Security Review Report
NM-0208 Mangrove

NETHERMIND
SECURITY

(Mar 19, 2024)

Contents

1 Executive Summary

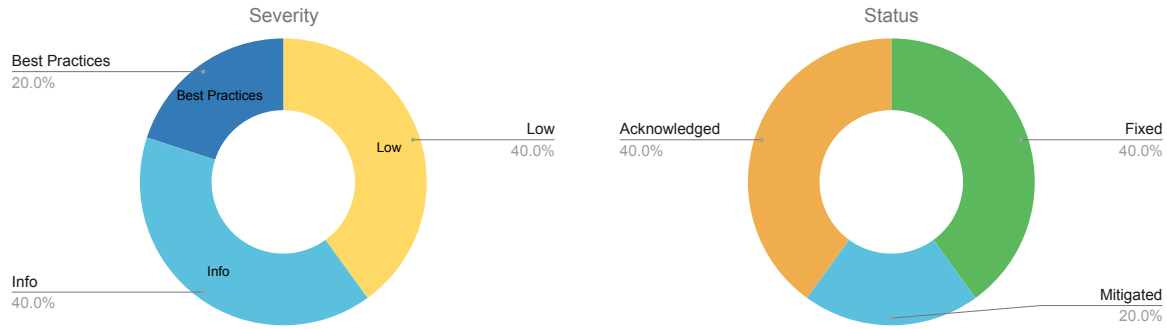
This document outlines the security review conducted by Nethermind Security for part of the strategy-lib developed by Mangrove. `Mangrove` is an on-chain order book DEX that allows liquidity providers to post arbitrary smart contracts as offers.

This code review focuses on two routing logic implementations: **UniswapV3RoutingLogic** and **OrbitLogic**. A routing logic serves as a user-specific implementation for managing the movement of funds in and out of their reserves. These two contracts offer pre-designed logic for users intending to manage their funds within a Uniswap V3 position or the Orbit lending protocol. The routing logic contracts automatically pull funds by closing the positions whenever the trade is taken. This ensures that the user's position can earn yield for the longest duration possible. When the offer is executed, funds will be delivered in a "just in time" manner.

The audited code comprises 429 lines of code. The `Mangrove` and `Nethermind` teams have actively communicated to clarify any remaining questions about the expected behavior of the protocol.

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** five points of attention, where two are classified as `Low` and three are classified as `Informational` or `Best Practice`. 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 discusses the risk rating methodology adopted for this audit. Section 5 details the issues. Section 6 discusses the documentation provided by the client for this audit. Section 7 presents the compilation, tests, and automated tests. Section 8 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** (0), **Medium** (0), **Low** (2), **Undetermined** (0), **Informational** (2), **Best Practices** (1). **(b) Distribution of status: Fixed** (2), **Acknowledged** (2), **Mitigated** (1), **Unresolved** (0)

Summary of the Audit

Audit Type	Security Review
Initial Report	Mar 19, 2024
Final Report	Mar 19, 2024
Methods	Manual Review, Automated Analysis
Repository	mangrove-strats
Commit Hash	**UniV3 logic**: 3fc73862c2d01651a6dd344e04abf0b052172a7a **Orbit logic**: 326859f16be3022f0607d7e31584c44fec7e2d68
Final Commit Hash	b76ab5f938377670ee561a36ad96bba2a1027202
Documentation	Developers documentation
Documentation Assessment	Medium
Test Suite Assessment	Low

2 Audited Files

	Contract	Lines of Code	Lines of Comments	Comments Ratio	Blank Lines	Total Lines
1	src/strategies/routing_logic/restaking/uni-v3/UniswapV3Manager.sol	142	81	57.0%	28	251
2	src/strategies/routing_logic/restaking/uni-v3/UniswapV3RoutingLogic.sol	182	106	58.2%	23	311
3	src/strategies/routing_logic/orbit/OrbitLogicStorage.sol	47	14	29.8%	7	68
4	src/strategies/routing_logic/orbit/OrbitLogic.sol	58	25	43.1%	7	90
	Total	**429**	**226**	**52.7%**	**65**	**720**

3 Summary of Issues

	Finding	Severity	Update
1	Funds within `UniswapV3Manager` are not considered when the position is updated	Low	Acknowledged
2	Missing slippage protection in the `pullLogic(...)` function	Low	Acknowledged
3	Unnecessary logic within `_getTokensArray(...)` function	Info	Fixed
4	oEther token is not supported by Orbit logic	Info	Mitigated
5	Comment improvements	Best Practices	Fixed

4 Risk Rating Methodology

The risk rating methodology used by Nethermind 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 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 formally pass to the client;

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.

