Mantle Fixed Yield Vault Audit Report

Signature: 5a7e6f42a43b873c4656c85c4d0d5b53c363d59374fbbca209d896331d214df4

Mantle Fixed Yield Vault Audit Report

Executive Summary

From Jan 17, 2024, to Jan 19, 2024, the mantle team engaged Fuzzland to conduct a thorough security audit of their project. The primary objective was identifying and mitigating potential security vulnerabilities, risks, and coding issues to enhance the project's robustness and reliability. Fuzzland conducted this assessment over 4 person days, involving 2 engineers who reviewed the code over a span of 2 day. Employing a multifaceted approach that included static analysis, fuzz testing, formal verification, and manual code review, the Fuzzland team identified 3 issue across different severity levels and categories.

Scope

Project Name	Fixed Yield Vault
Repo	🔗 github.com
Commit	`1bf6ace248e0d003a2f804e67142fb5eb177b5bc`
Fixed Commit	`33afef91efd981370c9e0b812d52c0abb627241f`
Language	Solidity
Scope	src/*.sol

Disclaimer

The audit does not ensure that it has identified every security issue in the smart contracts, and it should not be seen as a confirmation that there are no more vulnerabilities. The audit is not exhaustive, and we recommend further independent audits and setting up a public bug bounty program for enhanced security verification of the smart contracts. Additionally, this report should not be interpreted as personal financial advice or recommendations.

Auditing Process

- Static Analysis: We perform static analysis using our internal tools and Slither to identify potential vulnerabilities and coding issues.
- Fuzz Testing: We execute fuzz testing with our internal fuzzers to uncover potential bugs and logic flaws.
- Invariant Development: We convert the project into Foundry project and develop Foundry invariant tests for the project based on the code semantics and documentations.
- Invariant Testing: We run multiple fuzz testing tools, including Foundry and ItyFuzz, to identify violations of invariants we developed.
- Formal Verification: We develop individual tests for critical functions and leverage Halmos to prove the functions in question are not vulnerable.
- Manual Code Review: Our engineers manually review code to identify potential vulnerabilities not captured by previous methods.

Vulnerability Severity

We divide severity into three distinct levels: high, medium, low. This classification helps prioritize the issues identified during the audit based on their potential impact and urgency.

- **High Severity Issues** represent critical vulnerabilities or flaws that pose a significant risk to the system's security, functionality, or performance. These issues can lead to severe consequences such as fund loss, or major service disruptions if not addressed immediately. High severity issues typically require urgent attention and prompt remediation to mitigate potential damage and ensure the system's integrity and reliability.
- **Medium Severity Issues** are significant but not critical vulnerabilities or flaws that can impact the system's security, functionality, or performance. These issues might not pose an immediate threat but have the potential to cause considerable harm if left unaddressed over time. Addressing medium severity issues is important to maintain the overall health and efficiency of the system, though they do not require the same level of urgency as high severity issues.
- **Low Severity Issues** are minor vulnerabilities or flaws that have a limited impact on the system's security, functionality, or performance. These issues generally do not pose a significant risk and can be addressed in the regular maintenance cycle. While low severity issues are not critical, resolving them can help improve the system's overall quality and user experience by preventing the accumulation of minor problems over time.

Below is a summary of the vulnerabilities with their current status, highlighting the number of issues identified in each severity category and their resolution progress.

	Number	Resolved
High Severity Issues	0	0
Medium Severity Issues	0	0
Low Severity Issues	1	1
Info Severity Issues	1	1

Findings

[Low] Admin Role May Conflict

In the `BaseStakingRewards` contract, there is a public visibility `Role` function that has not been overridden. If multiple default administrators exist, they can remove each other's permissions, resulting in an inability to recover access for the affected administrators.

```
  function test_multAdminRole() public {
        bytes32 DEFAULT_ADMIN_ROLE = 0x00;
        address Admin1 = address(1);
        address Admin2 = address(2);
        //set admin 1 2 and revoke
        stakingRewards.grantRole(DEFAULT_ADMIN_ROLE,Admin1);
        stakingRewards.grantRole(DEFAULT_ADMIN_ROLE,Admin2);

        vm.startBroadcast(Admin2);
        stakingRewards.revokeRole(DEFAULT_ADMIN_ROLE,Admin1);
        stakingRewards.grantRole(stakingRewards.PAUSER_ROLE(), address(0x99
9));
        vm.stopBroadcast();

        vm.startBroadcast(Admin1);
        vm.expectRevert();
        stakingRewards.revokeRole(DEFAULT_ADMIN_ROLE,Admin2);
        vm.stopBroadcast();
    }
```

Recommendation:

It is recommended to set only one `DEFAULT_ADMIN_ROLE`, or set permissions to restore the function.

Status: Fixed

[Info] Redundant withdrawal request validation

In the `executeWithdraw()` function, there is a redundant validation of the withdrawal request. The function first calls `canExecuteWithdraw()` to validate the request, but the same validation is also performed in the subsequently called `_executeWithdraw()` function. This duplicate checking results in unnecessary gas consumption.

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

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

Recommendation:

Remove the `canExecuteWithdraw()` check in the `executeWithdraw()` function since the necessary validation is already included in the `_executeWithdraw()` function.

Status: Fixed