STATE MIND

Mellow LRT Obol

28-05-2024 - 30-05-2024

Table of contents

1. Project brief

Title	Description
Client	Mellow
Project name	Mellow LRT Obol
Timeline	28-05-2024 - 30-05-2024
Initial commit	4ceae7e28979a7ecc5ff4b34531e4e548efe67c2
Final commit	1c885ad9a2964ca88ad3e59c3a7411fc0059aa34

Short Overview

Mellow LRT functions as an LRT constructor, enabling users to deploy and manage their LRTs securely. Key features include robust access control and strategic asset management through modules and strategies.

DefaultObolStakingStrategy.sol and StakingModule.sol contracts are specifically designed for Obol's Vault. This Vault stakes ETH into Obol's validator sets (through Lido Simple DVT Module) and then restakes it based on a chosen strategy.

Project Scope

The audit covered the following files:

- DefaultBondModule.sol
- AdminProxy.sol
- WStethRatiosAggregatorV3.sol
- DefaultProxyImplementation.sol
- ChainlinkOracle.sol
- SimpleDVTStakingStrategy.sol
- ConstantAggregatorV3.sol
- Vault.sol
- StakingModule.sol
- RestrictingKeeper.sol

2. Finding severity breakdown

All vulnerabilities discovered during the audit are classified based on their potential severity and have the following classification:

Severity	Description
Critical	Bugs leading to assets theft, fund access locking, or any other loss of funds to be transferred to any party.
High	Bugs that can trigger a contract failure. Further recovery is possible only by manual modification of the contract state or replacement.
Medium	Bugs that can break the intended contract logic or expose it to DoS attacks, but do not cause direct loss of funds.
Informational	Bugs that do not have a significant immediate impact and could be easily fixed.

Based on the feedback received from the Client regarding the list of findings discovered by the Contractor, they are assigned the following statuses:

Status	Description
Fixed	Recommended fixes have been made to the project code and no longer affect its security.
Acknowledged	The Client is aware of the finding. Recommendations for the finding are planned to be resolved in the future.

3. Summary of findings

Severity	# of Findings
Critical	0 (0 fixed, 0 acknowledged)
High	0 (0 fixed, 0 acknowledged)
Medium	0 (0 fixed, 0 acknowledged)
Informational	9 (2 fixed, 7 acknowledged)
Total	9 (2 fixed, 7 acknowledged)

4. Conclusion

During the audit of the codebase, 9 issues were found in total:

- 9 informational severity issues (2 fixed, 7 acknowledged)

The final reviewed commit is 1c885ad9a2964ca88ad3e59c3a7411fc0059aa34

Deployment

Contract	Address
Vault (Proxy)	0x5E362eb2c0706Bd1d134689eC75176018385430B
Vault (Implementation)	0xe2D2E90122cb203CF1565a37ef90a256843A825A
VaultConfigurator	0xDee41701310f48744e6Bb4A5df6B5e714cE49133
Initializer	0x969A0c7699ad0AC38fE05117c81D662762443E07
Erc20TvlModule	0x2c73350310C2b8c721d8192bd7620D1DCB1219ce
StakingModule	0xD570E16E3B62F05EcF3ff2706D331B7f56453adA
ManagedRatiosOracle	0xFeAFe509fae65962EF81555E3f078D58aF7ca3e9

ChainlinkOracle	0x39D5F9aEbBEcba99ED5d707b11d790387B5acB63
ConstantAggregatorV3	0x278798AE6ea76ae75b381eA0D8DF140C1D5a7712
WStethRatiosAggregatorV3	0x966a3b1c9d477D113630290F037b12349649d1bd
DefaultProxyImplementation	0xB8eF363E1909665c18BF0CB72Cba9a8152413A2E
ManagedValidator	0xA1b3a352c3fC7cfcBD36381CC2D0b157d6843473
SimpleDVTStakingStrategy	0x078b1C03d14652bfeeDFadf7985fdf2D8a2e8108
TransparentUpgradeableProxy-ProxyAdmin	0x8E6C80c41450D3fA7B1Fd0196676b99Bfb34bF48
ProxyAdmin (Multisig 5/8)	0x81698f87C6482bF1ce9bFcfC0F103C4A0Adf0Af0
Admin (Multisig 5/8)	0x9437B2a8cF3b69D782a61f9814baAbc172f72003
CuratorAdmin (Multisig 3/6)	0x2E93913A796a6C6b2bB76F41690E78a2E206Be54
CuratorOperator (EOA)	0x2afc096981c2CFe3501bE4054160048718F6C0C8

5. Findings report

INFORMATIONAL-01	Unused **IStakingModule** events	Fixed at: 76dd4b4

Description

Lines:

- IStakingModule.sol#L94
- IStakingModule.sol#L101

The **StakingModule** contract does not use events from the **IStakingModule** interface. Apparently, these events should be called at the end of the **StakingModule._wethToWSteth()** and **StakingModule.convertAndDeposit()** functions.