5 Issues

5.1 [Low] Funds within `UniswapV3Manager` are not considered when the position is updated

File(s): UniswapV3RoutingLogic.sol

Description: The `_getPosition(...)` function serves to retrieve the position structure associated with a given position ID by interacting with the Uniswap V3 positions manager contract. This function is invoked within `pushLogic` and `pullLogic` functions, followed by a call to `_anyOfToken(...)` to verify that the requested token to be pulled or pushed is either token0 or token1 of the position. The code snippet is provided below:

```
// @audit fetch the position Id and position structure associated with the owner
(uint positionId, Position memory position) = _getPosition(fundOwner);
// @audit reverts if `token` is not token0 or token1 of the position
 _anyOfToken(token, position);
```

The `UniswapV3Manager` contract includes the `changePosition(...)` function, enabling users to update their position ID. If the updated position involves different tokens, the call to `_anyOfToken(...)` will fail. This failure causes the `pushLogic` and `pullLogic` transactions to revert, resulting in the user incurring a penalty on Mangrove. However, the user may still have funds within the `UniswapV3Manager` contract that are not considered when fulfilling the offer.

Recommendation(s): Consider attempting to fulfill the offer using the funds within `UniswapV3Manager` even when the user position is invalid or involves different tokens.

Status: Acknowledged

Update from the client: This issue will be ignored. Users are able to withdraw unused funds from the ERC1155 vault `UniswapV3Manager`. This routing logic will be used with active liquidity management strategies, so we do not want to have succeeding offers with incorrect tokens.

5.2 [Low] Missing slippage protection in the `pullLogic(...)` function

File(s): `UniswapV3RoutingLogic.sol`

Description: The `UniswapV3RoutingLogic` contract is used to source the liquidity from the Uniswap V3 positions into the Smart Offer contracts whenever Maker's offer is taken. This process is managed by the `pullLogic(...)` function.

When the taker executes the offer, the `pullLogic(...)` function checks whether the accumulated Uniswap V3 position fees, combined with the Maker's funds in the `UniswapV3Manager` contract, are enough to cover the `amount` of tokens requested by the offer. If that is the case, it will collect the fees from Uniswap and send the requested `amount` to the Maker's router. If there are not enough funds, the `pullLogic(...)` function will source the missing tokens by removing the Uniswap position itself via a call to the Uniswap's `decreaseLiquidity(...)` function.

The issue with this function is that it does not protect the Maker from price slippage. If the pool undergoes manipulation just before the transaction, the Maker may pay more than anticipated due to slippage. On the other hand, it might result in the `decreaseLiquidity(...)` call delivering fewer tokens than required to fulfill the offer, causing the transaction to revert. Consequently, the Maker loses his provision.

```
function pullLogic(IERC20 token, address fundOwner, uint amount, ...)
    external
    virtual
    override
    returns (uint pulled)
  {
    // ...
    if (_owedOf(token, position) + _inManager(token, fundOwner) < amount) {
      INonfungiblePositionManager.DecreaseLiquidityParams memory params;
      params.tokenId = positionId;
      params.liquidity = position.liquidity;
      params.deadline = type(uint).max;
      // @audit Missing slippage protection: minAmount0, minAmount1
      positionManager.decreaseLiquidity(params);
    }
    _collect(positionId);
    // ...
    _takeAllFromManager(tokens, fundOwner);
    // @audit If not enough was pulled from the position, this will revert
    require(
      TransferLib.transferToken(token, msg.sender, amount),
      "UniV3RoutingLogic/pull-failed"
    );
    // ...
  }
```

Recommendation(s): Consider implementing price slippage checks to protect the offer makers from the abovementioned risks.

Status: Acknowledged

Update from the client: We won't have access to such information during the order. Moreover, if the price of the order posted on Mangrove is within the price range of the Uniswap V3 position, any manipulation of the price incurring slippage will be profitable for the user posting the limit order.

The only downside we acknowledge is that a significant price slippage could either make the offer fail, thus losing the bounty or make it impossible to reposition according to the ratio after the transaction.

