# Security Review Report
# NM-0339 Mangrove Vault

(Oct 28, 2024)

# Contents

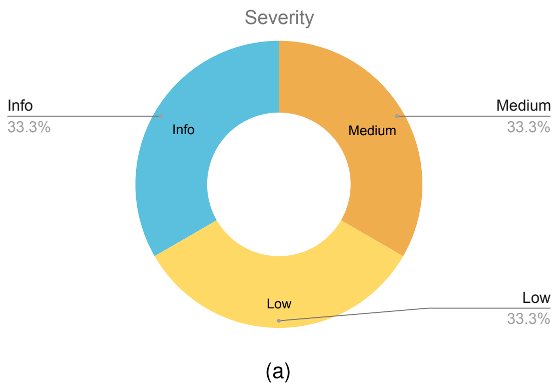
# 1 Executive Summary

This document presents the security review performed by Nethermind Security for Mangrove vault. The Mangrove team has developed the `MangroveVault` contract, a vault built on top of Mangrove's Kandel strategy. This vault manages two underlying assets: a base and a quote token, and it serves two types of actors: regular users and the contract owner. Users can mint vault shares by depositing the required amounts of underlying tokens. These tokens are subsequently allocated to the Kandel strategy managed by the vault, which implements an automated market-making strategy on Mangrove. Shareholders can burn their shares in exchange for an equivalent amount of underlying tokens.

The vault owner is responsible for setting and updating the Kandel parameters, as well as adjusting other vault parameters, including fee percentages and the state of the funds within the vault.

**The audit was performed using** (a) manual analysis of the codebase, (b) automated analysis tools, (c) simulation of the smart contract, and (d) creation of test cases. **Along this document, we report** three points of attention, where one is classified as `Medium`, one are classified as `Low`, and one is classified as `Informational` . The issues are summarized in Fig. 1.

**This document is organized as follows.** Section 2 presents the files in the scope. Section 3 summarizes the issues. Section 4 presents the system overview. Section 5 discusses the risk rating methodology. 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.



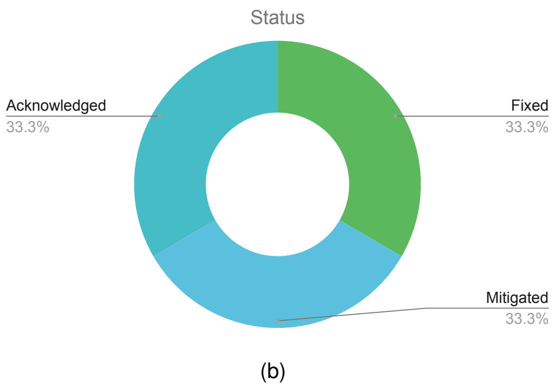


**Fig. 1: Distribution of issues: Critical** (0), **High** (0), **Medium** (1), **Low** (1), **Undetermined** (0), **Informational** (1), **Best Practices** (0).
**Distribution of status: Fixed** (1), **Acknowledged** (1), **Mitigated** (1), **Unresolved** (0)

### Summary of the Audit

| | |
|---|---|
| **Audit Type** | Security Review |
| **Initial Report** | Oct 25, 2024 |
| **Response from Client** | Regular responses during audit engagement |
| **Final Report** | Oct 28, 2024 |
| **Repository** | mangrove-vault |
| **Commit (Audit)** | d03e9731b0da0019d89467f418775e4a4cbd231c |
| **Commit (Final)** | 2d00ae71c90bda06f7026ae7c8a91283159383ca |
| **Documentation Assessment** | High |
| **Test Suite Assessment** | Medium |

