Sharpe Labs

Anirudha Thorat, Sharpe Labs Ltd.

Audit Report: Sharpe Magnum

Date: July 26, 2023

Contents

1 Introduction

Sharpe Magnum, a project spearheaded by Sharpe Labs, is an innovative platform at the forefront of the DeFi ecosystem. Leveraging the ERC4626 standard, it provides a novel approach to staking by creating yield-bearing tokens representing staked assets and their rewards. This report provides a comprehensive audit of Sharpe Magnum, delving into its architecture, functionality, and security.

2 Leveraged Liquid Staking

Leveraged liquid staking is a strategy that allows users to increase their potential rewards from staking by using leverage. This involves borrowing funds to stake more than the user's initial deposit through Flashloans and recursive staking. The potential rewards are amplified, as they are based on the total staked amount (initial deposit plus borrowed funds). However, this strategy also comes with increased risk, as the potential losses are also amplified. Although the potential risks are controlled by on-chain automation solutions deployed using Gelato and Chainlink.

In the context of the Magnum system, users can leverage their positions by depositing ETH, which the smart contract vault uses to repeat the process of lending on Aave. This effectively increases the user's staked amount and potential rewards.

3 ERC4626 Standard

magETH is a tokenized vault that follows the ERC4626 standard. ERC4626 is a proposed standard for tokens that represent shares of tokens that are managed by a vault contract. The vault contract can execute different strategies to optimize the yield of the underlying token, such as lending, farming, or swapping. Tokenized vaults allow the users to deposit and withdraw the underlying token and receive proportional shares of the vault's total assets.

Leveraged staking derivatives vaults are a type of tokenized vault that represents leveraged positions of recursive staking. Recursive staking is a technique that allows users to leverage liquid staking assets and earn compound rewards. Aave and Morpho are two protocols that support recursive staking of stETH, which is a staking derivative token built by Lido. Leveraged staking derivatives vaults use leverage to increase the exposure and yield of the underlying staked asset. They borrow funds from lending platforms and use them to stake more assets. Leverage can amplify both the returns and the risks of staking.

magETH is a leveraged staking derivative vault that represents leveraged stETH and the rewards earned from recursive staking through Aave. It allows the users to stake their ETH and receive magETH in return, which can be used in other protocols or exchanged back to WETH at any time.

The 1inch APIs are used to facilitates low-slippage swaps between different assets. By swapping some of the borrowed funds to stETH, magETH can increase its yield while closely maintaining its peg to ETH.

4 Magnum System Overview

Sharpe Magnum is a novel leverage staking protocol built on top of Liquid Staking Tokens and leverages a combination of lending and staking protocols to boost the staking yields and improve the capital efficiency of holding Ethereum. Sharpe Magnum allows users to deposit ETH and receive magETH, a tradable ERC20 token that represents their leveraged staking position. Users can earn leveraged staking yields while using their magETH in other DeFi applications, such as lending, borrowing, swapping, or providing liquidity.

Sharpe Magnum simplifies the staking experience by providing a single point of access for multiple staking protocols, such as Lido, Rocket Pool, etc. Users can choose the preferred LSTs they want to earn leveraged yields on and benefit from the security and decentralization of the underlying staking protocol. Sharpe Magnum also ensures the safety of the user's assets by monitoring the leverage staking positions on a block-by-block basis and automatically adjusting them to avoid liquidation risks.

Sharpe Magnum achieves leverage staking by using flashloan-based recursive borrowing through Aave and Morpho. The vaults use magETH as collateral to borrow more ETH, which is then used to mint more magETH, creating a recursive loop that increases the capital efficiency and yields for the user.

5 How do Magnum Positions work?

When a user deposits, the following steps are taken to open a Sharpe Magnum position:

1. Borrow a flashloan from Balancer/Aave for WETH.
2. Swap user's ETH to WETH.
3. Swap WETH to stETH through 1inch.
4. Supply stETH to Aave.
5. Borrow WETH from Aave to pay back the flashloan of WETH to Aave.

