum of the MCR if the Pool has an excess of funds, this is to ensure the value of the mutual increases over time.

The new MCR update functionality can be triggered by calling either `updateMCR()`, when buying or selling NXM, or when a claim is paid out. The synced MCR will only update if at least one hour has passed since the previous update, with the exception of when a claim is paid out. The MCR is not explicitly updated when a member buys cover or if cover expires, as it was confirmed by Nexus Mutual that this is to minimize the gas costs for these operations.

The MCR floor will increase when ever the MCR ratio (the ether value of the pool compared to the current MCR) exceeds 130%. The increase is capped to 1% per day (compounded hourly).

During the update, the instantaneous MCR is calculated and stored to determine the direction and end value of the reported MCR over time. The `desiredMCR` is a function of the total ether value of the assured cover, the current ether value of all assets in the pool and the gear factor. It can never be less than the `mcrFloor`.

The result returned by `getMCR()` continuously slides towards the `desiredMCR` at a rate of up to 5% per day (compounded hourly), but never exceeds a total change exceeding 1% of its last synced value.

The MCR smart contract was regarded as a new during the audit, as the changes required replaced most of the existing code.

4.3 Swap Operator

The previous `SwapAgent` smart contract was replaced with the new `SwapOperator` contract, which allows for more dynamic trading strategies. The contract allows the `SwapController` to exchange assets for ether or vice-versa, given that the asset's balance is outside the limits voted on by the members.

The swap logic (`swapETHForAsset` and `swapAssetForETH`) was removed from the `Pool` contract and added to the `SwapOperator` . To maintain the existing functionality, functions to transfer assets between the `Pool` and the `SwapOperator` were added. All assets are returned to the `Pool` at the end of the swap. The implementation does however introduce an intermediate transfer that could result in significant gas fees. An invoker pattern should be investigated if the gas fees are deemed to be exorbitant.

The contract also introduced a new function (`swapETHForStETH`) to allow the `SwapController` to deposit ether in the Lido smart contract and receive `stEth` in return. No functionality to redeem or burn the `stEth` was implemented, as the functionality is intended to only become available when Ethereum 2.0 goes live.

The implications of locking a portion of the mutual's assets on the MCR were assumed to be part of the design. If a large portion of the Pool's funds were to be staked, the contract may be unable to payout claims even if an MCR ratio greater than 100% is maintained. This would be due to a lack of liquidity in the `stETH/ETH` Uniswap pair or the burn functionality not being available yet.

5. Detailed findings

The following section details the findings of the audit.

5.1 High risk

No high-risk issues were present at the conclusion of the review.

5.2 Medium risk

No medium-risk issues were present at the conclusion of the review.

5.3 Low risk

No low-risk issues were present at the conclusion of the review.

5.4 Informational

5.4.1 Use of `transfer` function

Quotation.sol#L355

Description

The `Quotation.freeUpHeldCovers()` functions made use of the `transfer()` function to send ether. While `transfer()` is commonly used to prevent reentrancy attacks due to its 2300 gas limit, it relies on the receiving contract having a fallback function below this limit. As demonstrated in EIP-1884, which changed the gas cost of the `SLOAD` operation, gas costs can change. This could lead to a case where a contract has its fallback function increased above the 2300 limit, resulting in it becoming incompatible with the system. More information can be found in Consensys On Avoiding `transfer()`.

Recommendation

It is recommended that the `call()` function be used to send ether instead of `transfer(...)`. Alternatively, the `sendValue(...)` function from the OpenZeppelin Address library could be used.

Update

As the `freeUpHeldCovers()` function will only be invoked once, this issue does not present a risk.

5.4.2 Fallback implementations could revert `freeUpHeldCovers`

Quotation.sol#L386

Description

The `freeUpHeldCovers()` function loops through the first 106 covers and checks whether the member should be refunded. This is to recover any funds that might be locked in the contract after the removal of various KYC functions.