## 2 Audited Files

| | Contract | LoC | Comments | Ratio | Blank | Total |
|---|---|---|---|---|---|---|
| 1 | MangroveVault.sol | 537 | 400 | 74.5% | 123 | 1060 |
| 2 | MangroveVaultFactory.sol | 21 | 1 | 4.8% | 3 | 25 |
| 3 | oracles/IOracle.sol | 6 | 10 | 166.7% | 2 | 18 |
| 4 | oracles/chainlink/MangroveChainlinkOracleFactory.sol | 35 | 24 | 68.6% | 4 | 63 |
| 5 | oracles/chainlink/MangroveChainlinkOracle.sol | 67 | 51 | 76.1% | 12 | 130 |
| 6 | lib/MangroveVaultEvents.sol | 43 | 84 | 195.3% | 13 | 140 |
| 7 | lib/MangroveVaultErrors.sol | 14 | 61 | 435.7% | 11 | 86 |
| 8 | lib/GeometricKandelExtra.sol | 86 | 54 | 62.8% | 12 | 152 |
| 9 | lib/MangroveVaultConstants.sol | 8 | 6 | 75.0% | 3 | 17 |
| 10 | lib/DistributionLib.sol | 136 | 72 | 52.9% | 16 | 224 |
| 11 | lib/MangroveLib.sol | 17 | 1 | 5.9% | 4 | 22 |
| | **Total** | **970** | **764** | **78.8%** | **203** | **1937** |

## 3 Summary of Issues

| | Finding | Severity | Update |
|---|---|---|---|
| 1 | Missing check for stale data from Chainlink oracle | Medium | Acknowledged |
| 2 | The function `_swap` allows arbitrary calls and is underconstrained | Low | Mitigated |
| 3 | Use of `transfer` instead of low level call to send native assets | Info | Fixed |

## 4 System Overview

The Mangrove vault consists of the main `MangroveVault` contract, which implements essential functionalities for both users and the contract owner. Each vault holds two underlying assets: a base token and a quote token, and it is associated with its own Kandel strategy. The vault interacts with several external dependencies, as illustrated in the figure below:

- `MangroveVaultFactory`: Factory contract allowing the creation of `MangroveVault` instances.
- `MangroveChainLinkOracle`: An oracle adapter that connects to up to four Chainlink price feeds to provide the current price of the base/quote pair.
- `Mangrove`, `KandelSeeder` and `Kandel`: The vault interacts with these contracts to manage provisions and update its position on Kandel.
- `AllowedSwapContract`: An external contract whitelisted by the vault owner, enabling it to execute swap operations on behalf of the vault.

