Security Review Report
NM-0227 Range

NETHERMIND
SECURITY

(May 09, 2024)

Contents

1 Executive Summary

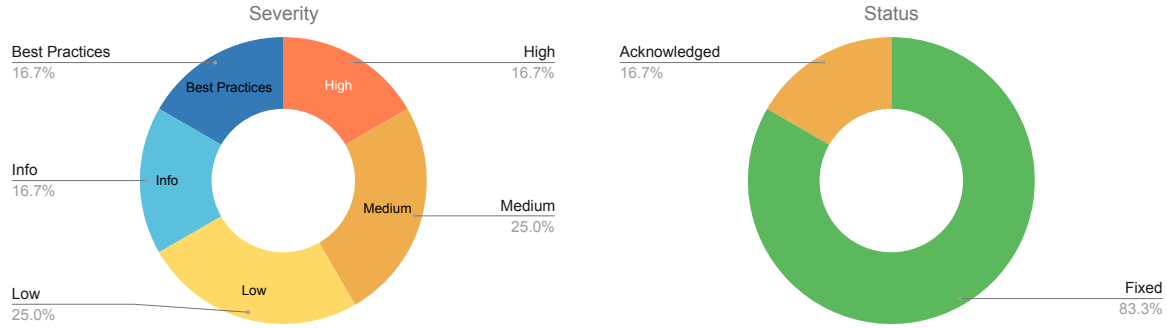
This document presents the security review conducted by Nethermind Security for Mangrove vault developed by the Range team. The contracts introduce a vault integrated with the Kandel strategy on Mangrove. Kandel is an Automated Market Making strategy that uses on-chain order flow to repost offers instantly.

The vaults factory, `RangeProtocolMangroveFactory`, facilitates deployment and upgrades of vaults by a whitelisted deployer, specifying the underlying tokens and associated Kandel strategy for each vault. Within the vault contract, `RangeProtocolMangroveVault`, two user types are defined: vault users and the vault manager. Users engage in minting and burning shares, while the manager handles position openings and liquidity movements to and from the Kandel strategy.

An additional commit hash 0495e8dff8390e1f211c81cd608544b1b0041017 was audited. The Range team implemented a swap functionality within the vault in this update. This feature allows the vault manager to perform swaps between the underlying tokens, enabling the re-balancing of the vault. Notably, the manager's balance is excluded from these swap operations.

The audited code comprises 587 lines of code in Solidity. The `Range` team has provided the necessary explanations to assist the audit. The audit was performed using: (a) manual analysis of the codebase, (b) automated analysis tools, and (c) simulation of the smart contracts. **Along this document, we report** 12 points of attention, two are classified as `High`, three are classified as `Medium`, three are classified as `Low` and four are classified as `Informational` or `Best Practices`. The issues are summarized in Fig. 1.

This document is organized as follows. Section 2 presents the files in the scope of this audit. Section 3 summarizes the issues. Section 4 presents the system overview. Section 5 discusses the risk rating methodology adopted for this audit. Section 6 details the issues. Section 7 discusses the documentation provided by the client for this audit. Section 8 presents the compilation, tests, and automated tests. Section 9 concludes the document.



(a) distribution of issues according to the severity

(b) distribution of issues according to the status

Fig 1: (a) Distribution of issues: Critical (0), **High** (2), **Medium** (3), **Low** (3), **Undetermined** (0), **Informational** (2), **Best Practices** (2). **(b) Distribution of status: Fixed** (10), **Acknowledged** (2), **Mitigated** (0), **Unresolved** (0)

Summary of the Audit

Audit Type	Security Review
Initial Report	May 09, 2024
Final Report	May 09, 2024
Methods	Manual Review, Automated Analysis
Repository	mangrove-vault
Commit Hash	6381216679fda0a69754eaf37bd293394029a284
Commit Hash	b43ee035b2409350991c94d09cbf3cf9fe4a1d14
Documentation	Comments within the code
Documentation Assessment	Low
Test Suite Assessment	Medium

2 Audited Files

	Contract	LoC	Comments	Ratio	Blank	Total
1	src/RangeProtocolMangroveFactory.sol	91	54	59.3%	23	168
2	src/RangeProtocolMangroveVault.sol	343	141	41.1%	62	546
3	src/RangeProtocolMangroveVaultStorage.sol	58	5	8.6%	15	78
4	src/interfaces/IRangeProtocolMangroveFactory.sol	24	1	4.2%	5	30
5	src/interfaces/IRangeProtocolMangroveVault.sol	71	1	1.4%	7	79
	Total	**587**	**202**	**34.4%**	**112**	**901**

3 Summary of Issues

	Finding	Severity	Update
1	Manager balance is substracted twice during the burning process	High	Fixed
2	Wrong accounting of current balance when a position is open	High	Fixed
3	Burning shares is restricted if offers fall below the density threshold	Medium	Acknowledged
4	Centralization risks in the `swap(...)` function	Medium	Acknowledged
5	Initial position parameters are not set on Kandel	Medium	Fixed
6	Partial funds transfer to Kandel results in share price manipulation	Low	Fixed
7	Possible share price fluctuation due to the reuse of `createPosition` function	Low	Fixed
8	Wrong balances calculation within the `_update(...)` function	Low	Fixed
9	Unchecked returned value of ERC20 `transfer(...)` and `transferFrom(...)` functions	Info	Fixed
10	Redundant computation of `baseQuoteTickIndex0` on every mint and burn call	Info	Fixed
11	A frequently used number literal `10_000` could be made constant	Best Practices	Fixed
12	Wrong NatSpec comment	Best Practices	Fixed

4 System Overview

The system implements a vault integrated with the Kandel strategy on Mangrove, an Automated Market Making strategy using on-chain order flow for instantaneous offer reposting. We distinguish between two main components: the vaults factory, `RangeProtocolMangroveFactory`, and the vault contract, `RangeProtocolMangroveVault`.