When a user withdraws, the following steps are taken to close a Sharpe Magnum position:

1. Borrow a flashloan of WETH from Aave/Balancer.
2. Supply WETH to Aave.
3. Withdraw all stETH from Aave.

4. Swap stETH to WETH through 1inch.
5. Pay back the flashloan of WETH to Balancer/Aave.

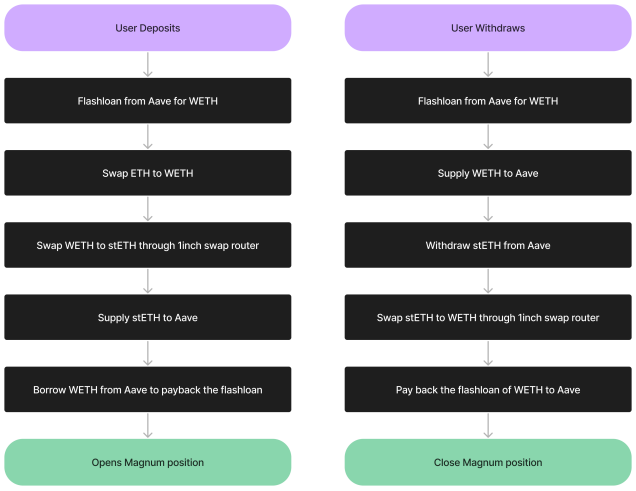

Figure 1: Flowchart showing how Magnum Positions work

Opening Positions

Let r be the initial staking ratio, R be the total stake ratio after leveraging, F be the flashloan amount, and S be the user's initial deposit.

1. Flashloan borrowing: $F = r \cdot S$
2. Total amount of assets after flashloan: $T = S + F$
3. AMM swap: T is swapped to stETH, denoted as T_{stETH}, considering the price impact (PI) and slippage (SL): $T_{stETH} = T \cdot (1 - PI - SL)$
4. Supply stETH to lending protocol: T_{stETH} is supplied to Aave, and the borrowed amount B is calculated as $B = R \cdot S - T_{stETH}$
5. Pay back the flashloan: F

The equation for the total staking ratio R is given by

$$R = (1 + r) \cdot (1 - PI - SL)$$

Closing Positions

Let W be the withdrawal amount from Aave, $stETH_W$ be the amount of stETH withdrawn after the flashloan, and CL be the closing liquidation threshold.

1. Flashloan borrowing: F
2. Supply flashloan amount to lending protocol: F
3. Withdraw stETH from lending protocol: $W = R \cdot S$ and $stETH_W = W/(1 - CL)$
4. AMM swap: $stETH_W$ is swapped back to WETH, denoted as $WETH_W$, considering price impact (PI) and slippage (SL): $WETH_W = stETH_W \cdot (1 - PI - SL)$
5. Pay back the flashloan: F

To account for the price impact, slippage, and closing liquidation threshold, the closing equation is:

$$WETH_W = \frac{R \cdot S}{1 - CL} \cdot (1 - PI - SL)$$

These equations showcase Sharpe Magnum's leveraged staking mechanism, incorporating the workings of AMMs, flashloans and lending protocols. The opening and closing of positions involve a series of steps that require precise calculations to ensure optimal capital efficiency and yield optimization while maintaining the required risk management standards.

6 Liquidation and Risk Management

Risk management is a crucial aspect of the Magnum system. The system implements a series of measures to manage risk and ensure the security and stability of the platform.

6.1 Liquidation

Liquidation in the Magnum system occurs when a user's position becomes undercollateralized. This could happen if the value of the collateral falls too much compared to the borrowed amount. The system monitors the health of each position and triggers a liquidation if the collateralization ratio falls below a certain threshold.

During the liquidation process, the system sells a portion of the user's collateral to repay the borrowed amount. The user is charged a liquidation penalty, which is a percentage of the liquidated collateral. The remaining collateral, after repaying the borrowed amount and the penalty, is returned to the user.

6.2 Liquidation Mechanism

In the event that the collateralization ratio of a position falls below the lending protocol's minimum threshold, the position is subject to liquidation. The collateralization ratio is calculated using the formula: $CR = \frac{C}{D} \times 100$, where C is the collateral value, and D is the debt value. Liquidation ensures the overall health and stability of the protocol.