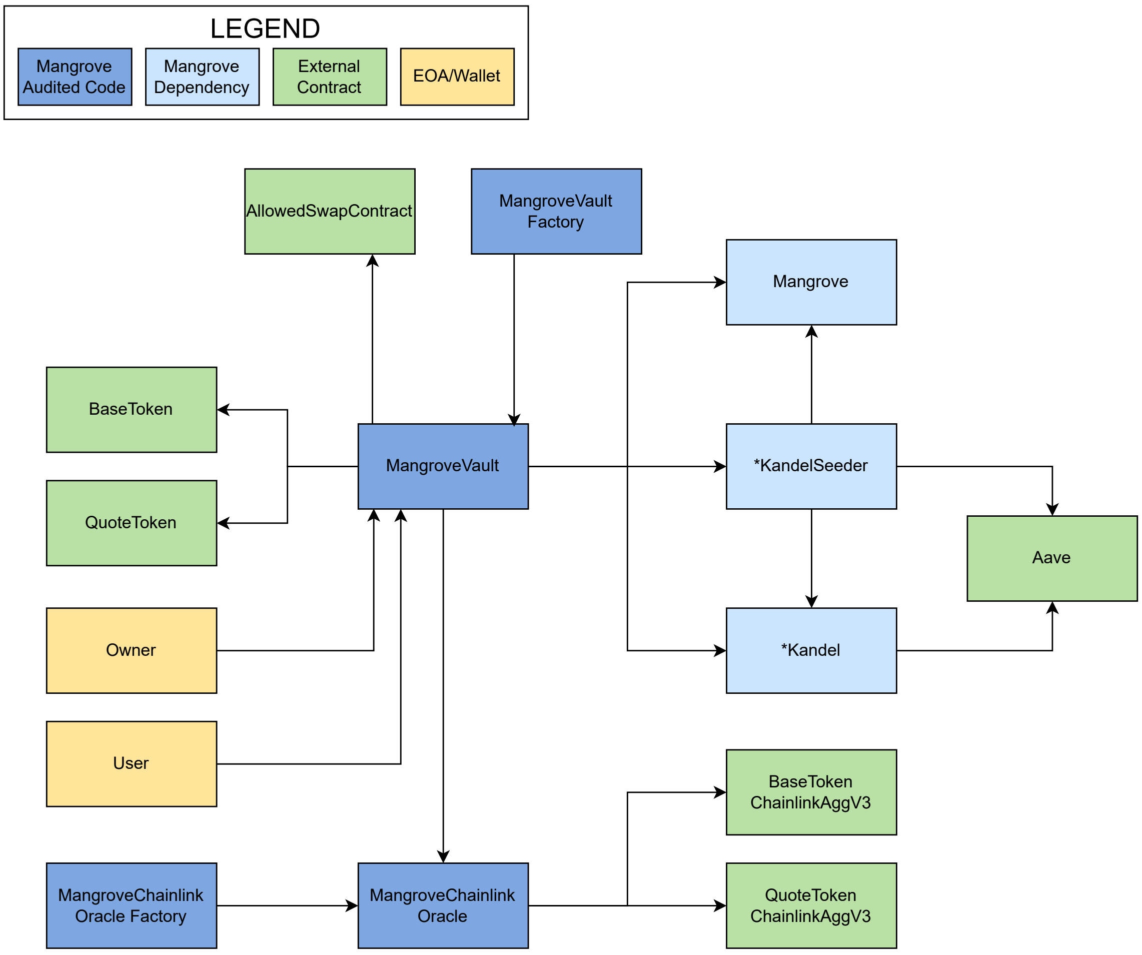


**Fig. 2: Mangrove Vault overview.**

In the following subsections, we outline the key functionalities of the Mangrove Vault, including both user and owner operations.

## 4.1 User Flow: Minting and Burning Vault Shares

### Minting Shares

The MangroveVault allows users to mint shares through the mint(. . . ) function, where they specify the desired mintAmount along with the maximum allowable base and quote amount to transfer to the vault.

```
function mint(uint256 mintAmount, uint256 baseAmountMax, uint256 quoteAmountMax)
    external
    whenNotPaused
    nonReentrant
    returns (uint256 shares, uint256 baseAmount, uint256 quoteAmount)
```

This function calculates the equivalent underlying tokens for the specified mintAmount using the current share price. If the computed amounts exceed the provided maximums, the function reverts. Otherwise, the tokens are transferred from the user to the vault, and the corresponding number of shares is minted to the user.

### Burning Shares

Shareholders can similarly burn their shares using the burn(. . . ) function, specifying the number of shares to burn and the minimum amounts to receive for each token.

```
function burn(uint256 shares, uint256 minAmountBaseOut, uint256 minAmountQuoteOut)
  external
  whenNotPaused
  nonReentrant
  returns (uint256 amountBaseOut, uint256 amountQuoteOut)
```

This function burns the specified number of shares and calculates the corresponding amounts of base and quote tokens. The tokens are transferred to the user if the amounts are not below the defined minimums.

### Representation of Shares

When the vault's total supply is zero, shares in the MangroveVault are represented as the total value in the vault, expressed in quote amounts, computed as follows:

```
uint256 sharesAmount = ((tick.inboundFromOutboundUp(baseAmount) + quoteAmount) * QUOTE_SCALE)
```

Where:

- baseAmount and quoteAmount represent the current balances of the base and quote tokens, including the Kandel balance.
- The inboundFromOutboundUp(...) function converts the baseAmount to a quote amount using the current tick.
- QUOTE_SCALE is a constant that adjusts the result to match the share decimals. It represents the difference between the share decimals and quote decimals.