- The Vault Factory: The `RangeProtocolMangroveFactory` contract manages different vaults' deployment and upgrade processes. It defines a single whitelisted deployer address that can interact with the contract. When creating a vault, the deployer specifies parameters such as the underlying token pair and the associated Kandel strategy address.
- The Vault contract: The `RangeProtocolMangroveVault` contract defines two types of users: vault users and the vault manager. Users can mint and burn shares, representing their ownership in the vault, while the manager is responsible for initiating positions on Kandel and managing liquidity movements. The flow involves users depositing underlying tokens to mint shares or redeeming shares for equivalent tokens, with funds routed to or from Kandel when a position is active.

The subsequent sections detail the various steps of the flow within the vault, covering position management, minting and burning processes, and liquidity movement.

4.1 Initial position creation

Upon deployment, the manager calls the `createPosition(...)` function, which opens the initial position, allowing users to start minting shares and utilizing the vault.

```
function createPosition(Position calldata position_) external payable override onlyManager
```

The manager provides a structure containing the position parameters, such as the number of price points, the index of the first ask, the step size, the `midTick`, and the tick offset. These parameters help specify the price range on Kandel. Additionally, the structure includes parameters required to create offers on Mangrove, such as the gas required to execute the offer (`gasreq`) and the gas price.

```
struct Position {
    uint32 gasprice;
    uint24 gasreq;
    uint32 stepSize;
    uint32 pricePoints;
    uint256 baseQuoteTickOffset;
    uint256 firstAskIndex;
    Tick midTick;
}
```

4.2 Minting shares

Users can mint vault shares by calling the `mint(...)` function, specifying the number of shares to mint (`mintAmount`) and the maximum amount of underlying tokens they are willing to spend (`maxAmountsIn`).

```
function mint(uint256 mintAmount, uint256[2] calldata maxAmountsIn)
```

This process involves users depositing a ratio of underlying vault tokens (`token0` and `token1`) and receiving an equivalent amount of vault shares. If a position is active on Kandel, deposited funds are transferred to Kandel to populate position offers. The conversion from shares to tokens is performed based on the ratio between the current balance of the contract and the total supply, where the current balance depends on whether a position is open or not. When a position is open, the system considers the Kandel balance of underlying tokens. Conversely, the vault balance is used instead if no position is open. The transaction reverts with a slippage error if the underlying tokens amount to be transferred exceeds the provided maximum (`maxAmountsIn`).

4.3 Redeeming tokens

Shareholders can burn their shares to redeem underlying tokens by calling the `burn(...)` function and specifying the amount to be burned (`burnAmount`) as well as the minimum amount of underlying tokens they want to receive (`minAmountsOut`).

```
function burn( uint256 burnAmount, uint256[2] calldata minAmountsOut)
```

If a position is open, tokens are withdrawn from Kandle, and offers are updated accordingly. Otherwise, tokens are transferred from the vault directly. If the conversion from the `burnAmount` to the underlying tokens amounts falls below the provided minimum (`minAmountsOut`), the transaction reverts with a slippage error.

Upon burning shares, a management fee is deducted. The vault keeps track of accumulated fees, leaving them in the vault. The manager can claim the fees anytime by invoking the `collectManagerBalance(...)` function.

```
function collectManagerBalance() external override onlyManager
```

4.4 Liquidity movement

The vault enables the manager to add or remove liquidity from Kandel. For example, this functionality can be used if the manager wants to update the price points where liquidity is deployed based on the market conditions. For that, the contract provides two different functions: The `removeliquidity()` function allows the manager to close the current position by withdrawing all the funds from Kandel to the vault and retracting all the position offers.

```
function removeLiquidity() external override onlyManager
```

The `addLiquidity(...)` function allows injecting funds back into Kandel and creating a new position. The manager can specify the exact amount to be moved for each underlying token and the new position parameters.

```
function addLiquidity(Position calldata position_, uint256 token0Amount, uint256 token1Amount) external override
↪  onlyManager
```

4.5 Vault rebalancing

In the latest code update, a swapping functionality was introduced to the vault. This functionality enables the manager to execute swaps between the vault's underlying tokens using the `swap(...)` function to rebalance the vault.

```
function swap(address target, bytes calldata swapData, bool isToken0, uint256 amountIn) external override onlyManager
```

This function interacts with one of the whitelisted `SwapRouter` contracts to perform the swap. It takes the following parameters:

- `target`: Address of the swap router where the swap will be executed.
- `swapData`: Calldata used to invoke the swap function in the swapRouter contract, encoding swap parameters.
- `isToken0`: A boolean indicating whether token0 is being swapped out of the vault (`true`) or token1 (`false`).
- `amountIn`: The amount of tokens to be swapped out of the vault.

A whitelist of `SwapRouter` contracts is managed exclusively by the factory owner. The owner can add an address to the whitelist using the `whiteListSwapRouter(...)` function.

```
function whiteListSwapRouter(address swapRouter) external override
```

Similarly, the owner can remove an address from the whitelist using the `removeSwapRouterFromWhitelist(...)` function.

```
function removeSwapRouterFromWhitelist(address swapRouter) external override
```

5 Risk Rating Methodology

The risk rating methodology used by Nethermind Security follows the principles established by the OWASP Foundation. The severity of each finding is determined by two factors: **Likelihood** and **Impact**.

