Security Review Report for Mantle

October 2025

Table of Contents

1. About Hexens

Hexens is a pioneering cybersecurity firm dedicated to establishing robust security standards for Web3 infrastructure, driving secure mass adoption through innovative protection technology and frameworks. As an industry elite experts in blockchain security, we deliver comprehensive audit solutions across specialized domains, including infrastructure security, Zero Knowledge Proof, novel cryptography, DeFi protocols, and NFTs.

Our methodology combines industry-standard security practices combined with unique methodology of two teams per audit, continuously advancing the field of Web3 security. This innovative approach has earned us recognition from industry leaders.

Since our founding in 2021, we have built an exceptional portfolio of enterprise clients, including major blockchain ecosystems and Web3 platforms.

2. Executive Summary

This audit focused on the updates to the Mantle Liquid Staking Platform contracts. The purpose of these changes is to allocate a portion of the protocol's ETH holdings to Aave's ETH mainnet markets, enabling the protocol to more efficiently support a larger volume of redemptions without compromising its economic stability.

Our security review spanned one week and included a thorough examination of all contracts modified in this update.

During the audit, we identified two medium-severity issues, along with one low-severity and three informational findings.

All identified issues were remediated by the development team and subsequently verified by our auditors.

Overall, this audit has contributed to an improvement in both the protocol's security posture and the overall quality of its codebase.

3. Security Review Details

▪ Review Led by

Trung Dinh, Lead Security Researcher

▪ Scope

The analyzed resources are located on:

🔗 https://github.com/mantle-lsp/contracts/pull/17

📌 Commit: `630e7195f96e0ab2ac86543698905262ddb8346a`

The issues described in this report were fixed in the following commit:

🔗 https://github.com/mantle-lsp/contracts/pull/17

📌 Commit: `9301723be80c6d67432f332544714c807e5ceb6b`

▪ Changelog

13 October 2025	Audit start
20 October 2025	Initial report
21 October 2025	Revision received
22 October 2025	Final report

4. Severity Structure

The vulnerability severity is calculated based on two components:

1. Impact of the vulnerability
2. Probability of the vulnerability

Impact	Probability			
	Rare	Unlikely	Likely	Very likely
Low	Low	Low	Medium	Medium
Medium	Low	Medium	Medium	High
High	Medium	Medium	High	Critical
Critical	Medium	High	Critical	Critical

- **Severity Characteristics**

Smart contract vulnerabilities can range in severity and impact, and it's important to understand their level of severity in order to prioritize their resolution. Here are the different types of severity levels of smart contract vulnerabilities:

Critical

Vulnerabilities that are highly likely to be exploited and can lead to catastrophic outcomes, such as total loss of protocol funds, unauthorized governance control, or permanent disruption of contract functionality.

High

Vulnerabilities that are likely to be exploited and can cause significant financial losses or severe operational disruptions, such as partial fund theft or temporary asset freezing.

Medium	Vulnerabilities that may be exploited under specific conditions and result in moderate harm, such as operational disruptions or limited financial impact without direct profit to the attacker.
Low	Vulnerabilities with low exploitation likelihood or minimal impact, affecting usability or efficiency but posing no significant security risk.
Informational	Issues that do not pose an immediate security risk but are relevant to best practices, code quality, or potential optimizations.