Once an initial share amount is minted (i.e., when totalSupply is not zero), the conversion between shares and underlying tokens is performed based on the proportion of shares relative to the total supply:

```
quoteAmount = shareAmount * quoteBalance / totalSupply
baseAmount = shareAmount * baseBalance / totalSupply
```

## 4.2 Vault Rebalancing

The vault includes a rebalancing functionality accessible only to the contract owner via the swap(...) function. This function allows the owner to execute swap operations on an external target contract, specifying the data for the call, the amount to transfer out from the vault, and the minimum amount to be received. The sell boolean indicates whether the swap operation involves selling or buying base tokens.

```
function swap(address target, bytes calldata data, uint256 amountOut, uint256 amountInMin, bool sell)
  external
  onlyOwner
  nonReentrant
```

The target contract must be among the allowed swap contracts defined in the allowedSwapContracts mapping, which excludes the vault and Kandel addresses.

## 4.3 Funds State and Position Management

The Mangrove Vault defines three distinct states for its funds: VAULT, PASSIVE, and ACTIVE.

**Vault State:** In this state, the deposited funds remain within the vault. Any open positions on Kandel are closed, and tokens are transferred back to the vault.

**Passive State:** When the vault is in a passive state, the underlying tokens are deposited into the Kandel strategy but are not actively used in market making. No offers are created on Mangrove, and any existing active offers are retracted.

**Active State:** The active state is the only case where funds are actively market-making. In that case, all tokens are deposited into Kandel and used to create the distribution of offers.

Upon deployment, the vault is in the Vault state. The contract owner can update the state through the `setPosition(. . . )` function, which also allows updates to Kandel parameters.

```
function setPosition(KandelPosition memory position) external onlyOwner
```

After each mint, burn, swap, or Kandel parameter update, the vault adjusts the Kandel position to reflect changes in the underlying balances and parameters. If an offer update fails due to low density or other reasons, all Kandel offers are retracted to prevent under-collateralization.

## 4.4 Vault Fees

The vault incurs two types of fees: management fees and performance fees.

- **Management Fees:** Charged as a yearly percentage of the total vault value in quote tokens.
- **Performance Fees:** A percentage applied to the accrued interest on the vault.

Fees are taken in shares, which are minted to the `feeRecipient` address. This operation is implemented within the `_accrueFee()` function, which is called before each mint or burn to ensure accurate share pricing.

```
function _accrueFee() internal returns (uint256 newTotalInQuote, Tick tick)
```

## 4.5 Oracle Integration

The `MangroveVault` interacts with the `MangroveChainlinkOracle` contract to obtain the current tick for the underlying base/quote tokens. This contract serves as an adapter for Mangrove, supporting up to four Chainlink price feeds and combining the fetched prices to return a quote/base price. It exposes a single external function:

```
function tick() external view returns (Tick)
```

The oracle contract can be deployed via the `MangroveChainlinkOracleFactory` factory contract, which enables the creation of `MangroveChainlinkOracle` instances and tracks deployed oracle addresses.

# 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 the finding is to be uncovered and exploited by an attacker. 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 [Medium] Missing check for stale data from Chainlink oracle

**File(s)**: `ChainlinkConsumer.sol`

**Description**: The `getTick(...)` function is designed to calculate the tick value for a base/quote pair using the price fetched from the Chainlink oracle. This function internally calls `getPrice(...)` to retrieve the latest price from Chainlink. However, it currently fails to check the `updatedAt` timestamp of the returned price, which may lead to the acceptance of stale prices.

Moreover, the function does not check whether the L2 Arbitrum sequencer is down. In such a scenario, the oracle data is not up to date and the returned price will become stale.

```
function getPrice(AggregatorV3Interface _aggregator) internal view returns (uint256) {
  if (address(_aggregator) == address(0)) return 1;

  (, int256 price,,,) = _aggregator.latestRoundData();
  // @audit missing check for stale prices
  if (price < 0) revert MangroveVaultErrors.OracleInvalidPrice();
  return uint256(price);
}
```

**Recommendation(s)**: Consider implementing additional checks to ensure the price is not stale.

```
function getPrice(AggregatorV3Interface _aggregator) internal view returns (uint256) {
-         (, int256 price,,,) = _aggregator.latestRoundData();
+        (uint80 roundId, int256 price,, uint256 updatedAt, uint80 answeredInRound) =
+            AggregatorV3Interface(_quotePriceFeed).latestRoundData();
+        require(answeredInRound >= roundId, "Stale price!");
+        require(updatedAt != 0, "Round not complete!");
+        require(block.timestamp - updatedAt <= VALID_TIME_PERIOD);
}
```

Additionally, ensure the L2 Sequencer is up, following the recommendation by Chainlink team.

