Security Audit Report for Mantle Fixed Yield Vault

Date: February 17, 2025 **Version:** 1.0

Contact: contact@blocksec.com

Contents

Report Manifest

Item	Description
Client	Mantle
Target	Mantle Fixed Yield Vault

Version History

Version	Date	Description
1.0	February 17, 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 of Mantle Fixed Yield Vault [1], which consists of a series of contracts designed to yield rewards (i.e., ETH tokens) for users who stake mETH tokens.

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.

Project	Version	Commit Hash
Mantle Fixed Yield Vault	`Version 1`	`9f68eb54143ceb3cc8da67cb351388b3abd38daa`
	`Version 2`	`62ae50458d408153b173a74cb94cdd1df74ab673`

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 1.1. 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] `https://github.com/mantle-lsp/fixed-yield-vault`

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 Software Security

* Reentrancy
* DoS
* Access control
* Data handling and data flow
* Exception handling
* Untrusted external call and control flow
* Initialization consistency
* Events operation
* Error-prone randomness
* Improper use of the proxy system

1.3.2 DeFi Security

* Semantic consistency
* Functionality consistency
* Permission management
* Business logic
* Token operation
* Emergency mechanism
* Oracle security
* Whitelist and blacklist
* Economic impact
* Batch transfer

1.3.3 NFT Security

* Duplicated item
* Verification of the token receiver
* Off-chain metadata security

1.3.4 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 1.1.

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 four categories:

- **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.
- **Fixed** The item has been confirmed and fixed by the client.

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

Chapter 2 Findings

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

- High Risk: 2
- Medium Risk: 1
- Low Risk: 1
- Recommendation: 5
- Note: 3

ID	Severity	Description	Category	Status
1	High	Lack of update for the variable `userLastActionTime`	DeFi Security	Fixed
2	High	Incorrect update for the variable `userLastActionTime`	DeFi Security	Fixed
3	Medium	Improper token handling in the function `recoverTokens()`	DeFi Security	Fixed
4	Low	Lack of checks for price validity	DeFi Security	Fixed
5	-	Add validation checks for inputs of the `initialize()` function	Recommendation	Fixed
6	-	Remove redundant require checks in the `executeWithdraw()` function	Recommendation	Fixed
7	-	Remove the redundant component in the `canExecuteWithdraw()` function's return value	Recommendation	Fixed
8	-	Refactor the for loop of the function `_calculateRewardRateBySnapshot()`	Recommendation	Fixed
9	-	Revise the misleading annotation for the `rewardRate` variable	Recommendation	Fixed
10	-	Potential centralization risks	Note	-
11	-	Potential DoS due to the increasement of the variable `rateHistory`	Note	-
12	-	Potential DoS due to insufficient supply of `rewardsToken`	Note	-

The details are provided in the following sections.

2.1 DeFi Security

2.1.1 Lack of update for the variable `userLastActionTime`

Severity High

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the function `requestWithdraw()` of the `BaseStakingRewards` contract, the reward accrues without updating the variable `userLastActionTime` for the user. This lack of update for the variable `userLastActionTime` allows users to claim extra rewards for their active balance.

```
function requestWithdraw(uint256 amount)
    external
    nonReentrant
    whenNotPaused
    returns (uint256)
{
    require(amount <= _balances[msg.sender] - _lockedBalances[msg.sender], "Insufficient withdrawable balance");

    userCumulativeRewards[msg.sender] = earned(msg.sender);
    _lockedBalances[msg.sender] += amount;

    return _requestWithdraw(amount);
}
```

Listing 2.1: src/BaseStakingRewards.sol

Impact Users can claim extra rewards for their active balance.

Suggestion Revise the code logic accordingly.

2.1.2 Incorrect update for the variable `userLastActionTime`

Severity High

Status Fixed in `Version 2`

Introduced by `Version 1`

Description The last action time (i.e., `userLastActionTime`) is used as the start time to define the reward interval. However, reward accumulation (i.e., `earned()` function) does not occur during withdrawal execution (e.g., `executeWithdraw()` function) while the `userLastActionTime` variable is updated.

```
function executeWithdraw(uint256 withdrawId)
    external
    nonReentrant
    whenNotPaused
{
    require(canExecuteWithdraw(msg.sender, withdrawId), "Withdrawal not ready or already executed");

    uint256 amount = _executeWithdraw(msg.sender, withdrawId);

    _totalSupply -= amount;
    _balances[msg.sender] -= amount;
    _lockedBalances[msg.sender] -= amount;
    userLastActionTime[msg.sender] = block.timestamp;

    SafeERC20.safeTransfer(stakingToken, msg.sender, amount);
    emit Withdrawn(msg.sender, amount);
}
```

Listing 2.2: src/BaseStakingRewards.sol

```
function _calculateRewards(address account) internal view returns (uint256) {
    uint256 activeBalance = _balances[account] - _lockedBalances[account];
    if (activeBalance == 0) return 0;

    return _calculateRewardRateBySnapshot(activeBalance, userLastActionTime[account], block.
        timestamp);
}
```

Listing 2.3: src/BaseStakingRewards.sol

Impact This design can lead to a loss of reward for users.

