BLOCKSEC

Security Audit Report for mETH Protocol

Date: October 21, 2025 **Version:** 1.0

Contact: contact@blocksec.com

Contents

Report Manifest

Item	Description
Client	Mantle
Target	mETH Protocol

Version History

Version	Date	Description
1.0	October 21, 2025	First release

Signature

About BlockSec BlockSec focuses on the security of the blockchain ecosystem and collaborates with leading DeFi projects to secure their products. BlockSec is founded by top-notch security researchers and experienced experts from both academia and industry. They have published multiple blockchain security papers in prestigious conferences, reported several zero-day attacks of DeFi applications, and successfully protected digital assets that are worth more than 14 million dollars by blocking multiple attacks. They can be reached at Email, Twitter and Medium.

Chapter 1 Introduction

1.1 About Target Contracts

Information	Description
Type	Smart Contract
Language	Solidity
Approach	Semi-automatic and manual verification

The target of this audit is the code repository [1] of mETH Protocol of Mantle.

The Mantle Liquid Staking Platform (LSP) is a permissionless ETH liquid staking protocol deployed on Ethereum L1 and governed by Mantle Governance. It is a core product of the Mantle Ecosystem. Its receipt token, mETH, is a reward-accumulating ERC-20 token that represents staked ETH plus accrued rewards. mETH is designed to maximize utility, usable across DeFi applications on Ethereum L1, Mantle Network L2, and other centralized platforms. The contract architecture features a Staking contract as the main user interface, an UnstakeRequestsManager for tracking redemptions, and an Oracle system, validated by the OracleQuorumManager, to ensure data integrity. The ReturnsAggregator processes all staking rewards before they are compounded or used to fulfill unstake requests.

Note this audit only focuses on the smart contracts in the following directories/files:

- src/interfaces/IStaking.sol
- src/interfaces/IPauser.sol
- src/Staking.sol
- src/Pauser.sol
- src/liquidityBuffer/*

Other files are not within the scope of the audit. Additionally, all dependencies of the smart contracts within the audit scope are considered reliable in terms of both functionality and security, and are therefore not included in the audit scope.

The auditing process is iterative. Specifically, we would audit the commits that fix the discovered issues. If there are new issues, we will continue this process. The commit SHA values during the audit are shown in the following table. Our audit report is responsible for the code in the initial version (`Version 1`), as well as new code (in the following versions) to fix issues in the audit report. Code prior to and including the baseline version (`Version 0`), where applicable, is outside the scope of this audit and assumes to be reliable and secure.

Project	Version	Commit Hash
mETH Protocol	Version 0	9301723be80c6d67432f332544714c807e5ceb6b
	Version 1	630e7195f96e0ab2ac86543698905262ddb8346a
	Version 2	415408e4f8a250cfa16324890b7a23a514a6128c

[1] https://github.com/mantle-lsp/contracts

1.2 Disclaimer

This audit report does not constitute investment advice or a personal recommendation. It does not consider, and should not be interpreted as considering or having any bearing on, the potential economics of a token, token sale or any other product, service or other asset. Any entity should not rely on this report in any way, including for the purpose of making any decisions to buy or sell any token, product, service or other asset.

This audit report is not an endorsement of any particular project or team, and the report does not guarantee the security of any particular project. This audit does not give any warranties on discovering all security issues of the smart contracts, i.e., the evaluation result does not guarantee the nonexistence of any further findings of security issues. As one audit cannot be considered comprehensive, we always recommend proceeding with independent audits and a public bug bounty program to ensure the security of smart contracts.

The scope of this audit is limited to the code mentioned in Section **??**. Unless explicitly specified, the security of the language itself (e.g., the solidity language), the underlying compiling toolchain and the computing infrastructure are out of the scope.

1.3 Procedure of Auditing

We perform the audit according to the following procedure.

- **Vulnerability Detection** We first scan smart contracts with automatic code analyzers, and then manually verify (reject or confirm) the issues reported by them.
- **Semantic Analysis** We study the business logic of smart contracts and conduct further investigation on the possible vulnerabilities using an automatic fuzzing tool (developed by our research team). We also manually analyze possible attack scenarios with independent auditors to cross-check the result.
- **Recommendation** We provide some useful advice to developers from the perspective of good programming practice, including gas optimization, code style, and etc.

We show the main concrete checkpoints in the following.