Likelihood measures how likely an attacker will uncover and exploit the finding. This factor will be one of the following values:

a) **High**: The issue is trivial to exploit and has no specific conditions that need to be met;

b) **Medium**: The issue is moderately complex and may have some conditions that need to be met;

c) **Low**: The issue is very complex and requires very specific conditions to be met.

When defining the likelihood of a finding, other factors are also considered. These can include but are not limited to Motive, opportunity, exploit accessibility, ease of discovery, and ease of exploit.

Impact is a measure of the damage that may be caused if an attacker exploits the finding. This factor will be one of the following values:

a) **High**: The issue can cause significant damage, such as loss of funds or the protocol entering an unrecoverable state;

b) **Medium**: The issue can cause moderate damage, such as impacts that only affect a small group of users or only a particular part of the protocol;

c) **Low**: The issue can cause little to no damage, such as bugs that are easily recoverable or cause unexpected interactions that cause minor inconveniences.

When defining the impact of a finding, other factors are also considered. These can include but are not limited to Data/state integrity, loss of availability, financial loss, and reputation damage. After defining the likelihood and impact of an issue, the severity can be determined according to the table below.

		Severity Risk		
Impact	**High**	Medium	High	Critical
	Medium	Low	Medium	High
	Low	Info/Best Practices	Low	Medium
	Undetermined	Undetermined	Undetermined	Undetermined
		Low	**Medium**	**High**
		Likelihood		

To address issues that do not fit a High/Medium/Low severity, Nethermind Security also uses three more finding severities: **Informational**, **Best Practices**, and **Undetermined**.

a) **Informational** findings do not pose any risk to the application, but they carry some information that the audit team intends to pass to the client formally;

b) **Best Practice** findings are used when some piece of code does not conform with smart contract development best practices;

c) **Undetermined** findings are used when we cannot predict the impact or likelihood of the issue.

6 Issues

6.1 [High] Manager balance is substracted twice during the burning process

File(s): RangeProtocolMangroveVault.sol

Description: The `burn(...)` function is utilized by users to burn their Vault tokens and receive `token0` and `token1` in return. When calculating the amounts to be returned, the current balance of the contract is utilized and the manager balance is subtracted.

The `getUnderlyingBalances(...)` function is used to retrieve the current underlying balances of the contract, excluding the Manager funds. However, the Manager balance is subtracted again from the returned amount by this function in the `burn(...)` function, resulting in the removal of these funds twice.

```
function burn(...) external override nonReentrant returns (uint256 amount0, uint256 amount1) {
   // ...
   // @audit The manager balance is deduced from the current balance
    (uint256 amount0Current, uint256 amount1Current) = getUnderlyingBalances();

    // @audit The manager balance is subtracted again from the current balance
    amount0Current -= _managerBalance0;
    amount1Current -= _managerBalance1;
    // ...
}
```

This double subtraction leads to incorrect computations when burning the vault tokens, causing users to receive less tokens than expected or the transaction to revert with an underflow error.

Recommendation(s): Consider removing the duplicated Manager balance subtraction within the `burn(...)` function.

Status: Fixed

Update from the client: Fix is applied in the commit 523203683cab875e479dfeb352b89b5e8e6421be

6.2 [High] Wrong accounting of current balance when a position is open

File(s): RangeProtocolMangroveVault.sol

Description: When a user burns their shares, the vault calculates the equivalent amounts of `token0` and `token1`, which will be withdrawn from the Kandel strategy to the vault through a call to the `withdrawFunds(...)` function. Following this withdrawal, management fees are computed and subtracted from the withdrawn amount, and the remaining tokens are transferred to the user. The manager's funds are left in the vault and accounted for within the `_managerBalance0` and `_managerBalance1` variables.

```
function burn(...) external override nonReentrant returns (uint256 amount0, uint256 amount1) {
    // ...
    // @audit Compute amounts to withdraw
    amount0 = amount0Current * burnAmount / vars.totalSupply;
    amount1 = amount1Current * burnAmount / vars.totalSupply;

    if (vars.inPosition) {
        // @audit Requested tokens are transferred to the vault
        vars.kandel.withdrawFunds(amount0, amount1, address(this));
        // ...
    }
    // ...
    // @audit The management fees are computed and the Manager balance is updated
    _applyManagingFee(amount0, amount1);
    (amount0, amount1) = _netManagingFee(amount0, amount1);
    // ...
    // @audit The withdrawn amount minus the management fees are transferred to the user
    vars.token0.transfer(msg.sender, amount0);
    vars.token1.transfer(msg.sender, amount1);

    emit Burned(msg.sender, burnAmount, amount0, amount1);
}
```

The `getUnderlyingBalances()` function is invoked within different functions to retrieve the current balance, it is utilized for converting shares to tokens and vice versa. When a position is currently open, this function returns the current Kandel balance of each token and subtracts the manager's balance from it.

```
function getUnderlyingBalances() public view override returns (uint256 amount0Current, uint256 amount1Current) {
    (amount0Current, amount1Current) = _inPosition
      ? (_token0.balanceOf(address(_kandel)), _token1.balanceOf(address(_kandel)))
      : (_token0.balanceOf(address(this)), _token1.balanceOf(address(this)));

    // @audit The manager balance is subtracted from the current balance regardless of the `_inPosition` flag
    amount0Current -= _managerBalance0;
    amount1Current -= _managerBalance1;
}
```

This issue results in incorrect accounting of the current balance and converting shares to tokens when a position is open. Since the manager's funds are already withdrawn to the vault but still subtracted from the Kandel balance, this could lead to underflow errors when fetching the current balance, particularly in cases where the manager balance exceeds the Kandel strategy balance.