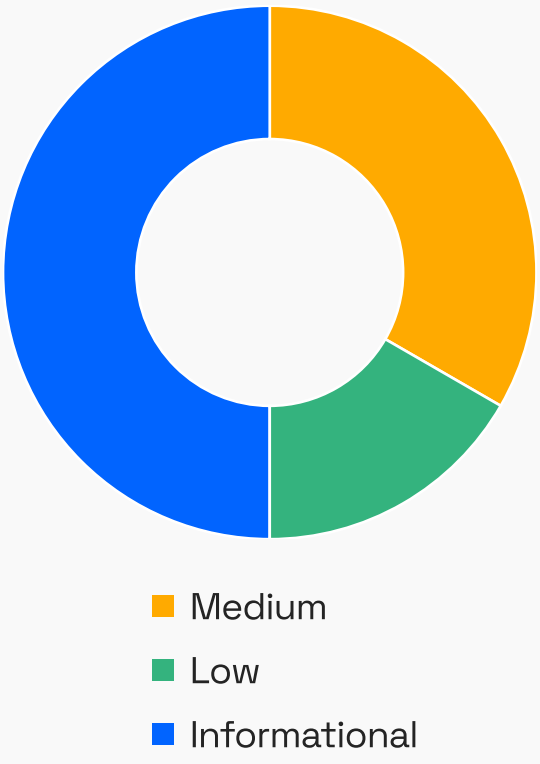
- **Issue Symbolic Codes**

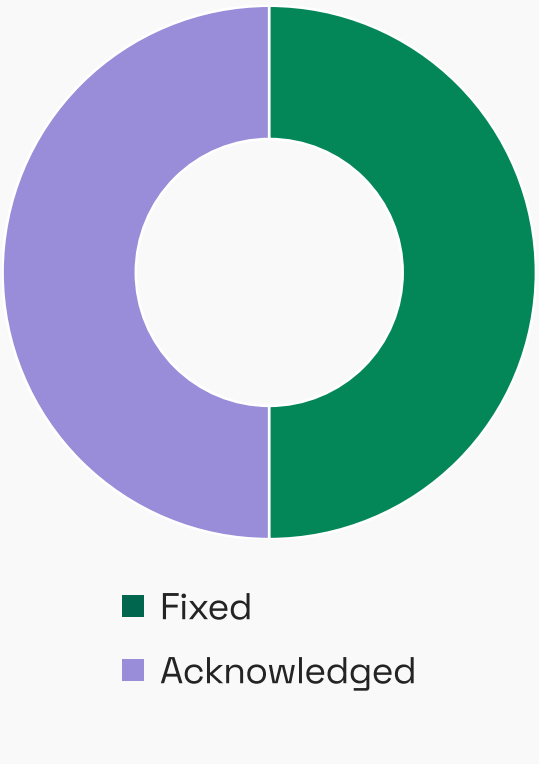
Each identified and validated issue is assigned a unique symbolic code during the security research stage.

Due to the structure of the vulnerability reporting flow, some rejected issues may be missing.

5. Findings Summary

Severity	Number of findings
Critical	0
High	0
Medium	2
Low	1
Informational	3
Total:	**6**





6. Weaknesses

This section contains the list of discovered weaknesses.

MANT4-3 | There is no function to claim Aave Incentives

Acknowledged

Severity:	Probability:	Impact:
Medium	Unlikely	Medium

Path:

src/liquidityBuffer/PositionManager.sol

Description:

Aave offers incentives - such as staking or liquidity mining rewards (see: https://aave.com/docs/primitives/incentives) - to users who supply assets to the protocol. These rewards are typically distributed as additional tokens (e.g., AAVE or other governance tokens) and can be claimed by users through Aave's incentive mechanisms.

However, the current **PositionManager** contract lacks functionality to claim these incentives. This omission prevents users from fully benefiting from their supplied assets on Aave.

Remediation:

To resolve this, we should introduce a function that enables users to claim their Aave incentives. This would require integrating with Aave's **Incentives Controller** or **Rewards Distributor** contracts.

Commentary from the client:

"I think we can keep this because there is no incentive for the users that supply ETH."

MANT4-4 | ETH can still be allocated to the Liquidity Buffer even when it is paused

Fixed

Severity:	Probability:	Impact:
Medium	Unlikely	Medium

Path:

src/liquidityBuffer/LiquidityBuffer.sol#L317-L322

Description:

When the **LiquidityBuffer** contract is paused (**pauser.isLiquidityBufferPaused() == true**), ETH transferred from the **Staking** contract should **not** be allocated to the **LiquidityBuffer**, and the allocation transaction should revert.

However, there exists a scenario where the **Staking** contract is still able to allocate ETH to the **LiquidityBuffer** despite the pause condition.