Suggestion Revise the code logic accordingly.

2.1.3 Improper token handling in the function `recoverTokens()`

Severity Medium

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the contract `BaseStakingRewards`, the `EMERGENCY_ROLE` is allowed to recover tokens, which are accidentally transferred to the contract, except the `stakingToken`. However, this design is improper because users could accidentally transfer their staking tokens to the contract. As a result, in this scenario, the `EMERGENCY_ROLE` is not able to recover users' staking tokens.

```
function recoverTokens(address tokenAddress, uint256 tokenAmount)
    external
    onlyRole(EMERGENCY_ROLE)
{
    require(tokenAddress != address(stakingToken), "Cannot withdraw the staking token");
    SafeERC20.safeTransfer(IERC20(tokenAddress), msg.sender, tokenAmount);
    emit Recovered(tokenAddress, tokenAmount);
}
```

Listing 2.4: src/BaseStakingRewards.sol

Impact The `EMERGENCY_ROLE` is not able to recover users' `stakingToken` via the `recoverTokens()` function.

Suggestion Revise the code logic accordingly.

2.1.4 Lack of checks for price validity

Severity Low

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the contract BaseStakingRewards, the price of mETH is queried from the RedStone protocol for the token conversion. However, there is a lack of validity check for the queried price. As a result, there might be a loss for the protocol or users if an expired price is used for the token conversion.

```
function mETHToETH(uint256 mETHAmount) public view returns (uint256 ethAmount) {
    int256 rate = priceFeed.latestAnswer();
    require(rate > 0, "Invalid exchange rate");
    return mETHAmount * uint256(rate) / (10 ** priceFeed.decimals());
}
```

Listing 2.5: src/BaseStakingRewards.sol

Impact An expired price could result in an incorrect token conversion between mETH and ETH, leading to a loss for either the protocol or users.

Suggestion Revise the code logic accordingly.

Feedback from the project The priceFeed contract has been removed. Rewards are now calculated directly based on users' staked balance of mETH tokens and the new APR to determine the amount (i.e., totalReward) of rewardsToken to be distributed. Moreover, the stakingToken and the rewardsToken are distinct.

2.2 Additional Recommendation

2.2.1 Add validation checks for inputs of the initialize() function

Status Fixed in Version 2

Introduced by Version 1

Description In the contract BaseStakingRewards, the inputs of the initialize() function are not properly validated. It is recommended to add non-zero checks for inputs of the initialize() function.

```
function initialize(
    address _owner,
    address _rewardsToken,
    address _stakingToken,
    address _priceFeedContractAddress
) public initializer {
    __ReentrancyGuard_init();
    __Pausable_init();
    __AccessControlEnumerable_init();
    __BaseRewardRateManager_init();

    _grantRole(DEFAULT_ADMIN_ROLE, _owner);
    _grantRole(PAUSER_ROLE, _owner);
    _grantRole(MANAGER_ROLE, _owner);
    _grantRole(EMERGENCY_ROLE, _owner);
    _grantRole(RATE_MANAGER_ROLE, _owner);

    rewardsToken = IERC20(_rewardsToken);
```

```
    stakingToken = IERC20(_stakingToken);
    priceFeed = IPriceFeed(_priceFeedContractAddress);
}
```

Listing 2.6: src/BaseStakingRewards.sol

Suggestion Add validation checks for inputs of the `initialize()` function.

2.2.2 Remove redundant require checks in the `executeWithdraw()` function

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the function `executeWithdraw()` of the contract `BaseStakingRewards`, the check (line 168) is redundant as it is invoked in the function `_executeWithdraw()`. It is recommended to remove the redundant check.

```
163 function executeWithdraw(uint256 withdrawId)
164     external
165     nonReentrant
166     whenNotPaused
167 {
168     require(canExecuteWithdraw(msg.sender, withdrawId), "Withdrawal not ready or already
            executed");
169
170     uint256 amount = _executeWithdraw(msg.sender, withdrawId);
171
172     _totalSupply -= amount;
173     _balances[msg.sender] -= amount;
174     _lockedBalances[msg.sender] -= amount;
175     userLastActionTime[msg.sender] = block.timestamp;
176
177     SafeERC20.safeTransfer(stakingToken, msg.sender, amount);
178     emit Withdrawn(msg.sender, amount);
179 }
```

Listing 2.7: src/BaseStakingRewards.sol

```
function _executeWithdraw(address user, uint256 withdrawId) internal returns (uint256) {
    require(canExecuteWithdraw(user, withdrawId), "Withdrawal not ready or already executed");

    WithdrawRequest storage request = withdrawRequests[user][withdrawId];
    uint256 amount = request.amount;
    request.executed = true;

    emit WithdrawExecuted(user, withdrawId, amount);
    return amount;
}
```

Listing 2.8: src/WithdrawRequestManager.sol

Suggestion Remove the redundant checks from the function `executeWithdraw()`.