Recommendation(s): Considering the current design of leaving the manager fees in the vault, update the `getUnderlyingBalances()` function to not subtract the manager balances when a position is open (`_inPosition` is `true`). This will ensure consistency in balance calculations when the manager's funds are already withdrawn. Additionally, to reinforce this assumption, implement a validation logic in the `addLiquidity(...)` function, disallowing the transfer of manager funds into the Kandel strategy.

Status: Fixed

Update from the client: Fix is applied in the commit b01c3f86df01e34f30def5e448c107c5e246519a

6.3 [Medium] Burning shares is restricted if offers fall below the density threshold

File(s): RangeProtocolMangroveVault.sol

Description: When a user executes the `burn(...)` function to burn a specific share amount, the vault withdraws the corresponding amounts of `token0` and `token1` from the Kandel strategy. Subsequently, offers in the position are populated using the updated contract balance; this involves reducing the offered volume.

However, in Mangrove, a parameter named `density` is defined, representing the minimum ratio of outbound tokens per gas unit for offers. In the case where updating offers results in falling below this threshold, the transaction reverts with a `mgv/writeOffer/density/tooLow` error.

```
function burn(...) external override nonReentrant returns (uint256 amount0, uint256 amount1) {
    // ...
    // @audit Compute amounts to withdraw
    amount0 = amount0Current * burnAmount / vars.totalSupply;
    amount1 = amount1Current * burnAmount / vars.totalSupply;

    if (vars.inPosition) {
        // @audit Requested tokens are withdrawn from Kandel
        vars.kandel.withdrawFunds(amount0, amount1, address(this));
        // ...
        // @audit This call will fail if the offers fall below density
        _kandel.populateChunkFromOffset(
            0,
            vars.position.pricePoints,
            baseQuoteTickIndex0,
            _firstAskIndex,
            vars.token1.balanceOf(address(vars.kandel)), // bidGives
            vars.token0.balanceOf(address(vars.kandel)) // askGives
        );
    }
    // ...
}
```

Consequently, some funds within the Kandel strategy will be locked, preventing users from burning their shares and withdrawing tokens until the position is closed.

Recommendation(s): Revisit the design to ensure users can always withdraw their tokens. This can be achieved by retracting offers if the requested amount would cause them to fall below the density threshold. Alternatively, maintain liquidity within the strategy to ensure that the minimal density of the position offers is always covered.

Status: Acknowledged

Update from the client: Fix is applied in commits 9752f196e14fbc70f735d0a16087b2cd8bb67ee0 and 1fa4d918dd6efe4a2a8f85473271bc0eb4f8e9e3

Update from Nethermind Security: The issue has been marked as *Acknowledged* since the provided commit does not include a fix for it. The newly introduced `RepopulationFailedInBurnCall` event will serve to notify Range team to take necessary actions.

6.4 [Medium] Centralization risks in the `swap(...)` function

File(s): RangeProtocolMangroveVault.sol

Description: The `swap(...)` function allows the vault manager to swap between the vault's underlying tokens to rebalance the vault. This function accepts `swapData` as input, which is later utilized to execute a swap operation on the Uniswap V3 `SwapRouter` contract. This data (`swapData`) encodes various parameters required for executing the swap, including the `recipient` address where the output tokens are directed. Consequently, the manager can deplete the vault tokens by setting the `recipient` address to their wallet. As a result, tokens are swapped, and the output token is redirected to the manager instead of remaining in the vault.

```
function swap(
    address target,
    bytes calldata swapData, // @audit swapData includes a `recipient` address
    bool isToken0,
    uint256 amountIn
)
    external
    override
    onlyManager
{
    if (!_whitelistedSwapRouters[target]) revert VaultErrors.SwapRouterIsNotWhitelisted();

    IERC20 token0_ = _token0;
    IERC20 token1_ = _token1;

    uint256 token0BalanceBefore = token0_.balanceOf(address(this));
    uint256 token1BalanceBefore = token1_.balanceOf(address(this));

    (isToken0 ? token0_ : token1_).approve(target, amountIn);

    // @audit Manager can route the output tokens to a different address,
    // by setting a different recipient within the `swapData`
    Address.functionCall(target, swapData);
    // ...
}
```

This design introduces a notable centralization risk within the protocol, as a malicious manager can drain the vault funds through two consecutive swap operations, one for each underlying token. Moreover, the manager can manipulate other parameters, such as the swap price.

Recommendation(s): To mitigate the centralization risk, it is advisable to reconsider the design and clearly communicate to users the privileges granted to the vault manager. Furthermore, potential measures to reduce centralization risks include using a Multisig wallet for the manager to decrease the likelihood of compromise and implementing validation for swap inputs where possible. Additionally, it is recommended to disallow swaps where zero tokens are returned to the vault.

Status: Acknowledged

Update from the client: We have added a check such that at least 1 unit of out token must be received by vault. The manager will be KYCed entity and they are expected to act responsibly. In the event of its compromise there will be a risk that manager can route the swap through an illiquid pool and arbitrage. This risk will be accepted as the limitation of the system until the factory owner removes the swap router address from whitelist.
commit hash: 62bed7086335ce31a0f91c16ac7845aea40c797d

6.5 [Medium] Initial position parameters are not set on Kandel

File(s): RangeProtocolMangroveVault.sol

Description: The `createPosition(...)` function is invoked after deployment to initialize the initial position. Subsequently, users can utilize the `mint(...)` function to mint vault shares and populate the offers on Kandel.

```
function createPosition(Position calldata position_) external payable override onlyManager {
    if (_inPosition) revert VaultErrors.PositionAlreadyCreated();
    _MGV.fund{ value: msg.value }(address(_kandel));
    // @audit parameters are not set on the Kandel instance
    _position = position_;
    _inPosition = true;
    emit PositionCreated(position_);
}
```

The `mint(...)` function deposits funds into Kandel by calling the `depositFunds(...)` function, followed by a call to `populateChunkFromOffset()`, passing the necessary parameters of the position. The documentation for this function indicates that it should be used after `populateFromOffset()` to populate offers with the same parameters set within that function.