Since there is no on-chain validation that the `userAddress` is an Externally Owned Account (EOA) and since members can migrate their addresses, it is possible that the `userAddress` might be a smart contract that is no longer payable or that explicitly reverts when calling its fallback function. This will revert the entire transaction and prevent successful execution of the `freeUpHeldCovers` function.

Recommendation

`Quotation.freeUpHeldCovers()` should be changed to a batch function which takes an array of `coverIDs`. This would allow more selective execution of the function and avoid unintended consequences of attempting to pay to contract addresses.

Update

All addresses affected by the single invocation of `freeUpHeldCovers()` were determined to be externally owned accounts. Fork tests on this function ran without issue.

5.4.3 Design comments

Actions to improve the functionality and readability of the codebase are outlined below.

Defense in depth

The `Incidents.redeemPayout(...)` function can be used to redeem payouts against other cover products, provided the `contractAddress` for that product matches its `productID`. Discussions with Nexus Mutual revealed that there were off-chain processes to prevent this from happening; however, on-chain protection against this issue would provide a further layer of protection. It is recommended that a check is added to `redeemPayout(...)` to ensure that the provided `coverId` is a supported product type.

Update

The scenario in which this would pose a risk is considered highly improbable, and introducing on-chain defense is not considered a worthwhile trade-off against issues such a code addition may introduce.

Refactoring suggestions

Portions of the code can be refactored to improve readability and consistency, as indicated below.

1. SwapOperator#L267: The `swapETHForStETH` function stakes `ether` by directly depositing into the Lido smart contract. No exchange is used and it is still possible to use `swapETHForAsset` to obtain `stEth` using an exchange. The function name should reflect its implementation, a suitable name would be similar to `stakeETHforStETH`.

5.5 Closed

5.5.1 Incorrect modifier for `addIncident` (medium risk)

Incidents.sol#L119

Description

At the start of the audit the `addIncident` made use of the `onlyAdvisoryBoard` modifier opposed to the `onlyGovernance` modifier. This would have allowed any member of the Advisory Board to add a product, take out cover and then perform a payout.

Recommendation

Change the `addIncident` modifier to `onlyGovernance` as per the design specification.

Update

This issue was independently identified by Nexus Mutual and resolved during the audit in commit `978f153`.

5.5.2 Unrestricted initializer (medium risk)

Incidents.sol#L82

Description

The `initialize(...)` function of the Incidents contract was callable by any address. This may allow an attacker to front run the contract's initialization immediately following its deployment and set an arbitrary burn rate.

Recommendation

Discussions with Nexus Mutual indicated that the contract's initial burn rate could be hard-coded. This would remove the need for the `initialize(...)` function, allowing it to be removed from the contract code.

Should initialization be required for this or other contracts in future, the `initialize(...)` function should be protected by a modifier that restricts initialization to a trusted address or addresses.

Updated

In 7141a96, the `initialize` function was altered to have no parameters and set `BURN_RATE` to a hard-coded value of 20.

5.5.3 No explicit check to ensure asset is valid (low risk)

SwapOperator.sol#L80, *SwapOperator.sol#L111*

Description

The `SwapOperator.sol` smart contract did not validate that assets were pool assets. Instead, the code assumed the asset address was valid and proceeded with the default values returned from the mapping.

Fortunately, the max and min validations at the end of the swap functions reverted for an asset not part of the pool. This impacts the maintainability of the codebase, as future developments might overlook the extended implications of changing the min and max validations.

Recommendation

The functions should validate that the asset address is in the `Pool.assets` array before attempting to retrieve the asset's details. Adding unit tests to specifically test this scenario would also help prevent any future code changes from introducing a flaw related to this issue.

Update

In a0174ef, checks were added to ensure that the asset was valid by ensuring that the asset's `minAmount` and `maxAmount` were not both equal to 0.

5.5.4 Design comments (informational)

Actions to improve the functionality and readability of the codebase are outlined below.