6.3 Risk Management Measures

Risk management is a crucial aspect of Sharpe Magnum's architecture. The protocol employs automation as well as owner access to deleverage the vault manually in case of a blackswan event to safeguard user assets and ensure the stability of the system.

- **Collateralization Ratio:** The system enforces a minimum collateralization ratio to prevent under-collateralization..
- **Risk Parameters:** The system allows governance to adjust the collateralization ratios based on the market conditions.
- **Position Monitoring:** The system continuously monitors the health of each position and triggers liquidation if necessary.

These measures help manage risk and ensure the long-term sustainability of the Magnum platform.

7 Key Features

The Sharpe Magnum protocol is designed with a number of key features aimed at optimizing staking yields. These include improved APYs of staking pools, non-custodial protocol operation, and the same guarantees and liquidity as underlying staking mechanisms. The key features of the Magnum system include:

- **Leveraged Liquid Staking:** Users can leverage their staked WETH to earn increased rewards.
- **Governance:** SHARPE token holders can vote on proposals to change the system parameters.
- **Incentive Mechanisms:** Users who stake their magETH in the Partner Program earns SHARPE tokens.

8 Summary

This document represents the final report of the comprehensive audit conducted on the Sharpe Magnum DeFi platform. The audit involved an exhaustive review

of the platform's code, architecture, and security measures. The objective was to identify and mitigate any potential vulnerabilities and ensure the platform's robustness, security, and reliability.

9 Methodology

The audit employed a multi-faceted approach combining automated testing, manual code review, and system-level analysis. Tools like Slither were used for static testing. The audit process also involved a thorough manual review of the code to identify logic flaws, security vulnerabilities, and other potential issues that automated testing tools may miss.

10 Recommendations

Based on the audit findings, several recommendations are provided to enhance the security and efficiency of the Sharpe Magnum platform:

- Implement additional input validation checks to prevent potential vulnerabilities, such as reentrancy attacks and integer overflow/underflow.
- Enhance the code documentation to improve code readability and maintainability for future updates and external reviews.
- Consider adding more unit tests and edge cases to cover a wider range of scenarios during testing.
- Implement access control mechanisms to restrict sensitive functions' usage and ensure proper governance.
- Collaborate with third-party security experts for additional external audits to strengthen the platform's security posture.

11 MAGETH FUNCTIONS

11.1 Internal Functions:

11.1.1 Swap:

```
function swap(bytes memory _data) internal returns(uint)
```

This is an internal function that swaps two tokens using 1inchV5.

Name	Type	Description
_data	bytes	Data representing swap route and swap data (taken from 1inch external API call)

11.1.2 TotalAssets:

`function totalAssets() internal view override returns (uint256)`

This function returns the total amount of underlying assets held by the vault.

11.1.3 PreviewDeposit:

`function previewDeposit(uint256 assets) internal view override returns (uint256)`

This function allows users to simulate the effects of their deposit at the current block.

Name	Type	Description
assets	uint256	Amount of Eth user wants to deposit

11.1.4 PreviewRedeem:

`function previewRedeem(uint256 shares) internal view override returns (uint256)`

This function allows users to simulate the effects of their deposit at the current block.

Name	Type	Description
shares	uint256	Amount of share user wants to redeem

11.2 External Functions:

11.2.1 MaxDeposit:

`function maxDeposit(address) external pure override returns (uint256)`

Maximum amount of the underlying asset that can be deposited into the Vault for the receiver, through a deposit call.

11.2.2 Deposit:

`function deposit(uint256 assets, address receiver, bytes calldata _data1) external payable onlyEOA nonReentrant override returns (uint256 shares)`

Mints shares Vault shares to the receiver by depositing assets of underlying tokens.

Name	Type	Description
assets	uint256	Amount of asset user wants to deposit
receiver	address	Address of share receiver
_data1	bytes	Data representing swap route and swap data (taken from 1inch external API call)

11.2.3 Withdraw:

```
function withdraw(
        uint256 shares,
        address receiver,
        address owner,
        bytes calldata _data1
) external nonReentrant onlyEOA override returns(uint256 assets)
```

Burns shares from the owner and sends exactly assets of underlying tokens to receiver.