1.3.1 Security Issues

* Access control
* Permission management
* Whitelist and blacklist mechanisms
* Initialization consistency
* Improper use of the proxy system
* Reentrancy
* Denial of Service (DoS)
* Untrusted external call and control flow
* Exception handling
* Data handling and flow
* Events operation

* Error-prone randomness
* Oracle security
* Business logic correctness
* Semantic and functional consistency
* Emergency mechanism
* Economic and incentive impact

1.3.2 Additional Recommendation

* Gas optimization
* Code quality and style

Note *The previous checkpoints are the main ones. We may use more checkpoints during the auditing process according to the functionality of the project.*

1.4 Security Model

To evaluate the risk, we follow the standards or suggestions that are widely adopted by both industry and academy, including OWASP Risk Rating Methodology [2] and Common Weakness Enumeration [3]. The overall *severity* of the risk is determined by *likelihood* and *impact*. Specifically, likelihood is used to estimate how likely a particular vulnerability can be uncovered and exploited by an attacker, while impact is used to measure the consequences of a successful exploit.

In this report, both likelihood and impact are categorized into two ratings, i.e., *high* and *low* respectively, and their combinations are shown in Table **??**.

Table 1.1: Vulnerability Severity Classification

Impact \ Likelihood	High	Low
High	High	Medium
Low	Medium	Low

Accordingly, the severity measured in this report are classified into three categories: **High**, **Medium**, **Low**. For the sake of completeness, **Undetermined** is also used to cover circumstances when the risk cannot be well determined.

Furthermore, the status of a discovered item will fall into one of the following five categories:

[2] https://owasp.org/www-community/OWASP_Risk_Rating_Methodology

[3] https://cwe.mitre.org/

- **Undetermined** No response yet.
- **Acknowledged** The item has been received by the client, but not confirmed yet.
- **Confirmed** The item has been recognized by the client, but not fixed yet.
- **Partially Fixed** The item has been confirmed and partially fixed by the client.
- **Fixed** The item has been confirmed and fixed by the client.

Chapter 2 Findings

In total, we found **six** potential security issues. Besides, we have **four** recommendations and **four** notes.

- Medium Risk: 2
- Low Risk: 4
- Recommendation: 4
- Note: 4

ID	Severity	Description	Category	Status
1	Medium	Non-atomic operations lead to miscalculation of available funds	Security Issue	Fixed
2	Medium	Unaccounted borrow and repay functionality	Security Issue	Fixed
3	Low	Potential donation attack inflates total controlled value	Security Issue	Confirmed
4	Low	Lack of validation for the `AllocationCap` parameter	Security Issue	Fixed
5	Low	Uncorrectable cumulative drawdown leads to incorrect exchange rate	Security Issue	Fixed
6	Low	Improper permission assignment in the contract `PositionManager`	Security Issue	Fixed
7	-	Lack of duplicate checks on `managerAddress` in the `addPositionManager()` function	Recommendation	Fixed
8	-	Non Zero Address Checks	Recommendation	Confirmed
9	-	Lack of amount validation in the function `depositETH()`	Recommendation	Confirmed
10	-	Lack of role granting and revoking in the function `setLiquidityBuffer()`	Recommendation	Fixed
11	-	The interest should be periodically collected and transferred	Note	-
12	-	Potential Centralization Risks	Note	-
13	-	Potential collateral risk relying on `Aave`	Note	-
14	-	Potential liquidity insufficiency in `Aave` pool	Note	-

The details are provided in the following sections.

2.1 Security Issue

2.1.1 Non-atomic operations lead to miscalculation of available funds

Severity Medium

Status Fixed in `Version 2`

Introduced by `Version 1`

Description The contract LiquidityBuffer's function getAvailableBalance() is intended to return the amount of available Ether in the contract, where variable totalFundsReceived is increased in the function _receiveETHFromStaking() and variable totalFundsReturned is increased in the function _returnETHToStaking().

Under normal conditions, atomic composed functions like depositETH(), withdrawAndReturn(), and claimInterestAndTopUp() ensure that deposited Ether is immediately allocated and returned Ether is immediately sent back to the contract Staking. However, the existence of single-step functions such as withdrawETHFromManager(), returnETHToStaking(), and topUpInterestToStaking(), which can be invoked independently, allows for a non-atomic sequence of operations.