**Status**: Acknowledged

**Update from the client**: We decided to ignore this finding as such failure from Chainlink or Arbitrum's sequencer is highly unlikely and wouldn't prevent affected LPs to withdraw their funds. We note that other projects also apply this practice.

## 6.2 [Low] The function _swap allows arbitrary calls and is underconstrained

**File(s)**: MangroveVault.sol

**Description**: The `_swap(...)` function allows the contract owner to rebalance the vault by executing swaps between quote and base tokens. However, this function poses some risks due to its ability to make arbitrary calls to the `target` contract, which can lead to malicious behaviors.

For instance, the `target` address can be set to the addresses of the `BASE` or `QUOTE` tokens, a direct call to these token contracts allows for a direct transfer of the vault assets outside the contract. Additionally, there is no control on the swap rate, which can result in unfair swaps or the depletion of contract funds. This behaviour is possible as the `amountInMin` parameter, which is defined by the caller, could be set to zero or an excessively low value.

```
// @audit `amountInMin` can be set to a zero or very low value
function _swap(address target, bytes calldata data, uint256 amountOut, uint256 amountInMin, bool sell) internal {
    // @audit `allowedSwapContracts` does not exclude the base and quote token addresses
    if (!allowedSwapContracts[target]) {
      revert MangroveVaultErrors.UnauthorizedSwapContract(target);
    }
    // ...
    // @audit Arbitrary call to the `target` address
    target.functionCall(data);
    // ...
}
```

**Recommendation(s)**: Reinforce checks within the `_swap(...)` function to prevent potential misuse that could lead to the depletion of vault funds or unfair asset swaps.

**Status**: Mitigated

**Update from the client**: The issues were mitigated via multiple changes.

The first one is the addition of the manager role. A manager is only allowed to swap (rebalance) and set the position. This is the 2 functions that will be done by bots. Thus, the owner can be a multisig or any other trusted and secure account. The owner has all powers still but is the only one able to whitelist swap contract, change the fee data and more.

The second one is the addition of the max spread that is set by the owner of the vault only. Once we get the price from the oracle, we adjust it with the designated spread in order to get the minimum amount we should get. The final minimum amount to receive will be the maximum between the one passed to the swap function and the one we got from this operation. Here is the code to get the adjusted amount to receive :

```
function adjustedAmountInMin(uint256 amountOut, uint256 _amountInMin, bool sell)
    public
    view
    returns (uint256 amountInMin)
  {
    // If maxPriceSpread is set to type(uint).max, we use the default amountInMin
    if (maxPriceSpread == type(uint256).max) {
      return _amountInMin;
    }

    Tick tick = _currentTick();
    uint256 _maxPriceSpread = maxPriceSpread;

    if (sell) {
      // If we are selling, this means that we have to reduce the tick to reduce the amount of quote we receive
      // So we decrease the tick by the max price spread
      tick = Tick.wrap(Tick.unwrap(tick) - _maxPriceSpread.toInt256());
      amountInMin = tick.inboundFromOutboundUp(amountOut);
    } else {
      // If we are buying, this means that we have to increase the tick to increase the amount of quote we receive
      // So we increase the tick by the max price spread
      tick = Tick.wrap(Tick.unwrap(tick) + _maxPriceSpread.toInt256());
      amountInMin = tick.outboundFromInbound(amountOut);
    }
    // We make sure that the amountInMin is at least the amountInMin provided
    amountInMin = Math.max(amountInMin, _amountInMin);
  }
```

The third one is that the owner can not add the base nor quote token to the list of allowed swap contracts.