Name	Type	Description
shares	uint256	Amount of share user want to burn
receiver	address	Address of asset receiver
Owner	address	Address of the owner of shares
_data1	bytes	Data representing swap route and swap data (taken from 1inch external API call)

11.2.4 ExecuteOperation:

```
function executeOperation(
        address[] calldata assets,
        uint[] calldata amounts,
        uint[] calldata premiums,
        address initiator,
        bytes calldata params
    ) external override returns (bool)
```

The function is designed to be called by the Aave lending pool contract and executes a specific operation on behalf of the caller. The function is designed to be called by the Aave lending pool contract and executes a specific operation on behalf of the caller. There are four possible operation types that are deposit, withdraw, deleverage and leverage.

Name	Type	Description
assets	address[]	An array of addresses representing the assets to be borrowed.
amounts	uint[]	An array of unsigned integers representing the amounts of each asset to be borrowed.
premiums	uint[]	An array of unsigned integers representing the premiums associated with each asset
initiator	address	An Ethereum address representing the initiator of the operation
params	bytes	Params in bytes data format

11.2.5 Deleverage:

`function deleverage(uint amount, bytes calldata _data1) external nonReentrant onlyGovernance`

Deleverage function can only be called by governance and it is used to deleverage the vault.

Name	Type	Description
amount	uint256	amount of Weth that must be repaid to the morpho in order to deleverage the vault.
_data1	bytes	Data representing swap route and swap data (taken from 1inch external API call)

11.2.6 Leverage:

`function leverage(uint amount, bytes calldata _data1) external nonReentrant onlyGovernance`

Leverage function can only be called by governance and it is used to leverage the vault.

Name	Type	Description
amount	uint256	amount of Weth need to borrow to leverage the vault
_data1	bytes	Data representing swap route and swap data (taken from 1inch external API call)

11.2.7 Harvest:

```
function harvest(uint amount,
bytes calldata _data1)external onlyGovernance
```

Harvest function can only be called by governance and it is used to help maintain the desired level of leverage for the vault.

Name	Type	Description
amount	uint	Amount of stETH to harvest
_data1	bytes	Data representing swap route and swap data (taken from 1inch external API call)

11.2.8 AssetsOf:

`function assetsOf(address user) external view override returns (uint256)`

This function returns total assets of user.

Name	Type	Description
user	address	Address of user whose total asset we want in return

11.2.9 AssetsPerShare:

`function assetsPerShare() external view override returns (uint256)`

This function returns the current value of the assets per share.

11.2.10 MaxMint:

`function maxMint(address) external pure override returns (uint256)`

Maximum amount of shares that can be minted from the Vault for the receiver, through a mint call.

11.2.11 MaxWithdraw:

`function maxWithdraw(address owner) external view override returns (uint256)`

Maximum amount of the underlying asset that can be withdrawn from the owner balance in the Vault, through a withdraw call.

Name	Type	Description
owner	address	Address of owner

11.2.12 MaxRedeem:

`function maxRedeem(address owner) external view override returns (uint256)`

Maximum amount of Vault shares that can be redeemed from the owner balance in the Vault, through a redeem call.

Name	Type	Description
owner	address	Address of owner

11.2.13 SetGovernance:

`function setGovernance(address _governance) external onlyGovernance`

The "setGovernance" function is an external method that can only be called by the current governance entity.

Name	Type	Description
_governance	address	Address of new governance

11.2.14 AcceptGovernance:

`function acceptGovernance() external`

This function is used in conjunction with the "setGovernance" function, allowing for a smooth and secure transition of governance control from one entity to another.

11.2.15 SetCap:

`function setCap(uint _cap)external onlyGovernance`

This function should be called by the existing governance address and it is used to set the cap (cap is the vault limit or allowable deposit amount for non-beacon user).

Name	Type	Description
_cap	uint	Total vault limit or cap

11.2.16 SetDegenCap:

`function setDegenCap(uint _cap) external onlyGovernance`

This function should be called by the existing governance address and it is used to set the cap (cap is the total vault limit or total allowable deposit).