Specifically, there are three cases:

1. The LIQUIDITY_MANAGER_ROLE withdraws ETH via withdrawETHFromManager(), leaving it in the contract LiquidityBuffer, and subsequently the INTEREST_TOPUP_ROLE calls the function topUpInterestToStaking() to transfer that balance to the contract Staking without incrementing totalFundsReturned. This flawed logic leads to an inflated value being returned by the function getAvailableBalance().

2. The INTEREST_TOPUP_ROLE withdraws interest via claimInterestFromManager(), leaving it in the contract LiquidityBuffer, and subsequently the LIQUIDITY_MANAGER_ROLE calls the function returnETHToStaking() to transfer that balance to the contract Staking with incorrectly incrementing totalFundsReturned. This flawed logic leads to a deflated value being returned by the function getAvailableBalance().

3. The INTEREST_TOPUP_ROLE withdraws interest via claimInterestFromManager(), leaving it in the contract LiquidityBuffer, and subsequently the LIQUIDITY_MANAGER_ROLE calls the function allocateETHToManager() to resupply. After that, the LIQUIDITY_MANAGER_ROLE invokes the function withdrawAndReturn() by taking interest as part of totalFundsReturned. This flawed logic leads to a deflated value being returned by the function getAvailableBalance().

This ultimately results in an incorrect ETH-to-mETH conversion ratio within the contract Staking, which compromises the system's token valuation integrity.

```
function getAvailableBalance() public view returns (uint256) {
    return totalFundsReceived - totalFundsReturned;
}
```

Listing 2.1: src/liquidityBuffer/LiquidityBuffer.sol

```
function depositETH() external payable onlyRole(LIQUIDITY_MANAGER_ROLE) {
    _receiveETHFromStaking(msg.value);
    if (shouldExecuteAllocation) {
        _allocateETHToManager(defaultManagerId, msg.value);
    }
}

function withdrawAndReturn(uint256 managerId, uint256 amount) external onlyRole(
     LIQUIDITY_MANAGER_ROLE) {
    _withdrawETHFromManager(managerId, amount);
    _returnETHToStaking(amount);
}
```

```solidity

function allocateETHToManager(uint256 managerId, uint256 amount) external onlyRole(LIQUIDITY_MANAGER_ROLE) {
    _allocateETHToManager(managerId, amount);
}

function withdrawETHFromManager(uint256 managerId, uint256 amount) external onlyRole(LIQUIDITY_MANAGER_ROLE) {
    _withdrawETHFromManager(managerId, amount);
}

function returnETHToStaking(uint256 amount) external onlyRole(LIQUIDITY_MANAGER_ROLE) {
    _returnETHToStaking(amount);
}
```

Listing 2.2: src/liquidityBuffer/LiquidityBuffer.sol

```solidity
function claimInterestFromManager(uint256 managerId, uint256 minAmount) external onlyRole(INTEREST_TOPUP_ROLE) returns (uint256) {
    uint256 amount = _claimInterestFromManager(managerId);
    if (amount < minAmount) {
        revert LiquidityBuffer__InsufficientBalance();
    }
    return amount;
}

function topUpInterestToStaking(uint256 amount) external onlyRole(INTEREST_TOPUP_ROLE) returns (uint256) {
    if (address(this).balance < amount) {
        revert LiquidityBuffer__InsufficientBalance();
    }
    _topUpInterestToStakingAndCollectFees(amount);
    return amount;
}

function claimInterestAndTopUp(uint256 managerId, uint256 minAmount) external onlyRole(INTEREST_TOPUP_ROLE) returns (uint256) {
    uint256 amount = _claimInterestFromManager(managerId);
    if (amount < minAmount) {
        revert LiquidityBuffer__InsufficientBalance();
    }
    _topUpInterestToStakingAndCollectFees(amount);

    return amount;
}
```

Listing 2.3: src/liquidityBuffer/LiquidityBuffer.sol

Impact This ultimately results in an incorrect ETH-to-mETH conversion ratio within the contract `Staking`, which compromises the system's token valuation integrity.

Suggestion Revise the logic accordingly.

2.1.2 Unaccounted borrow and repay functionality

Severity Medium

Status Fixed in Version 2

Introduced by Version 1

Description The contract PositionManager provides the functions borrow() and repay() that allow the EXECUTOR_ROLE to borrow Ether using WETH collateral in the lending pool. An ambiguity exists because the resulting debt and required interest payments are not included in the contract LiquidityBuffer's balance calculation via the function getAvailableBalance(), nor do they involve any direct fund movement with the contracts LiquidityBuffer or Staking.