Recommendation

We recommend adding these events to the appropriate functions.

INFORMATIONAL-02	Incomplete sanity check	Acknowledged

Description

Line: StakingModule.sol#L62

In the function **StakingModule.convertAndDeposit()**, there's a sanity check to ensure that the **bufferedEther** is not less than the **unfinalizedStETH**. This check is intended to prevent deposits when there are insufficient buffer funds. However, this check does not account for the **amount** of ETH deposited in the **StakingModule._wethToWSteth()** function, which could be used for deposit in the staking module.

Recommendation

We recommend performing the check after the **_wethToWSteth(amount)** function call, ensuring the **amount** is considered in the available funds.

Client's comments

> In this function we want our ETH to be completely sent to the corresponding stakingModule. The logic of depositBufferedEther is that only that part of ETH that is greater than unfinalizedStETH can be sent to create new validators. Accordingly, before we deposit eth into steth, the system must have a state that bufferedEther >= unfinalizedStETH, which is checked in the code.

INFORMATIONAL-03	Inflexible design for checking converted amounts in **processWithdrawals**	Fixed at: 499ee6c

Description

Line: SimpleDVTStakingStrategy.sol#L83

During the standard **convertAndDeposit** flow **WETH** is converted to **wstETH** and then it is deposited into the specific staking module. Atomic execution of the transaction ensures that the submitted ether goes to the desired module.

In function SimpleDVTStakingStrategy.processWithdrawals() only convert happens (if needed) to cover withdrawals. So ether is deposited into **Lido**, but from there it can be deposited to any staking module. So assets of **Vault** with **SimpleDVTStakingStrategy** connected may go to another module different from the module in **StakingModule**

The parameter **maxAllowedRemainder** is used to restrict the amount of converted **WETH** during withdrawals to ensure that not too many tokens go into other modules. But such a solution is vulnerable to front-run attacks, an attacker can deposit **wstETH** (if it will be enabled as a deposit token) before **processWithdrawals()** transaction or he can directly transfer some weis to violate the check at SimpleDVTStakingStrategy.sol#L83. Also if the operator or admin are compromised, such architecture allows them to convert all the **WETH** and especially process all the withdrawals.

Recommendation

We recommend adding a parameter to limit then **amountForStake** argument instead of using **maxAllowedRemainder** and setting a maximum limit for the current balance of **wETH** to prevent large accumulations in the **Vault** and depositing in different modules. This will ensure that **SimpleDVTStakingStrategy.convertAndDeposit()** will be called firstly in that case.

```
function processWithdrawals(
    address[] memory users,
    uint256 amountForStake
) external returns (bool[] memory statuses) {
  _requireAtLeastOperator();
  if (IERC20(weth).balanceOf(address(vault)) > maxAllowedBalanceForWithdraw) revert CustomError();

  if (users.length == 0) return statuses;
  emit ProcessWithdrawals(users, amountForStake, msg.sender);

  if (amountForStake == 0) return vault.processWithdrawals(users);
  if (amountForStake > maxAllowedAmountForStake) revert CustomError();


  ...
```

INFORMATIONAL-04	Status of delegate call in **processWithdrawals()** should be checked	Acknowledged

Description

Line: SimpleDVTStakingStrategy.sol#L72

An operator or admin sets the value of **amountForStake** greater than zero in a case when additional **wstETH** is needed to cover withdrawals. The **convert** function is called via Vault's **delegateCall** which doesn't revert, it just returns the status of the call. So, in case of failure of the call at SimpleDVTStakingStrategy.sol#L72 execution will simply continue and withdrawals may be processed with unexpected statuses.

Recommendation

We recommend checking the status and reverting in case the **vault.delegateCall()** fails.

INFORMATIONAL-05	Gas optimization: transient storage in Reentrancy Guard	Acknowledged

Description

Lines:

- Vault.sol#L12
- VaultConfigurator.sol#L8

Using **ReentrancyGuardTransient** instead of **ReentrancyGuard** reduces gas usage.

From OpenZeppelin Docs:

```
If EIP-1153 (transient storage) is available on the chain you're deploying at,
consider using {ReentrancyGuardTransient} instead.
```

Recommendation

We recommend applying changes to save gas.

Client's comments

> We would like to use only the release version of OZ to avoid possible vulnerabilities.