Name	Type	Description
_cap	uint	Vault capacity for Beacon holders

11.2.17 SetRewardDistributor:

`function setRewardDistributor(address _rewardDistributor)external onlyGovernance`

This function sets the address of the reward distributor contract which distributes morpho rewards to magETH holder.

Name	Type	Description
_rewardDistributor	address	address of the reward distributor contract

11.2.18 SetTraitIdAndMaxAge:

`function setTraitIdAndMaxAge(uint256 _traitId, uint64 _maxAge)external onlyGovernance`

This contract sets traitID and max age of beacon.

Name	Type	Description
_traitId	uint256	traitID of the beacon
_maxAge	uint64	Max age of the beacon

11.2.19 WrapIdleEthToWeth:

`function WrapIdleEthToWeth(uint amount)external onlyGovernance`

This function wraps input ETH amount which is idle in vault contract.

Name	Type	Description
amount	uint	Amount of ETH to wrap

11.2.20 ReinvestIdleSteth:

`function reinvestIdleSteth(uint amount)external onlyGovernance`

This function can only be called by governance to reinvest the idle stETH of the vault on morpho.

Name	Type	Description
amount	uint	Amount of stETH to reinvest in vault

11.2.21 ReinvestIdle:

`function reinvestIdle(uint amount, bytes calldata _data)external onlyGovernance`

This function can only be called by governance to reinvest the idle WETH of the vault on morpho.

Name	Type	Description
amount	uint	Amount of WETH to reinvest in vault
_data	bytes	Additional data (if needed)

11.2.22 ClaimReward:

`function claimReward(address _account, uint _claimable, bytes32[] calldata _proof)external`

This function is used by magETH holders to claim morpho token. This function verifies whether the caller can claim morpho tokens using Merkle tree proofs.

Name	Type	Description
_account	address	Address of reward claimer
_claimable	uint	Morpho reward amount to claim
_proof	bytes32[]	Merkle tree proof that proves the account can claim input claimable amount

11.2.23 ConvertToShares:

`function convertToShares(uint256 assets) public view returns (uint256)`

This function returns the amount of shares that the Vault would exchange for the amount of assets provided.

Name	Type	Description
assets	uint256	Amount of assets

11.2.24 ConvertToAssets:

```
function convertToAssets(uint256 shares) public view returns (uint256)
```

This function returns the amount of assets that the Vault would exchange for the amount of shares provided.

Name	Type	Description
shares	uint256	Amount of shares

11.2.25 ConvertToSupply:

```
function convertToSupply(uint256 assets) public view returns (uint256 supplyAmount, uint256
```

This function gives the amount of supply token and borrow token required to remove in order to remove the input assets amount from the morpho position while keeping the leverage the same.

Name	Type	Description
assets	uint256	Amount of assets

11.2.26 GetVaultsActualBalance:

```
function getVaultsActualBalance() public view returns(uint amount)
```

This function gives the theoretical amount of stETH that the vault will have after paying all the borrowed WETH on morpho.

11.2.27 GetBorrowedAmountForDeleverage:

```
function getBorrowedAmountForDeleverage() public view returns(uint amount)
```

This function returns the amount of borrowed WETH needed to repay to deleverage the vault.

11.2.28 GetBorrowedAmountForLeverage:

```
function getBorrowedAmountForLeverage() public view returns(uint amount)
```

This function returns the amount of borrowed WETH needed to supply to leverage the vault.

11.2.29 GetHarvestAmount:

```
function getHarvestAmount() public view returns(uint amount)
```

This function returns the amount of stETH that needs to be supplied and Weth that needs to be borrowed to re-balance the vault position on morpho and bring the leverage to the desired level.

11.3 Private Functions

11.3.1 TriggerFlashLoan:

```
function triggerFlashLoan(uint8 operation, uint amount,
                          bytes calldata _data1, address _receiver) private
```

The triggerFlashLoan function is a private function that is used to trigger a flash loan from the Aave lending pool. The flash loan can be used to perform various operations like opening or closing positions, deleveraging, or leveraging.