Specifically, the system lacks clarity on the identity of the recipient of the Ether borrowed through the function borrow() and the source of the interest payments necessary for the function repay(), which creates an opaque operational dependency.

Furthermore, a potential failure to execute the function repay() promptly can lead to the WETH collateral being liquidated. This liquidation risk results in a reduced return from the contract LiquidityBuffer's function getInterestAmount() and prevents the function withdrawAndReturn() from executing for the full requested amount, which ultimately results in loss for the users.

```
function repay(uint256 amount) external payable override onlyRole(EXECUTOR_ROLE) {
    require(msg.value > 0, 'No ETH sent');

    // Get debt token to check current debt
    address debtToken = pool.getReserveVariableDebtToken(address(weth));
    uint256 currentDebt = IERC20(debtToken).balanceOf(address(this));

    uint256 repayAmount = amount;
    if (amount == type(uint256).max) {
        repayAmount = currentDebt;
    }

    // Use the smaller of the two amounts
    if (repayAmount > currentDebt) {
        repayAmount = currentDebt;
    }

    require(msg.value >= repayAmount, 'Insufficient ETH for repayment');

    // Wrap ETH to WETH
    weth.deposit{value: repayAmount}();

    // Repay the debt
    pool.repay(
        address(weth),
        repayAmount,
        uint256(DataTypes.InterestRateMode.VARIABLE),
        address(this)
    );

```

```
        // Refund excess ETH
        if (msg.value > repayAmount) {
            _safeTransferETH(msg.sender, msg.value - repayAmount);
        }

        emit Repay(msg.sender, repayAmount, uint256(DataTypes.InterestRateMode.VARIABLE));
    }

    function borrow(uint256 amount, uint16 referralCode) external override onlyRole(EXECUTOR_ROLE)
         {
        require(amount > 0, 'Invalid amount');

        // Borrow WETH from pool
        pool.borrow(
            address(weth),
            amount,
            uint256(DataTypes.InterestRateMode.VARIABLE),
            referralCode,
            address(this)
        );

        // Unwrap WETH to ETH
        weth.withdraw(amount);

        // Transfer ETH to caller safely
        _safeTransferETH(msg.sender, amount);

        emit Borrow(msg.sender, amount, uint256(DataTypes.InterestRateMode.VARIABLE));
    }
```

Listing 2.4: src/liquidityBuffer/PositionManager.sol

Impact This could cause loss for the users.

Suggestion Revise the logic accordingly.

2.1.3 Potential donation attack inflates total controlled value

Severity Low

Status Confirmed

Introduced by `Version 1`

Description The contract `Staking`'s function `totalControlled()` includes interest calculated from the contract `LiquidityBuffer`, which itself relies on the balance of aWETH retrieved by the function `getUnderlyingBalance()` in the contract `PositionManager`. The vulnerability arises because a malicious user can donate `aWETH` directly to the contract `PositionManager`, which artificially inflates its underlying balance and consequently the interest calculation returned by the function `getInterestAmount()`. Although the contract `Staking` has safeguards like a `minimumStakeBound` for the function `stake()`, an attacker can bypass this defense in a newly deployed contract by first staking a minimal amount and then performing an unstaking operation, which leaves the `mETH.totalSupply()` at 1 wei before the donation is executed. This sequence of actions allows the attacker to inflate the total value returned by the function `totalControlled()` relative to the small `mETH.totalSupply()`.

```
function getUnderlyingBalance() external view returns (uint256) {
    IERC20 aWETH = IERC20(pool.getReserveAToken(address(weth)));
    return aWETH.balanceOf(address(this));
}
```

Listing 2.5: src/liquidityBuffer/PositionManager.sol

```
function getInterestAmount(uint256 managerId) public view returns (uint256) {
    PositionManagerConfig memory config = positionManagerConfigs[managerId];
    // Get current underlying balance from position manager
    IPositionManager manager = IPositionManager(config.managerAddress);
    uint256 currentBalance = manager.getUnderlyingBalance();

    // Calculate interest as: current balance - allocated balance
    PositionAccountant memory accounting = positionAccountants[managerId];

    if (currentBalance > accounting.allocatedBalance) {
        return currentBalance - accounting.allocatedBalance;
    }

    return 0;
}
```