2.2.3 Remove the redundant component in the `canExecuteWithdraw()` function's return value

Status Fixed in Version 2

Introduced by Version 1

Description In the `WithdrawRequest` struct, the `amount` variable is verified to be greater than 0 during the assignment process, but it remains unchanged afterward. Therefore, it is recommended to eliminate the redundant check (`request.amount` > 0) from the return condition in the `canExecuteWithdraw()` function.

```
function executeWithdraw(uint256 withdrawId)
    external
    nonReentrant
    whenNotPaused
{
    require(canExecuteWithdraw(msg.sender, withdrawId), "Withdrawal not ready or already executed");

    uint256 amount = _executeWithdraw(msg.sender, withdrawId);

    _totalSupply -= amount;
    _balances[msg.sender] -= amount;
    _lockedBalances[msg.sender] -= amount;
    userLastActionTime[msg.sender] = block.timestamp;

    SafeERC20.safeTransfer(stakingToken, msg.sender, amount);
    emit Withdrawn(msg.sender, amount);
}
```

Listing 2.9: src/BaseStakingRewards.sol

```
function _executeWithdraw(address user, uint256 withdrawId) internal returns (uint256) {
    require(canExecuteWithdraw(user, withdrawId), "Withdrawal not ready or already executed");

    WithdrawRequest storage request = withdrawRequests[user][withdrawId];
    uint256 amount = request.amount;
    request.executed = true;

    emit WithdrawExecuted(user, withdrawId, amount);
    return amount;
}
```

Listing 2.10: src/WithdrawRequestManager.sol

Suggestion Remove the redundant component in the return value of the `canExecuteWithdraw()` function.

2.2.4 Refactor the for loop of the function `_calculateRewardRateBySnapshot()`

Status Fixed in Version 2

Introduced by Version 1

Description In the contract `BaseRewardRateManager`, the `_calculateRewardRateBySnapshot()` function's for loop can be optimized via conducting a `break` instruction when `endTime` is less than or equal to `rateHistory[i].timestamp`. It is recommended to refactor the for loop for the gas optimization.

```
function _calculateRewardRateBySnapshot(uint256 balance, uint256 startTime, uint256 endTime)
     internal view returns (uint256) {
    // Return 0 if no rate history or if earliest rate is after endTime
    if (rateHistory.length == 0 || rateHistory[0].timestamp > endTime) return 0;

    // If startTime is after the latest rate, use the latest rate for calculation
    if (startTime >= rateHistory[rateHistory.length - 1].timestamp) {
        uint256 duration = endTime - startTime;
        return (balance * rateHistory[rateHistory.length - 1].rate * duration) / (365 days *
            10000);
    }

    uint256 totalRewards = 0;
    uint256 currentTime = startTime;

    for (uint256 i = 0; i < rateHistory.length; i++) {
        // skip past rate changes
        if (rateHistory[i].timestamp < startTime) continue;
        // get the rate for the period
        uint256 currentRate = i == 0 ? rateHistory[i].rate : rateHistory[i-1].rate;
        // get the end of the period
        uint256 periodEnd = Math.min(endTime, rateHistory[i].timestamp);
        // calculate the duration of the period (in seconds)
        uint256 periodDuration = periodEnd - currentTime;
        // calculate the rewards for the period
        totalRewards += (balance * currentRate * periodDuration);
        // update the current time to the end of the period
        currentTime = periodEnd;
    }
```

Listing 2.11: src/BaseRewardRateManager.sol

Suggestion Remove the redundant component in the `canExecuteWithdraw()` function's return value.

2.2.5 Revise the misleading annotation for the `rewardRate` variable

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the contract `BaseRewardRateManager`, the annotation (i.e., 2% APY default) of the variable `rewardRate` is misleading. the term `APY` implies a calculation based on compound interest, whereas the variable actually represents an interest rate calculated using simple interest.

Suggestion Revise the annotation accordingly.

2.3 Note

2.3.1 Potential centralization risks

Introduced by `Version 1`

Description There are multiple roles in the protocol that can conduct various privileged operations (e.g., `recoverTokens()`, `setStakingCap()`, `proposeRewardRate()`, and `pause()`), which introduces potential centralization risks. If the private keys of the privileged accounts are lost or maliciously exploited, it could pose a significant risk to the protocol.

2.3.2 Potential DoS due to the increasement of the variable `rateHistory`

Introduced by `Version 1`

Description With the increasement of the number of reward rates, the variable `rateHistory` potentially becomes an array with a large size. Since most operations (e.g., `stake()` and `requestWithdraw()`) will trigger to go through the variable `rateHistory` to calculate total reward, those operations potentially face a denial of service issue due to insufficient gas when the size of the array `rateHistory` is too big.

2.3.3 Potential DoS due to insufficient supply of `rewardsToken`

Introduced by `Version 1`

Description In the `BaseStakingRewards` contract, the project team should ensure a sufficient supply of `rewardsToken` to prevent potential Denial-of-Service (DoS) issues during the reward claiming process.

BOOST WEB3 THROUGH NEXT-GENERATION SECURITY & USABILITY INNOVATIONS