INFORMATIONAL-06	ChainlinkOracles doesn't support a negative answer	Acknowledged

Description

Line: ConstantAggregatorV3.sol#L11

ChainlinkOracle reverts if **answer < 0** ChainlinkOracle.sol#L65.

Recommendation

We recommended adding an answer check to the **ConstantAggregatorV3** constructor.

```
if (_answer < 0) revert InvalidOracleData();
```

INFORMATIONAL-07	Sanity check for **Vault**'s underlying token in **SimpleDVTStakingStrategy**	Acknowledged

Description

Line: SimpleDVTStakingStrategy.sol#L8

SimpleDVTStakingStrategy is based on interaction with two tokens: **WETH** and **wstETH** (we don't account for **StETH**). In both functions: SimpleDVTStakingStrategy.convertAndDeposit() and SimpleDVTStakingStrategy.processWithdrawals(), convert from **WETH** to **wstETH** occurs. But there is no check that **WETH** and **wstETH** are underlying tokens of connected **Vault**. Cases when **wstETH** is not the underlying token are dangerous because after the conversion there will be no possibility of withdrawing **wstETH**.

Recommendation

We recommend adding a check in the constructor that **Vault** contains **WETH** and **wstETH** as underlying tokens or making such checks in a separate validator.

Client's comments

> This can complicate the strategy deployment process by adding a requirement to first add all underlying Tokens and then initialize the strategy. Moreover, even if wsteth is present as the underlyingToken, the vault admin can subsequently delete it, which is why the check in the constructor will not be effective. We believe that a vault admin should take into account the specifics of how a strategy works before adding it.

INFORMATIONAL-08	Potential frontrunning of deposit	Acknowledged

Description

Line: StakingModule.sol#L66

The **StakingModule.convertAndDeposit()** function is vulnerable to frontrunning. An attacker can monitor the mempool, identify a transaction calling this function, and preemptively call **depositSecurityModule.depositBufferedEther()** using the signatures available in the transaction. This causes the original transaction to revert, thereby blocking the **StakingModule._wethToWSteth()** conversion.

Recommendation

We recommend using a private mempool for transactions with non-zero **amount** to prevent frontrunning.

INFORMATIONAL-09	Unnecessary convert of **WETH** during **convertAndDeposit()**	Acknowledged

Description

Line: StakingModule.sol#L48

In function convertAndDeposit() **maxDepositsCount** is calculated based on the limit in **DepositSecurityModule**, available ether in buffer and available keys in **StakingModule** (we omit allocation mechanics of **StakingRouter** for simplicity). After that, the conversion of **WETH** balance of **Vault** occurs. But there are cases when ether in **Lido** buffer would be enough to cover deposits.

Example:

```
wethBalance = 100 ether
bufferedEther = 10000 ether
unfinalizedStETH = 10 ether

availableEther = wethBalance + bufferedEther - unfinalizedStETH =  10090 ether

limitInDSM = 150
depositsAvailable = StakingRouter.getStakingModuleMaxDepositsCount(stakingModuleId, availableEther)
depositsAvailable = 315

maxDepositsCount = min(limitInDSM, depositsAvailable) = 150

amount = min(wethBalance, maxDepositsCount * 32 ether) = 100 ether
```

In the above example, the whole balance of **Vault** will be converted, but the ether in the buffer before the deposit will be enough to cover 150 deposits (4800 ETH). So, the converted ether would just stay in **Lido** and may go into other **Staking Modules**.

Recommendation

We recommend converting only the necessary amount of ether from **Vault** balance for deposit in **StakingModule**.

```
function convertAndDeposit(...) external onlyDelegateCall {
    uint256 wethBalance = IERC20(weth).balanceOf(address(this));
    uint256 unfinalizedStETH = withdrawalQueue.unfinalizedStETH();
    uint256 bufferedEther = ISteth(steth).getBufferedEther();

    if (bufferedEther < unfinalizedStETH)
        revert InvalidWithdrawalQueueState();
    uint256 maxDepositsCount = Math.min(
        IStakingRouter(depositSecurityModule.STAKING_ROUTER())
            .getStakingModuleMaxDepositsCount(
                stakingModuleId,
                wethBalance + bufferedEther - unfinalizedStETH
            ),
        depositSecurityModule.getMaxDeposits()
    );

    unit256 amount = 0
    uint256 availableEtherBeforeDeposit = bufferedEther - unfinalizedStETH;
    if (availableEtherBeforeDeposit < 32 * maxDepositsCount) {
        amount = Math.min(
            wethBalance,
            32 * maxDepositsCount - availableEtherBeforeDeposit
        );  // <-- Math.min() just for double check
    }

    ...

}
```

STATE
MIND