Listing 2.6: src/liquidityBuffer/LiquidityBuffer.sol

```
function topUp() external payable onlyRole(TOP_UP_ROLE) {
    unallocatedETH += msg.value;
}
```

Listing 2.7: src/Staking.sol

```
function totalControlled() public view returns (uint256) {
    OracleRecord memory record = oracle.latestRecord();
    uint256 total = 0;
    total += unallocatedETH;
    total += allocatedETHForDeposits;
    /// The total ETH deposited to the beacon chain must be decreased by the deposits processed by the off-chain
    /// oracle since it will be accounted for in the currentTotalValidatorBalance from that point onwards.
    total += totalDepositedInValidators - record.cumulativeProcessedDepositAmount;
    total += record.currentTotalValidatorBalance;
    total += liquidityBuffer.getAvailableBalance();
    total -= liquidityBuffer.cumulativeDrawdown();
    total += unstakeRequestsManager.balance();
    return total;
}
```

Listing 2.8: src/Staking.sol

```
function ethToMETH(uint256 ethAmount) public view returns (uint256) {
    // 1:1 exchange rate on the first stake.
    // Using `METH.totalSupply` over `totalControlled` to check if the protocol is in its bootstrap phase since
    // the latter can be manipulated, for example by transferring funds to the `ExecutionLayerReturnsReceiver`, and
    // therefore be non-zero by the time the first stake is made
    if (mETH.totalSupply() == 0) {
        return ethAmount;
    }

    // deltaMETH = (1 - exchangeAdjustmentRate) * (mETHSupply / totalControlled) * ethAmount
    // This rounds down to zero in the case of `(1 - exchangeAdjustmentRate) * ethAmount * mETHSupply <
    // totalControlled`.
    // While this scenario is theoretically possible, it can only be realised feasibly during the protocol's
    // bootstrap phase and if `totalControlled` and `mETHSupply` can be changed independently of each other. Since
    // the former is permissioned, and the latter is not permitted by the protocol, this cannot be exploited by an
    // attacker.
    return Math.mulDiv(
        ethAmount,
        mETH.totalSupply() * uint256(_BASIS_POINTS_DENOMINATOR - exchangeAdjustmentRate),
        totalControlled() * uint256(_BASIS_POINTS_DENOMINATOR)
    );
}
```

Listing 2.9: src/Staking.sol

Impact A malicious donation of `aWETH` can be used to inflate the `totalControlled()` value, which is a key component in the ETH-to-mETH conversion ratio, ultimately causing loss to new stakers.

Suggestion Revise the logic accordingly.

2.1.4 Lack of validation for the `AllocationCap` parameter

Severity Low

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the function `updatePositionManager()`, the contract allows the `POSITION_MANAGER_ROLE` to modify the `allocationCap` value of a `PositionManager`. However, the function `updatePositionManager()` lacks validation for `newAllocationCap`, to ensure that `newAllocationCap` is not less than the funds already allocated to an existing position.

```
function updatePositionManager(
    uint256 managerId,
    uint256 newAllocationCap,
```

```
        bool isActive
    ) external onlyRole(POSITION_MANAGER_ROLE) {
        if (managerId >= positionManagerCount) {
            revert LiquidityBuffer__ManagerNotFound();
        }

        PositionManagerConfig storage config = positionManagerConfigs[managerId];

        // Update total allocation capacity
        totalAllocationCapacity = totalAllocationCapacity - config.allocationCap + newAllocationCap
            ;

        config.allocationCap = newAllocationCap;
        config.isActive = isActive;

        emit ProtocolConfigChanged(
            this.updatePositionManager.selector,
            "updatePositionManager(uint256,uint256,bool)",
            abi.encode(managerId, newAllocationCap, isActive)
        );
    }
```

Listing 2.10: src/liquidityBuffer/LiquidityBuffer.sol

Impact This may cause DoS when allocating `Ether` to the `positionManager`.

Suggestion Add parameter validation logic in the function.