However, in the current implementation, `populateChunkFromOffset()` is called first on Kandel, and there's no preceding call to set the position parameters on Kandel. These parameters are only stored locally in the vault's `_position` structure within the `createPosition(...)` function. This could lead to unexpected behavior and erroneous ticks computation since the Kandel Strategy uses parameters different from the ones currently set on the vault.

```
function mint(...) external override nonReentrant whenNotPaused returns (uint256 amount0, uint256 amount1) {
   // ...
   if (vars.inPosition) {
       // @audit depositing funds into Kandel
       vars.token0.approve((address(vars.kandel)), amount0);
       vars.token1.approve(address(vars.kandel), amount1);
       vars.kandel.depositFunds(amount0, amount1);
       // ...

       // @audit call to `populateChunkFromOffset` not preceded by setting parameters
       _kandel.populateChunkFromOffset(
           0,
           vars.position.pricePoints,
           baseQuoteTickIndex0,
           _firstAskIndex,
           vars.token1.balanceOf(address(vars.kandel)), // bidGives
           vars.token0.balanceOf(address(vars.kandel)) // askGives
       );
   }
}
```

Recommendation(s): Consider setting parameters on Kandel for the first position.

Status: Fixed

Update from the client: Fix is applied in the commit 9385c82a47ec09b1b454825d442ab329902de310

6.6 [Low] Partial funds transfer to Kandel results in share price manipulation

File(s): RangeProtocolMangroveVault.sol

Description: When a user invokes the mint function with the `inPosition` variable set to `false`, they transfer an `amount0` and `amount1` of outbound and inbound tokens, respectively, to the contract. These tokens remain within the contract until a position is created.

Subsequently, the manager calls the `addLiquidity(...)` function, which will move the funds from the vault to the Kandel Strategy, initializing a new position. However, this function allows the manager to specify the amount of tokens to transfer, potentially being less than the total vault balance.

In scenarios where only a fraction of the funds is moved to the Kandel Strategy, the `getUnderlyingBalances(...)` function reports the balance solely in Kandel, without accounting for remaining vault assets. Consequently, this results in a decrease in share price; Users who burn their shares after this step will get less tokens than deserved.

```
function getUnderlyingBalances() public view override returns (uint256 amount0Current, uint256 amount1Current) {
  (amount0Current, amount1Current) = _inPosition
      // @audit only funds moved to Kandel are considered
      ? (_token0.balanceOf(address(_kandel)), _token1.balanceOf(address(_kandel)))
      : (_token0.balanceOf(address(this)), _token1.balanceOf(address(this)));
      // ...
}
```

Recommendation(s): Consider enforcing moving all the users funds to Kandel, excluding the manager balance.

Status: Fixed

Update from the client: Fix is applied in commit b01c3f86df01e34f30def5e448c107c5e246519a

6.7 [Low] Possible share price fluctuation due to the reuse of `createPosition` function

File(s): RangeProtocolMangroveVault.sol

Description: The `createPosition(...)` function is intended to be invoked once after the vault's deployment when no funds have yet been deposited. It initializes a position within the vault, allowing users to begin utilizing its functionalities by minting and burning shares. Upon invocation, it sets the `_inPosition` boolean to `true` and populates the `_position` structure with the provided data.

However, the current implementation fails to enforce this assumption. The manager can call the function anytime after deployment as long as no position is already open. For instance, if the manager closes an existing position and subsequently calls `createPosition()`, all user funds remain inaccessible within the vault. Moreover, it initiates an empty position, resetting the share price to its initial value.

```
function createPosition(Position calldata position_) external payable override onlyManager {
    // @audit A position is created and no funds are moved to Kandel
    if (_inPosition) revert VaultErrors.PositionAlreadyCreated();
    _MGV.fund{ value: msg.value }(address(_kandel));
    _position = position_;
    _inPosition = true;

    emit PositionCreated(position_);
}
```

Given the function is restricted to the vault manager only, the likelihood of this unintended behaviour happening is low. However, it increases the centralization risks within the system.

Recommendation(s): Consider updating the function implementation to enforce the assumption that it is only called once post-deployment.

Status: Fixed

Update from the client: Fix is applied in commit b01c3f86df01e34f30def5e448c107c5e246519a

6.8 [Low] Wrong balances calculation within the `_update(...)` function

File(s): RangeProtocolMangroveVault.sol

Description: The `_update(...)` function serves to update the `_userVaults` mapping upon any token transfer. The mapping is responsible for tracking deposited amounts of underlying tokens per user.

However, the current implementation of `_update(...)` has an issue in how it computes sender balances to update the `_userVaults` mapping. Upon invocation, the function initiates the token transfer by calling `super._update(...)`, which will update both the sender and receiver balances. Subsequently, it computed the sender's balance before the transfer by taking his current balance, assuming the transfer hasn't yet occurred. As a result, the sender's balance before and after the transfer is inaccurately computed, leading to incorrect updates of the `_userVaults`.

```
function _update(address from, address to, uint256 value) internal override {
    // @audit Transfer of tokens and update of balances
    super._update(from, to, value);
    // ...

    // @audit The current balance of the sender is already updated
    uint256 senderBalanceBefore = balanceOf(from);
    uint256 senderBalanceAfter = senderBalanceBefore - value;

    // @audit The `_userVaults` is updated with incorrect balances
    uint256 token0Amount =
        _userVaults[from].token0 - (_userVaults[from].token0 * senderBalanceAfter) / senderBalanceBefore;

    uint256 token1Amount =
        _userVaults[from].token1 - (_userVaults[from].token1 * senderBalanceAfter) / senderBalanceBefore;

    _userVaults[from].token0 -= token0Amount;
    _userVaults[from].token1 -= token1Amount;

    _userVaults[to].token0 += token0Amount;
    _userVaults[to].token1 += token1Amount;
}
```