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

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

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

    ...
}
```

The pause check is only executed inside **_allocateETHToManager()**, which runs only if **shouldExecuteAllocation** is set to **true**.

Therefore, if **shouldExecuteAllocation** is **false** and the **LiquidityBuffer** contract is paused, the **depositETH()** call will still succeed, and ETH will continue to be allocated.

Additionally, once the ETH is in the **LiquidityBuffer**, it cannot be returned to the **Staking** contract while the contract remains paused, since calling **returnETHToStaking()** will revert (at line 444).

Remediation:

Consider adding a pause check within the function **_receiveETHFromStaking()**

```
 function _receiveETHFromStaking(uint256 amount) internal {
++     if (pauser.isLiquidityBufferPaused()) {
++         revert LiquidityBuffer__Paused();
++     }

       totalFundsReceived += amount;
       emit ETHReceivedFromStaking(amount);
 }
```

MANT4-8 | Attacker can arbitrage before slashing

Acknowledged

Severity:	Probability:	Impact:
Low	Rare	Medium

Description:

Function **Staking._unstakeRequest()** handles a user's unstake request by transferring the corresponding mETH to the staking contract. Within this function, it calculates the amount of ETH to be withdrawn based on the current exchange rate.

This calculated amount is then stored in the request as **ethRequested**.

```
function _unstakeRequest(uint128 methAmount, uint128 minETHAmount) internal returns (uint256) {
  if (pauser.isUnstakeRequestsAndClaimsPaused()) {
      revert Paused();
  }

  if (methAmount < minimumUnstakeBound) {
      revert MinimumUnstakeBoundNotSatisfied();
  }

  uint128 ethAmount = uint128(mETHToETH(methAmount));
  if (ethAmount < minETHAmount) {
      revert UnstakeBelowMinimumETHAmount(ethAmount, minETHAmount);
  }

  uint256 requestID =
      unstakeRequestsManager.create({requester: msg.sender, mETHLocked: methAmount, ethRequested:
ethAmount});
  emit UnstakeRequested({id: requestID, staker: msg.sender, ethAmount: ethAmount, mETHLocked:
methAmount});

  SafeERC20Upgradeable.safeTransferFrom(mETH, msg.sender, address(unstakeRequestsManager),
methAmount);

  return requestID;
}
```

After that, the user can claim their unstaked ETH by calling the **claim()** function - they will receive the exact **ethRequested** amount calculated at the time of their unstake request.

```
function claim(uint256 requestID, address requester) external onlyStakingContract {
  UnstakeRequest memory request = _unstakeRequests[requestID];
 ...

  delete _unstakeRequests[requestID];
  totalClaimed += request.ethRequested;

  emit UnstakeRequestClaimed({
      id: requestID,
      requester: requester,
      mETHLocked: request.mETHLocked,
      ethRequested: request.ethRequested,
      cumulativeETHRequested: request.cumulativeETHRequested,
      blockNumber: request.blockNumber
  });

  // Claiming the request burns the locked mETH tokens from this contract.
  // Note that it is intentional that burning happens here rather than at unstake time.
  // Please see the docs folder for more information.
  mETH.burn(request.mETHLocked);

  Address.sendValue(payable(requester), request.ethRequested);
}
```

However, this raises a concern: if a slashing event occurs between the unstake request and the claim, causing the mETH price to drop, the user still receives the fixed **ethRequested** amount.

An attacker could exploit this by monitoring transactions that trigger slashing and executing their claim before the slashing occurs, allowing them to withdraw more ETH than they should.

Remediation:

If the exchange rate decreases by the time of claiming, the amount of ETH the user receives should be recalculated. In other words, the claimable ETH should be the lesser value between the amounts computed using the exchange rate at the time of the unstake request and at the time of the claim.

Commentary from the client:

“It's protocol design. When users choose to unstake, they give up future rewards and risks. Fixed exchange rate ensures user certainty.”

MANT4-2 | Restrict topUpInterestToStaking() to Only Transfer Claimed Interest

Fixed

Severity:	Probability:	Impact:
Informational	Rare	Informational

Description:

The yield collected from Aave within the position managers can first be held in the buffer contract before being transferred back to the staking contract via **topUpInterestToStaking()**. When executed, this function increases the unallocated ETH balance on the staking side.

However, the current implementation of **topUpInterestToStaking()** allows any ETH balance held by the buffer - including allocated funds - to be sent back, instead of restricting the transfer to only the actual earned interest. This can lead to incorrect increases in the staking contract's unallocated ETH, resulting in double accounting since the buffer's allocated funds are already counted as part of the unallocated ETH.

Example scenario:

1. The staking contract calls **LiquidityBuffer.depositETH()** with **shouldExecuteAllocation = false,** resulting in:

```
LiquidityBuffer.totalFundsReceived = 10 ETH
```

   → 10 ETH is transferred from the staking contract to the liquidity buffer.

2. The function **LiquidityBuffer.topUpInterestToStaking()** is then called with **amount = 10 ETH**, which sets:

```
Staking.unallocatedETH = 10 ETH
```

After step 2, the LiquidityBuffer contract no longer holds any ETH, yet the function **getAvailableBalance()** still returns:

```
totalFundsReceived - totalFundsReturned = 10 - 0 = 10 ETH
```

This results in **double accounting** of 10 ETH when **Staking.totalControlled()** is called.

Remediation:

Update the **topUpInterestToStaking()** function so that it only transfers the actual yield collected from the position managers. This can be achieved in one of the following ways:

1. **Restrict the transfer amount** - ensure the function cannot send more than the currently claimable interest.
2. **Simplify the interface** - remove the **amount** parameter and automatically transfer the total claimable interest minus any amount already sent.

MANT4-6 | Redundant borrow and repay functions in PositionManager

Fixed

Severity:	Probability:	Impact:
Informational	Rare	Informational

Path:

src/liquidityBuffer/PositionManager.sol#L110-L167

Description:

The **PositionManager.borrow()** and **PositionManager.repay()** functions are restricted to the **EXECUTOR_ROLE**, which is assigned to the **LiquidityBuffer** contract. However, these functions are never called within the **LiquidityBuffer** contract, making them redundant and unused.

```
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));
}
```

Remediation:

Remove the unused **borrow()** and **repay()** functions from the **PositionManager** contract to simplify the codebase and reduce potential maintenance overhead.

MANT4-7 | Redundant withdrawal handling in PositionManager contract

Acknowledged

Severity:	Probability:	Impact:
Informational	Rare	Informational

Path:

src/liquidityBuffer/PositionManager.sol#L91-L94

Description:

In the **PositionManager.withdraw()** function, there is logic to handle the case where a user specifies **amount** equal to **type(uint256).max** to withdraw all of their WETH from Aave:

```
uint256 amountToWithdraw = amount;
if (amount == type(uint256).max) {
    amountToWithdraw = userBalance;
}
```

However, this function is restricted and can only be called by the **LiquidityBuffer** contract. There are only two scenarios where **LiquidityBuffer** triggers a withdrawal:

- In the **LiquidityBuffer._withdrawETHFromManager()** function, the **amount** parameter must be less than or equal to **accounting.allocatedBalance** (lines 427–430), making it impossible for **amount** to equal **type(uint256).max**.

```
function _withdrawETHFromManager(uint256 managerId, uint256 amount) internal {
    ...

    // Check sufficient allocation
    if (amount > accounting.allocatedBalance) {
        revert LiquidityBuffer__InsufficientAllocation();
    }

    ...

    manager.withdraw(amount);
}
```

- In the **LiquidityBuffer._claimInterestFromManager()** function, the withdrawal amount represents the accrued interest, calculated via **LiquidityBuffer.getInterestAmount()**:

```
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;
}
```

The return value of this function can never be **type(uint256).max**, since that would require **currentBalance** to be greater than or equal to **type(uint256).max**, which is impossible.

In conclusion, the handling of the **type(uint256).max** withdrawal case in **PositionManager.withdraw()** is redundant.

Remediation:

Consider removing the **type(uint256).max** withdrawal case in **PositionManager.withdraw()**.

Commentary from the client:

"I think we can keep this as an emergency call."


hexens
x
MANTLE
NETWORK