2.1.5 Uncorrectable cumulative drawdown leads to incorrect exchange rate

Severity Low

Status Fixed in `Version 2`

Introduced by `Version 1`

Description The return value of the contract `Staking`'s function `totalControlled()` is for calculating the mETH-to-Ether exchange rate, where it subtracts the `cumulativeDrawdown` to reflect losses from the `LiquidityBuffer`. The vulnerability arises because the function `addCumulativeDrawdown()` is the sole method that can modify the `cumulativeDrawdown` state variable, yet its design only permits increasing the value. This contradiction means that if the `DRAWDOWN_MANAGER_ROLE` mistakenly adds an incorrect or excessive amount to the `cumulativeDrawdown` value, they have no corresponding function to correct or decrease the figure. This inability to rectify an over-reported loss means that the total funds controlled by the protocol will be persistently understated, which directly affects the exchange rate mechanism.

```
    function totalControlled() public view returns (uint256) {
        OracleRecord memory record = oracle.latestRecord();
        uint256 total = 0;
        total += unallocatedETH;
        total += allocatedETHForDeposits;
        /// The total ETH deposited to the beacon chain must be decreased by the deposits processed
             by the off-chain
```

```
        /// oracle since it will be accounted for in the currentTotalValidatorBalance from that
            point onwards.
        total += totalDepositedInValidators - record.cumulativeProcessedDepositAmount;
        total += record.currentTotalValidatorBalance;
        total += liquidityBuffer.getAvailableBalance();
        total -= liquidityBuffer.cumulativeDrawdown();
        total += unstakeRequestsManager.balance();
        return total;
    }
```

Listing 2.11: src/Staking.sol

```
    function addCumulativeDrawdown(uint256 drawdownAmount) external onlyRole(DRAWDOWN_MANAGER_ROLE)
         {
        cumulativeDrawdown += drawdownAmount;

        emit ProtocolConfigChanged(
            this.addCumulativeDrawdown.selector,
            "addCumulativeDrawdown(uint256)",
            abi.encode(drawdownAmount)
        );
    }
```

Listing 2.12: src/liquidityBuffer/LiquidityBuffer.sol

Impact This design may lead to a persistent exchange rate error between mETH and Ether.

Suggestion Revise the logic accordingly.

2.1.6 Improper permission assignment in the contract PositionManager

Severity Low

Status Fixed in Version 2

Introduced by Version 1

Description In the contract PositionManager, any address with the EXECUTOR_ROLE can invoke the functions deposit(), withdraw(), repay(), and borrow(). Although the LiquidityBuffer contract is granted the EXECUTOR_ROLE, it is designed to only invoke deposit() and withdraw(). This implies that the functions repay() and borrow() are intended to be invoked by an EXECUTOR_ROLE holder other than the contract LiquidityBuffer.

This can create an accounting inconsistency in the contract LiquidityBuffer. For instance, if the contract LiquidityBuffer invokes the function deposit(), and a separate address with EXECUTOR_ROLE subsequently invokes the function withdraw(), the LiquidityBuffer's internal accounting will become incorrect. As a result, its accounting.allocatedBalance will not be reduced as expected, which will furthermore lead to an incorrect counting for interest.

```
    function deposit(uint16 referralCode) external payable override onlyRole(EXECUTOR_ROLE) {
        if (msg.value > 0) {
            // Wrap ETH to WETH
```

Listing 2.13: src/liquidityBuffer/PositionManager.sol

```
function withdraw(uint256 amount) external override onlyRole(EXECUTOR_ROLE) {
    require(amount > 0, 'Invalid amount');
```

Listing 2.14: src/liquidityBuffer/PositionManager.sol

```
function repay(uint256 amount) external payable override onlyRole(EXECUTOR_ROLE) {
    require(msg.value > 0, 'No ETH sent');
```

Listing 2.15: src/liquidityBuffer/PositionManager.sol

```
function borrow(uint256 amount, uint16 referralCode) external override onlyRole(EXECUTOR_ROLE)
    {
    require(amount > 0, 'Invalid amount');
```

Listing 2.16: src/liquidityBuffer/PositionManager.sol

Impact This design results in potential accounting inconsistencies within the contract `LiquidityBuffer`, leading to incorrect tracking of allocated balances and interest.

Suggestion The roles should be separated to ensure that only the contract `LiquidityBuffer` can call the functions `deposit()` and `withdraw()`.