Refactoring suggestions

Portions of the code can be refactored to improve readability and consistency, as indicated below.

1. Incidents.sol#L173,#L278: `decimalPrecision` is defined and used in multiple places and could be converted to a contract-level constant.
2. Incidents.sol#L216: The parameter `iterations` could be renamed to `maxIterations` to match its equivalent in the PooledStaking contract.
3. SwapOperator.sol#L67, #L141 #L98, #L201: The `SwapOperator.sol` smart contract defined two pairs of external and internal functions with identical names. This can cause confusion when calling the function internally or when reviewing the code. It is best practice to append an underscore to internal functions to prevent any ambiguity. Furthermore, having identical function names can result in some tools omitting or merging results when analyzing the source code.
4. MCR.sol#L35-L37 Some of the threshold values in the smart contracts are proportional and not absolute. To improve code readability, the state variable names should clearly distinguish between absolute and percentage based values. The following variables should be renamed:
    - `maxMCRIncrement -> maxMCRIncrementPercentage`
    - `maxMCRFloorIncrement -> maxMCRFloorIncrementPercentage`
    - `mcrFloorIncrementThreshold -> mcrFloorIncrementThresholdRatio`
5. MCR.sol#L154, #L169, #L209, #L213, #L217: Calculations using ratios and percentages were designed to have 4 decimal precision, by multiplying and dividing by 10 000. To improve the readability of the code the decimal precision should be defined as a constant and used throughout the contract.
6. MCR.sol#L94, #L143, #L147, #L151, #L175, #L199, #L205 : Use of `now` In Solidity version 0.7.0, the `now` keyword was deprecated. Developers are encouraged to use `block.timestamp` instead to ensure forward compatibility.

Update

1. This will remain as-is. In future, a similar pattern may be used for currencies with different numbers of decimal places.
2. Fixed in 59cef2d.
3. Fixed in b5be56b.
4. A comment explaining that these functions are expressed in basis points was added in 7103df9, while the variable names remained unchanged.
5. This constant was added in 7103df9.
6. Fixed in 7103df9.

Gas optimizations

SwapOperator.sol#L150: The swap functions make use of the `pool.getAssetDetails()` function to obtain information about the asset being traded, however, it discards the balance returned. The internal function then explicitly queries the balance again, after the assets from the pool have been transferred to the `SwapOperator`. This second balance query is unnecessary, as it would be functionally identical to first transfer the assets (SwapOperator.sol#L84), and then query (#L72) the asset details and reuse the balance (#L150). This will require adding the balance to the `AssetData` struct, but since it is only used in memory it should still result in gas cost savings.

Update

In b3ced4a, the `getAssetDetails()` function was altered to no longer return `balance`.

Fix spelling, grammar and naming convention errors

Spelling and grammar mistakes and contraventions of Solidity naming conventions were identified in the codebase. Fixing these mistakes can help improve the end-user experience by providing clear information on errors encountered, and improve the maintainability and auditability of the codebase.

1. Incidents.sol#L266: "depeged" -> "depegged"

Update

1. Fixed in ebd9e6f.

Improve comment accuracy

The following actions could be taken to improve the accuracy of code comments:

1. Incidents.sol#L163-164: The require statement messages are incorrect and should be negated, i.e.
    - "Incidents: Cover start date is before the incident" -> "Incidents: Cover start date is *after* the incident"
    - "Incidents: Cover end date is after the incident" -> "Incidents: Cover end date is *before* the incident"
2. Incidents.sol#L182: The comment on the payout amount calculation does not match the calculation it describes and should be changed to:

```
// coveredTokenAmount * coverAmount / maxAmount
```

Update

1. Fixed in ebd9e6f.
2. Fixed in ebd9e6f.

ABOUT
CONTACT US
SMART CONTRACT AUDITING
PENETRATION TESTING
AUDIT REPORTS
PRIVACY POLICY
TERMS OF SERVICE