Recommendation(s): Update the logic considering the transfer has already been executed. One solution is to update the computation of the sender's balances as follows:

```
uint256 senderBalanceAfter = balanceOf(from);
uint256 senderBalanceBefore = senderBalanceAfter + value;
```

Alternatively, consider moving the `super._update(...)` call after storing the sender's balance before the transfer, ensuring accurate calculations.

Status: Fixed

Update from the client: Fix is applied in commit b075ac14c04e42aef184f51c063af8f84db122d2

6.9 [Info] Unchecked returned value of ERC20 `transfer(...)` and `transferFrom(...)` functions

File(s): `RangeProtocolMangroveVault.sol`

Description: The `mint(...)`, `burn(...)` and `collectManagerBalance(...)` functions utilize the ERC20 `transfer(...)` and `transferFrom(...)` functions to handle transfers of `token0` and `token1` between users and the vault. These functions return a boolean value indicating whether the transfer was successful. However, the current implementation doesn't check the returned value, potentially resulting in unexpected behavior, particularly for tokens that do not revert on failure. Consequently, users may mint or burn shares without the ERC20 token transfer succeeding as intended.

Recommendation(s): Consider using the `safeTransfer(...)` and `safeTransferFrom(...)` from OpenZeppelin `SafeERC20` library. Alternatively, implement a validation for the returned value from these functions, ensuring the transfer was successful.

Status: Fixed

Update from the client: Fix is applied in commit 1a8020fbc85bb4511c8cbe1272390222d48af62c

6.10 [Info] Redundant computation of `baseQuoteTickIndex0` on every mint and burn call

File(s): `RangeProtocolMangroveVault.sol`

Description: The `baseQuoteTickIndex0` is currently computed within each call to the `mint(...)` and `burn(...)` functions. This computation involves the position parameters, which remain unchanged within the same position.

```
// @audit Computation can be done only once per position
Tick baseQuoteTickIndex0 = Tick.wrap(
    (Tick.unwrap(vars.position.midTick))
        - int256(vars.position.baseQuoteTickOffset * vars.position.firstAskIndex)
        + int256(vars.position.baseQuoteTickOffset) / 2
);
```

This results in unnecessary gas overhead for both mint and burn operations, as the value can be computed once per position.

Recommendation(s): It is recommended to eliminate the redundant computation of `baseQuoteTickIndex0` and perform it only once for each new position, storing it in the contract.

Status: Fixed

Update from the client: Fix is applied in commit a9b8841fe5de1f3b1450fe41fed0ed628652ff7f

Update from Nethermind Security: The provided commit fixes the issue. Note that the current solution relies on the manager providing the tick computed in a correct way instead of performing the computation on-chain.

Update from the client: Fix is applied in commits: 1fa4d918dd6efe4a2a8f85473271bc0eb4f8e9e3 and 88a927105e6cde72c99d8c8392807d679dc4ee13.

6.11 [Best Practices] A frequently used number literal `10_000` could be made constant

File(s): `RangeProtocolMangroveVault.sol`

Description: The functions `_netManagingFee(...)` and `_applyManagingFee(...)` calculate and apply a management fee to the amounts of two different tokens. In both, the integer literal `10_000` expresses the management fee as a percentage, representing 100% in basis points. Where applicable, it is considered a best practice to use named constants instead of number literals to improve the codebase's readability and give more context to the reader.

Recommendation(s): Consider refactoring the code to use a named constant with a meaningful name like `ONE_HUNDRED_PERCENT`.

Status: Fixed

Update from the client: Fix is applied in commit c102e3f09ca1139dfdadc4a8d420c24d41660622

6.12 [Best Practices] Wrong NatSpec comment

File(s): `RangeProtocolMangroveVault.sol`

Description: The NatSpec documentation of the `pause()` and `unpause()` functions states that the burning feature is also paused. However, the current implementation allows burning when the contract is paused.

Recommendation(s): Consider updating the NatSpec documentation of `pause()` and `unpause()` functions to reflect the actual implementation.

Status: Fixed

Update from the client: Fix is applied in the commit d960075e8292b61ebd17b08cfde88fda68d9048f.

7 Documentation Evaluation

Software documentation refers to the written or visual information describing software's functionality, architecture, design, and implementation. It provides a comprehensive overview of the software system and helps users, developers, and stakeholders understand how the software works, how to use it, and how to maintain it. Software documentation can take different forms, such as user manuals, system manuals, technical specifications, requirements documents, design documents, and code comments. Software documentation is critical in software development, enabling effective communication between developers, testers, users, and other stakeholders. It helps to ensure that everyone involved in the development process has a shared understanding of the software system and its functionality. Moreover, software documentation can improve software maintenance by providing a clear and complete understanding of the software system, making it easier for developers to maintain, modify, and update the software over time. Smart contracts can use various types of software documentation. Some of the most common types include:

- Technical whitepaper: A technical whitepaper is a comprehensive document describing the smart contract's design and technical details. It includes information about the purpose of the contract, its architecture, its components, and how they interact with each other;
- User manual: A user manual is a document that provides information about how to use the smart contract. It includes step-by-step instructions on how to perform various tasks and explains the different features and functionalities of the contract;
- Code documentation: Code documentation is a document that provides details about the code of the smart contract. It includes information about the functions, variables, and classes used in the code, as well as explanations of how they work;
- API documentation: API documentation is a document that provides information about the API (Application Programming Interface) of the smart contract. It includes details about the methods, parameters, and responses that can be used to interact with the contract;
- Testing documentation: Testing documentation is a document that provides information about how the smart contract was tested. It includes details about the test cases that were used, the results of the tests, and any issues that were identified during testing;
- Audit documentation: Audit documentation includes reports, notes, and other materials related to the security audit of the smart contract. This type of documentation is critical in ensuring that the smart contract is secure and free from vulnerabilities.