2.2 Recommendation

2.2.1 Lack of duplicate checks on `managerAddress` in the `addPositionManager()` function

Status Fixed in `Version 2`

Introduced by `Version 1`

Description The function `addPositionManager()` enables administrators to register new position managers. However, the function `addPositionManager()` does not verify whether the `managerAddress` already exists in the current configuration. As a result, the same manager address could be added multiple times, leading to multiple IDs referencing the same `managerAddress`.

```
function addPositionManager(
    address managerAddress,
    uint256 allocationCap
) external onlyRole(POSITION_MANAGER_ROLE) returns (uint256 managerId) {
    managerId = positionManagerCount;
    positionManagerCount++;

    positionManagerConfigs[managerId] = PositionManagerConfig({
        managerAddress: managerAddress,
        allocationCap: allocationCap,
        isActive: true
    });
    positionAccountants[managerId] = PositionAccountant({
        allocatedBalance: 0,
        interestClaimedFromManager: 0
    });
```

```

    totalAllocationCapacity += allocationCap;
    emit ProtocolConfigChanged(
        this.addPositionManager.selector,
        "addPositionManager(address,uint256)",
        abi.encode(managerAddress, allocationCap)
    );
}
```

Listing 2.17: src/liquidityBuffer/LiquidityBuffer.sol

Suggestion Ensure that duplicate additions are prevented.

2.2.2 Non Zero Address Checks

Status Confirmed

Introduced by Version 1

Description In function `initialize()` and `addPositionManager()`, several address variables (e.g., `init.weth`, `managerAddress`) are not checked to ensure they are not zero. It is recommended to add such checks to prevent potential misoperations.

```
function initialize(Init memory init) external initializer {
    __AccessControlEnumerable_init();

    weth = init.weth;
    pool = init.pool;
    liquidityBuffer = init.liquidityBuffer;

    // Set up roles
    _grantRole(DEFAULT_ADMIN_ROLE, init.admin);
    _grantRole(MANAGER_ROLE, init.manager);
    _grantRole(EXECUTOR_ROLE, address(init.liquidityBuffer));

    // Approve pool to spend WETH
    weth.approve(address(pool), type(uint256).max);
}
```

Listing 2.18: src/liquidityBuffer/PositionManager.sol

```
function addPositionManager(
    address managerAddress,
    uint256 allocationCap
) external onlyRole(POSITION_MANAGER_ROLE) returns (uint256 managerId) {
    managerId = positionManagerCount;
    positionManagerCount++;

    positionManagerConfigs[managerId] = PositionManagerConfig({
        managerAddress: managerAddress,
        allocationCap: allocationCap,
        isActive: true
    });
    positionAccountants[managerId] = PositionAccountant({
```

```
            allocatedBalance: 0,
            interestClaimedFromManager: 0
        });

        totalAllocationCapacity += allocationCap;
        emit ProtocolConfigChanged(
            this.addPositionManager.selector,
            "addPositionManager(address,uint256)",
            abi.encode(managerAddress, allocationCap)
        );
    }
```

Listing 2.19: src/liquidityBuffer/LiquidityBuffer.sol

Suggestion Add non-zero address checks accordingly.

2.2.3 Lack of amount validation in the function `depositETH()`

Status Confirmed

Introduced by `Version 1`

Description The function `depositETH()` does not validate whether `msg.value` is greater than zero, allowing zero-value transactions to proceed. This results in unnecessary downstream logic execution, including invocations of `_receiveETHFromStaking()` and `_allocateETHToManager()`, potentially emitting meaningless zero-value events. It is recommended to implement a check to prevent zero-value deposits.

```
    function depositETH() external payable onlyRole(LIQUIDITY_MANAGER_ROLE) {
        _receiveETHFromStaking(msg.value);
        if (shouldExecuteAllocation) {
            _allocateETHToManager(defaultManagerId, msg.value);
        }
    }
```

Listing 2.20: src/liquidityBuffer/LiquidityBuffer.sol

```
    function _allocateETHToManager(uint256 managerId, uint256 amount) internal {
        if (pauser.isLiquidityBufferPaused()) {
            revert LiquidityBuffer__Paused();
        }

        if (managerId >= positionManagerCount) revert LiquidityBuffer__ManagerNotFound();
        // check available balance
        if (address(this).balance < amount) revert LiquidityBuffer__InsufficientBalance();

        // check position manager is active
        PositionManagerConfig memory config = positionManagerConfigs[managerId];
        if (!config.isActive) revert LiquidityBuffer__ManagerInactive();
        // check allocation cap
        PositionAccountant storage accounting = positionAccountants[managerId];
        if (accounting.allocatedBalance + amount > config.allocationCap) {
            revert LiquidityBuffer__ExceedsAllocationCap();
```

```
    }

    // Update accounting BEFORE external call (Checks-Effects-Interactions pattern)
    accounting.allocatedBalance += amount;
    totalAllocatedBalance += amount;
    emit ETHAllocatedToManager(managerId, amount);

    // deposit to position manager AFTER state updates
    IPositionManager manager = IPositionManager(config.managerAddress);
    manager.deposit{value: amount}(0);
}
```