Such an attack can be mitigated by the size of the uniswap V3 pool. The bigger the pool, the lower the slippage attack probability.

5.3 [Info] Unnecessary logic within `_getTokensArray(...)` function

File(s): `UniswapV3RoutingLogic.sol`

Description: The `_getTokensArray(...)` takes three tokens as input and returns an array of two or three tokens, depending on whether the first token is included in the position tokens: token0 and token1.

This function is utilized within the `pullLogic(...)` and `pushLogic(...)` functions, where it's consistently preceded by a call to `_anyOfToken(...)`. The latter function ensures that the token in the input is either token0 or token1 of the position. Consequently, the validations within `_getTokensArray(...)` become redundant since it will always return an array of two elements: token0 and token1.

Recommendation(s): Consider simplifying the logic to avoid redundant checks.

Status: Fixed

Update from the client: The function has now been revised and takes only the position as a parameter regardless of the token passed as a wrong token will result in a revert anyway.

```
/// @notice Get the tokens array
/// @param position the position
/// @return tokens the tokens array
function _getTokensArray(Position memory position) internal pure returns (IERC20[] memory tokens) {
    tokens = new IERC20[](2);
    tokens[0] = IERC20(position.token0);
    tokens[1] = IERC20(position.token1);
}
```

Fixed in https://github.com/mangrovedao/mangrove-strats/pull/254/commits/3457c426da4c64cc5e6a3b92104107c7ee10d7a5

5.4 [Info] `oEther` token is not supported by Orbit logic

File(s): `OrbitLogic.sol`

Description: The Orbit protocol offers the `oEther` token that manages native Ether as an underlying token. This token inherits from the `oToken` contract, which does not implement the `underlying()` function. Consequently, calls to `mint()` and `redeemUnderlying()` on this token involve a direct transfer of native ETH rather than an ERC20 transfer.

However, Mangrove exclusively supports ERC20 tokens for both inbound or outbound tokens. Currently, the implementation of `OrbitLogic` contract doesn't handle the transfer or receipt of native ETH. Therefore, to align with Mangrove's specifications, `oEther` must be excluded from the list of supported tokens by the router.

To address this, the `OrbitLogic` contract relies on the specific behavior where a call to `underlying(...)` fails for the `oEther` token. In such cases, the token is skipped and not included in the `overlying` mapping.

```
function setUpStorage() public {
  OToken[] memory cTokens = spaceStation.getAllMarkets();
  for (uint i = 0; i < cTokens.length; i++) {
    OErc20 cToken = OErc20(address(cTokens[i]));
    // @audit call to `underlying()` fails for oEther currently
    try cToken.underlying() returns (address underlying) {
      overlying[IERC20(underlying)] = cToken;
    } catch {
      continue;
    }
  }
}
```

However, in cases where the token implementation changes, and the call to `underlying()` unexpectedly returns a placeholder value. The `oEther` token will be added to the overlying mapping.

Recommendation(s): Consider explicitly excluding the `oEther` token from Orbit to prevent any potential issues arising from changes in token implementation.

Status: Mitigated

Update from the client: These mapping slots can be set permissionlessly using the `SpaceStation` contract. The information given is used as a shortcut to avoid unnecessary storage reads. Thus, passing a placeholder to this function won't affect Mangrove because the tokens are passed directly from Mangrove. Unless the underlying token passed is an already registered address.

In the case of a significant change in the Orbit protocol contract, the logic contracts may be changed and redeployed along with a new audit.

5.5 [Best Practices] Comment improvements

File(s): `OrbitLogic.sol`

Description: Some comments contain typos or incorrect information. A non-exhaustive list of places in the codebase where this is present is shown below:

- Typo in the word "exchange" in the comment inside the `pullLogic(...)` function;
- Incorrect comment in the `pullLogic(...)` function: It states that the pulled funds are transferred **to** the `fundOwner`, but this is not the case since they are transferred **from** the `fundOwner` to the `msg.sender`;

Recommendation(s): Consider fixing the comments listed above.

Status: Fixed

Update from the client: Fixed on commit b76ab5f938377670ee561a36ad96bba2a1027202.

6 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 Mangrove documentation

The `Mangrove` team was available to provide information about the implemented changes as well as addressing any inquiries or concerns raised by the Nethermind auditors.

7 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.