Name	Type	Description
operation	uint8	Operation can be numbers ranging from 0 to 3, 0 representing deposit, 1 representing withdraw, 2 representing deleverage, and 3 representing leverage.
amount	uint	Flashloan Amount of Weth
_data1	bytes	Data representing swap route and swap data (taken from 1inch external API call)
_receiver	address	Address of receiver for withdrawal operation

```
_open1xLeverage:
function _open1xLeverage(uint amount, bytes calldata data, address receiver) private

This function is used in the deposit function when the vault's current leverage is 1x.
This function deposit asset on morpho by supply without borrowing any Weth.
```

Name	Type	Description
amount	uint	amount of ETH supplied by user
data	bytes	Data representing swap route and swap data (taken from 1inch external API call)
receiver	address	Address of receiver

11.3.2 _close1xLeverage:

```
function _close1xLeverage(uint amount, bytes calldata data, address _receiver) private
```

This function is used in the withdraw function when the vault's current leverage is 1x. This function withdraws the amount of stETH from morpho and converts it into Weth. Then it transfers the ETH to the receiver by unwrapping the Weth.

Name	Type	Description
amount	uint	Amount of stETH to remove
data	bytes	Data representing swap route and swap data (taken from 1inch external API call)
_receiver	address	Address of receiver

12 Security

The security of the Sharpe Magnum platform has been a paramount focus during the audit. The following security measures have been implemented to safeguard the platform and its users:

- Thorough input validation and parameter checks to prevent potential attack vectors.
- Implementation of the Checks-Effects-Interactions pattern to mitigate reentrancy vulnerabilities.
- Usage of pull payments to avoid potential issues with function calls inside loops.
- Adoption of OpenZeppelin libraries to ensure the use of secure and battle-tested smart contract standards.
- Robust access control mechanisms to restrict sensitive functions to authorized entities only.
- Regular monitoring and updates to address emerging security threats and vulnerabilities.

13 Security Tests

The security of the Sharpe Magnum platform has been a paramount focus during the audit. Various measures have been implemented to safeguard the platform and its users from potential vulnerabilities and risks.

13.1 Reentrancy

To mitigate reentrancy attacks on sensitive functions such as `deposit()`, `withdraw()`, `leverage()`, and `deleverage()`, the vault contract utilizes `@openzeppelin`'s `ReentrancyGuard` modifier (`nonReentrant`). This ensures that there are no nested (reentrant) calls, reducing the risk of reentrancy vulnerabilities.

13.2 Static Analysis

The audit conducted static analysis of the smart contracts to identify potential contract vulnerabilities. The `Slither` tool was used to test the security of the smart contracts. The following issues were detected in the Slither static test:

13.2.1 High Severity Issues

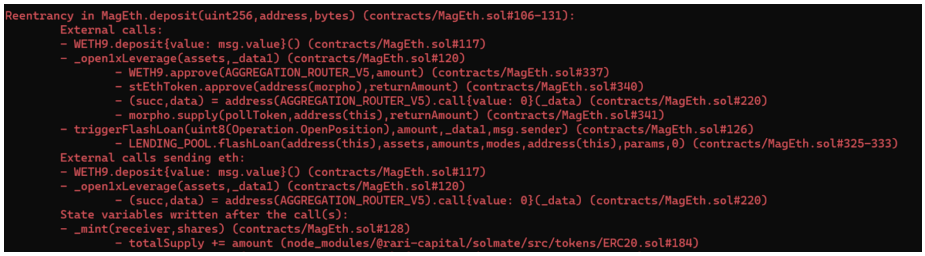

Figure 2: Reentrancy Vulnerability

Reentrancy Vulnerability: Slither detected a reentrancy vulnerability in the `deposit` function of `MagEth.sol`. To address this issue, OpenZeppelin's `ReentrancyGuard` modifier (`nonReentrant`) has been utilized, ensuring there are no nested (reentrant) calls. The vulnerability has been fixed.