These types of documentation are essential for smart contract development and maintenance. They help ensure that the contract is properly designed, implemented, and tested, and they provide a reference for developers who need to modify or maintain the contract in the future.

Remarks about Range documentation

The Range contracts documentation is presented through inline comments and NatSpec documentation within the code, providing explanations and insights about their implementation and design choices. Additionally, the Range team was available to address any inquiries or concerns raised by the Nethermind Security team.

8 Test Suite Evaluation

8.1 Contracts Compilation

```
forge build
[] Compiling...
[] Compiling 59 files with 0.8.20
[] Solc 0.8.20 finished in 10.56s
Compiler run successful with warnings:
Warning (3420): Source file does not specify required compiler version! Consider adding "pragma solidity ^0.8.20;"
--> test/RangeProtocolMangroveFactory.t.sol
```

8.2 Tests Output

```
forge test
[] Compiling...
No files changed, compilation skipped

Ran 25 tests for test/RangeProtocolMangroveVault.t.sol:RangeProtocolMangroveVaultTest
[PASS] testAddLiquidity() (gas: 2033776)
[PASS] testAddLiquidityByNonManager() (gas: 1723585)
[PASS] testBurn() (gas: 1930955)
[PASS] testBurnWithExceededSlippage() (gas: 2476223)
[PASS] testBurnWithExtraBalance() (gas: 39001)
[PASS] testBurnWithZeroAmount() (gas: 19003)
[PASS] testCollectManagerBalanceByNonManager() (gas: 15983)
[PASS] testCreatePositionByNonManager() (gas: 24809)
[PASS] testDeployment() (gas: 46697)
[PASS] testMintWhenVaultIsPaused() (gas: 45902)
[PASS] testMintWithExceededSlippage() (gas: 69278)
[PASS] testMintWithNonZeroTotalSupply() (gas: 1916379)
[PASS] testMintWithZeroAmount() (gas: 21256)
[PASS] testMintWithZeroTotalSupply() (gas: 1640198)
[PASS] testPause() (gas: 40259)
[PASS] testPauseByNonManager() (gas: 15985)
[PASS] testReInitialize() (gas: 16322)
[PASS] testRemoveLiquidity() (gas: 1724196)
[PASS] testRemoveLiquidityByNonManager() (gas: 1602323)
[PASS] testSetManagingFee() (gas: 24476)
[PASS] testSetManagingFeeAboveMax() (gas: 15573)
[PASS] testSetManagingFeeByNonManager() (gas: 15983)
[PASS] testUnpause() (gas: 31723)
[PASS] testUnpauseByNonManager() (gas: 15985)
[PASS] testWithdrawFromMangroveByNonManager() (gas: 16119)
Suite result: ok. 25 passed; 0 failed; 0 skipped; finished in 12.68s (33.28s CPU time)

Ran 1 test suite in 12.83s (12.68s CPU time): 25 tests passed, 0 failed, 0 skipped (25 total tests)
```

9 About Nethermind

Nethermind is a Blockchain Research and Software Engineering company. Our work touches every part of the web3 ecosystem - from layer 1 and layer 2 engineering, cryptography research, and security to application-layer protocol development. We offer strategic support to our institutional and enterprise partners across the blockchain, digital assets, and DeFi sectors, guiding them through all stages of the research and development process, from initial concepts to successful implementation.

We offer security audits of projects built on EVM-compatible chains and Starknet. We are active builders of the Starknet ecosystem, delivering a node implementation, a block explorer, a Solidity-to-Cairo transpiler, and formal verification tooling. Nethermind also provides strategic support to our institutional and enterprise partners in blockchain, digital assets, and decentralized finance (DeFi). In the next paragraphs, we introduce the company in more detail.

Blockchain Security: At Nethermind, we believe security is vital to the health and longevity of the entire Web3 ecosystem. We provide security services related to Smart Contract Audits, Formal Verification, and Real-Time Monitoring. Our Security Team comprises blockchain security experts in each field, often collaborating to produce comprehensive and robust security solutions. The team has a strong academic background, can apply state-of-the-art techniques, and is experienced in analyzing cutting-edge Solidity and Cairo smart contracts, such as ArgentX and StarkGate (the bridge connecting Ethereum and StarkNet). Most team members hold a Ph.D. degree and actively participate in the research community, accounting for 240+ articles published and 1,450+ citations in Google Scholar. The security team adopts customer-oriented and interactive processes where clients are involved in all stages of the work.

Blockchain Core Development: Our core engineering team, consisting of over 20 developers, maintains, improves, and upgrades our flagship product - the Nethermind Ethereum Execution Client. The client has been successfully operating for several years, supporting both the Ethereum Mainnet and its testnets, and now accounts for nearly a quarter of all synced Mainnet nodes. Our unwavering commitment to Ethereum's growth and stability extends to sidechains and layer 2 solutions. Notably, we were the sole execution layer client to facilitate Gnosis Chain's Merge, transitioning from Aura to Proof of Stake (PoS), and we are actively developing a full-node client to bolster Starknet's decentralization efforts. Our core team equips partners with tools for seamless node set-up, using generated docker-compose scripts tailored to their chosen execution client and preferred configurations for various network types.