Listing 2.21: src/liquidityBuffer/LiquidityBuffer.sol

```
function _receiveETHFromStaking(uint256 amount) internal {
    totalFundsReceived += amount;
    emit ETHReceivedFromStaking(amount);
}
```

Listing 2.22: src/liquidityBuffer/LiquidityBuffer.sol

Suggestion It is recommended to implement a check to prevent zero-value deposits.

2.2.4 Lack of role granting and revoking in the function `setLiquidityBuffer()`

Status Fixed in `Version 2`

Introduced by `Version 1`

Description The function `setLiquidityBuffer()` updates the state variable `liquidityBuffer` but does not revoke the `EXECUTOR_ROLE` from the old address and grant the `EXECUTOR_ROLE` to the new address.

```
function setLiquidityBuffer(address _liquidityBuffer) external onlyRole(MANAGER_ROLE) {
    liquidityBuffer = ILiquidityBuffer(_liquidityBuffer);
}
```

Listing 2.23: src/liquidityBuffer/PositionManager.sol

Suggestion It is recommended to update the role assignments when modifying the `liquidityBuffer` address.

2.3 Note

2.3.1 The interest should be periodically collected and transferred

Introduced by `Version 1`

Description The interest collected from the `positionManager` should be periodically collected and transferred to the contract `Staking`. Failure to do so will result in the `unallocatedETH` value within the contract `Staking` being understated, consequently impacting the correct mETH-to-ETH conversion ratio. Moreover, infrequent updates may cause the mETH-to-ETH conversion ratio to change sharply, introducing potential arbitrage opportunities.

```
function _topUpInterestToStakingAndCollectFees(uint256 amount) internal {
    if (pauser.isLiquidityBufferPaused()) {
        revert LiquidityBuffer__Paused();
    }
    uint256 fees = Math.mulDiv(feesBasisPoints, amount, _BASIS_POINTS_DENOMINATOR);
    uint256 topUpAmount = amount - fees;
    stakingContract.topUp{value: topUpAmount}();
    totalInterestToppedUp += topUpAmount;
    emit InterestToppedUp(topUpAmount);

    if (fees > 0) {
        Address.sendValue(feesReceiver, fees);
        totalFeesCollected += fees;
        emit FeesCollected(fees);
    }
}
```

Listing 2.24: src/liquidityBuffer/LiquidityBuffer.sol

```
function topUp() external payable onlyRole(TOP_UP_ROLE) {
    unallocatedETH += msg.value;
}
```

Listing 2.25: src/Staking.sol

2.3.2 Potential Centralization Risks

Introduced by `Version 1`

Description In this project, several privileged roles (e.g., `EXECUTOR_ROLE`) can conduct sensitive operations, which introduces potential centralization risks. For example, `EXECUTOR_ROLE` can borrow `Ether` using the `WETH` collateral of the positionManager based on the protocol. If the private keys of the privileged accounts are lost or maliciously exploited, it could pose a significant risk to the protocol.

2.3.3 Potential collateral risk relying on `Aave`

Introduced by `Version 1`

Description The system relies on `WETH` being deposited as collateral into the external `Aave` pool, which exposes the protocol to standard lending market risks. The underlying contradiction is that the collateral is subject to external market forces, which are beyond the control of the protocol. If the market experiences sudden and severe price volatility concerning `WETH`, the `Aave` pool could potentially incur bad debt. This inherent market risk impacts the protocol's core assets.

2.3.4 Potential liquidity insufficiency in `Aave` pool

Introduced by `Version 1`

Description The protocol's withdrawal mechanism is dependent on retrieving `WETH` from the external `Aave` pool to successfully fulfill user requests for funds, which introduces external liquidity risk. The inherent contradiction is that while the protocol supplies assets to the `Aave` pool, the availability of these assets for withdrawal is controlled by external borrowers. If the `Aave` pool's supplied `WETH` is extensively borrowed by other participants, the contract `LiquidityBuffer` may face an inability to retrieve sufficient `WETH` in a timely manner.

BOOST WEB3 THROUGH NEXT-GENERATION SECURITY & USABILITY INNOVATIONS