13.2.2 Medium Severity Issues

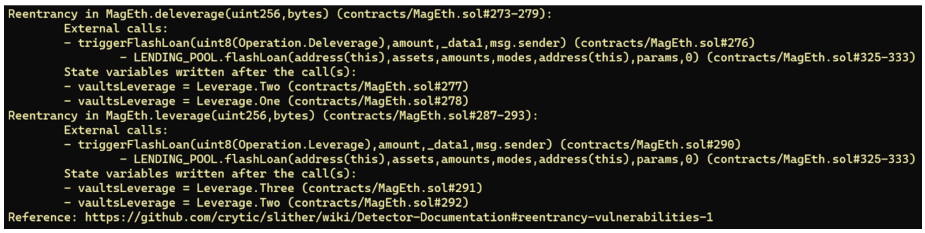

Figure 3: Reentrancy Vulnerability

Reentrancy Vulnerability: The analysis conducted by Slither identified a potential reentrancy vulnerability in the `deleverage` and `leverage` functions of `MagEth.sol`. To mitigate this risk, the code has been modified to incorporate OpenZeppelin's `ReentrancyGuard` modifier (`nonReentrant`), which helps prevent nested calls and potential reentrancy issues. The vulnerability has been fixed.

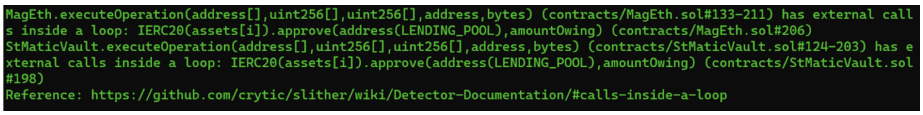

Figure 4: Calls Inside a Loop

13.2.3 Low Severity Issues

Calls Inside a Loop: Slither detected a potential vulnerability in the `executeOperation` function of `MagEth.sol`, which could lead to a denial-of-service attack due to calls being made inside a loop. To address this issue, the code has been modified to ensure that only the `MagEth` contract can initiate the `executeOperation` function. Additionally, the `triggerFlashLoan` function in `MagEth.sol` has been modified to allow borrowing only one asset from Aave, effectively limiting the loop to a maximum of one iteration. These changes mitigate the risk of calls inside a loop. The vulnerability has been fixed.

13.3 Manual Analysis

Apart from the static analysis, a manual review of the code was conducted to identify new vulnerabilities and verify the vulnerabilities found during the static analysis. The contracts were thoroughly manually analyzed, and their logic was checked.

13.4 `onlyEOA` Modifier

The `onlyEOA` (External Owned Account) modifier is used in Solidity contracts to restrict the execution of a function to only external owned accounts (EOAs), which are user-controlled Ethereum addresses. This is done to prevent contract-controlled addresses from calling certain functions that could potentially compromise the security of the contract or the tokens it controls. The `onlyEOA` modifier ensures that only externally owned accounts (i.e., user-controlled addresses) are allowed to execute the function, while contract-controlled addresses are not. The modifier is defined as follows:

```
modifier onlyEOA {
    require(msg.sender == tx.origin, "Only EOA");
    _;
}
```

13.5 Tools and Platforms Used for Audit

The audit utilized various tools and platforms to ensure a comprehensive review of the Sharpe Magnum platform. `Slither`, `Hardhat`, and `Remix IDE` were employed for static analysis, Ethereum development environment, and online IDE, respectively.

13.6 Number of Issues per Severity

The audit identified and resolved a few issues of varying severity. As a result of the thorough analysis and mitigation measures, the platform is now more robust and secure.

Issue Status	High	Medium	Low
Open	0	0	0
Closed	1	1	1

14 Future Roadmap

The future roadmap of Sharpe Magnum, as detailed in the lightpaper, outlines the platform's planned advancements and expansions:

- *Cross-chain Compatibility*: Integrating with other blockchain networks to expand the asset pool and user base.
- *Advanced Risk Management Algorithms*: Enhancing risk assessment mechanisms to ensure efficient and secure platform operations.
- *Additional Liquid Staking Derivatives*: Introducing more staking derivatives to cater to diverse user preferences and risk appetites.
- *Collaborations and Partnerships*: Forging strategic partnerships with other DeFi projects and industry participants to broaden the platform's reach.
- *Community Governance*: Implementing community-driven governance mechanisms to empower platform participants in decision-making processes.
- *Enhanced User Experience*: Continually improving user interfaces and accessibility to attract a broader user base.