DevOps and Infrastructure Management: Our infrastructure team ensures our partners' systems operate securely, reliably, and efficiently. We provide infrastructure design, deployment, monitoring, maintenance, and troubleshooting support, allowing you to focus on your core business operations. Boasting extensive expertise in Blockchain as a Service, private blockchain implementations, and node management, our infrastructure and DevOps engineers are proficient with major cloud solution providers and can host applications in-house or on clients' premises. Our global in-house SRE teams offer 24/7 monitoring and alerts for both infrastructure and application levels. We manage over 5,000 public and private validators and maintain nodes on major public blockchains such as Polygon, Gnosis, Solana, Cosmos, Near, Avalanche, Polkadot, Aptos, and StarkWare L2. Sedge is an open-source tool developed by our infrastructure experts, designed to simplify the complex process of setting up a proof-of-stake (PoS) network or chain validator. Sedge generates docker-compose scripts for the entire validator set-up based on the chosen client, making the process easier and quicker while following best practices to avoid downtime and being slashed.

Cryptography Research: At Nethermind, our Cryptography Research team is dedicated to continuous internal research while fostering close collaboration with external partners. The team has expertise across a wide range of domains, including cryptography protocols, consensus design, decentralized identity, verifiable credentials, Sybil resistance, oracles, and credentials, distributed validator technology (DVT), and Zero-knowledge proofs. This diverse skill set, combined with strong collaboration between our engineering teams, enables us to deliver cutting-edge solutions to our partners and clients.

Smart Contract Development & DeFi Research: Our smart contract development and DeFi research team comprises 40+ world-class engineers who collaborate closely with partners to identify needs and work on value-adding projects. The team specializes in Solidity and Cairo development, architecture design, and DeFi solutions, including DEXs, AMMs, structured products, derivatives, and money market protocols, as well as ERC20, 721, and 1155 token design. Our research and data analytics focuses on three key areas: technical due diligence, market research, and DeFi research. Utilizing a data-driven approach, we offer in-depth insights and outlooks on various industry themes.

Our suite of L2 tooling: Warp is Starknet's approach to EVM compatibility. It allows developers to take their Solidity smart contracts and transpile them to Cairo, Starknet's smart contract language. In the short time since its inception, the project has accomplished many achievements, including successfully transpiling Uniswap v3 onto Starknet using Warp.

- **Voyager** is a user-friendly Starknet block explorer that offers comprehensive insights into the Starknet network. With its intuitive interface and powerful features, Voyager allows users to easily search for and examine transactions, addresses, and contract details. As an essential tool for navigating the Starknet ecosystem, Voyager is the go-to solution for users seeking in-depth information and analysis;
- **Horus** is an open-source formal verification tool for StarkNet smart contracts. It simplifies the process of formally verifying Starknet smart contracts, allowing developers to express various assertions about the behavior of their code using a simple assertion language;
- **Juno** is a full-node client implementation for Starknet, drawing on the expertise gained from developing the Nethermind Client. Written in Golang and open-sourced from the outset, Juno verifies the validity of the data received from Starknet by comparing it to proofs retrieved from Ethereum, thus maintaining the integrity and security of the entire ecosystem.

Learn more about us at nethermind.io.

General Advisory to Clients

As auditors, we recommend that any changes or updates made to the audited codebase undergo a re-audit or security review to address potential vulnerabilities or risks introduced by the modifications. By conducting a re-audit or security review of the modified codebase, you can significantly enhance the overall security of your system and reduce the likelihood of exploitation. However, we do not possess the authority or right to impose obligations or restrictions on our clients regarding codebase updates, modifications, or subsequent audits. Accordingly, the decision to seek a re-audit or security review lies solely with you.

Disclaimer

This report is based on the scope of materials and documentation provided by you to Nethermind in order that Nethermind could conduct the security review outlined in **1. Executive Summary** and **2. Audited Files**. The results set out in this report may not be complete nor inclusive of all vulnerabilities. Nethermind has provided the review and this report on an as-is, where-is, and as-available basis. You agree that your access and/or use, including but not limited to any associated services, products, protocols, platforms, content, and materials, will be at your sole risk. Blockchain technology remains under development and is subject to unknown risks and flaws. The review does not extend to the compiler layer, or any other areas beyond the programming language, or other programming aspects that could present security risks. This report does not indicate the endorsement of any particular project or team, nor guarantee its security. No third party should rely on this report in any way, including for the purpose of making any decisions to buy or sell a product, service or any other asset. To the fullest extent permitted by law, Nethermind disclaims any liability in connection with this report, its content, and any related services and products and your use thereof, including, without limitation, the implied warranties of merchantability, fitness for a particular purpose, and non-infringement. Nethermind does not warrant, endorse, guarantee, or assume responsibility for any product or service advertised or offered by a third party through the product, 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, any hyperlinked websites, any websites or mobile applications appearing on any advertising, and Nethermind will not be a party to or in any way be responsible for monitoring any transaction between you and any third-party providers of products or services. As with the purchase or use of a product or service through any medium or in any environment, you should use your best judgment and exercise caution where appropriate. FOR AVOIDANCE OF DOUBT, THE REPORT, ITS CONTENT, ACCESS, AND/OR USAGE THEREOF, INCLUDING ANY ASSOCIATED SERVICES OR MATERIALS, SHALL NOT BE CONSIDERED OR RELIED UPON AS ANY FORM OF FINANCIAL, INVESTMENT, TAX, LEGAL, REGULATORY, OR OTHER ADVICE.