The fixes were implemented in commit 0d0b90e6526c3a119f670c57496a04aed1d23bb0.

## 6.3 [Info] Use of `transfer` instead of low level call to send native assets

**File(s)**: MangroveVault.sol

**Description**: The `withdrawNative(...)` function in `MangroveVault` currently uses `transfer()` to send native ether to the contract owner. This approach can result in several issues, particularly with multisig wallets that may implement complex fallback functions or additional logic which execution exceeds the gas limit imposed by `transfer()`. Consequently, the usage of `transfer()` could result in failed ether transfers in such scenarios.

**Recommendation(s)**: To avoid the limitations of `transfer()`, consider using a low-level `call` function to transfer native ether.

```
function withdrawNative() external onlyOwner {
-    payable(msg.sender).transfer(address(this).balance);
+   (bool success, bytes memory result) = payable(msg.sender).call{value : address(this.balance)}("");
+   require(success, "Transfer failed");
}
```

**Status**: Fixed

**Update from the client**: This was fixed by the recommended implementation on commit 04c0ed27f13bcd8de970cbdae10a2892c9c0d29b.

# 7 Documentation Evaluation

Software documentation refers to the written or visual information that describes the functionality, architecture, design, and implementation of software. 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 Mangrove Vault documentation**

The Mangrove team has provided comprehensive documentation in the provided README file. This documentation outlines the vault's primary functionalities and highlights its key dependencies, including Mangrove's Kandel strategy. Additionally, the code includes NatSpec documentation for different functions and explanatory comments on complex logic and computations.

# 8 Test Suite Evaluation

## 8.1 Compilation Output

```
> forge build

[] Compiling...
[] Compiling 115 files with Solc 0.8.25
[] Solc 0.8.25 finished in 22.39s
Compiler run successful!
```

## 8.2 Tests Output

```
> forge test

Ran 1 test for test/GeometricKandelExtra.t.sol:GeometricKandelExtraTest
[PASS] testFuzz_firstAskIndex(int24,int24,uint8,uint8) (runs: 256, : 63101, ~: 37614)
Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 116.85ms (116.43ms CPU time)

Ran 20 tests for test/MangroveVault.t.sol:MangroveVaultTest
[PASS] testFuzz_burnShares(uint8,uint128,uint16) (runs: 256, : 8686895, ~: 8705620)
[PASS] testFuzz_initialAndSecondMint(uint8,uint128,uint128) (runs: 256, : 8680355, ~: 8707451)
[PASS] testFuzz_initialMintAmountMatch(uint8,uint128) (runs: 256, : 8584339, ~: 8599264)
[PASS] test_BurningInActive() (gas: 12316763)
[PASS] test_BurningInPassive() (gas: 9056656)
[PASS] test_DensityAndFunds() (gas: 11099736)
[PASS] test_NoFunds() (gas: 8804373)
[PASS] test_PassivefundState() (gas: 8786135)
[PASS] test_aave() (gas: 12066447)
[PASS] test_auth() (gas: 7946369)
[PASS] test_denssityTooLow() (gas: 8188501)
[PASS] test_deployKandel() (gas: 232)
[PASS] test_setPosition() (gas: 11828328)
[PASS] test_setUnauthorizedSwapContract() (gas: 11101096)
[PASS] test_swap() (gas: 435154)
[PASS] test_swapInActive() (gas: 11744350)
[PASS] test_swapInPassive() (gas: 8986370)
[PASS] test_swapInVaults() (gas: 8945080)
[PASS] test_swapIncorrectSlippage() (gas: 11312392)
[PASS] test_unauthorizedSwapContract() (gas: 8174604)
Suite result: ok. 20 passed; 0 failed; 0 skipped; finished in 5.04s (5.72s CPU time)

Ran 2 test suites in 5.06s (5.16s CPU time): 21 tests passed, 0 failed, 0 skipped (21 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