33AUDITS & CO.

Inverter Oracle FM Audit Report

Introduction

A time-boxed security review of the protocol was done, focusing on the security aspects of the smart contracts. This audit was performed by 33Audits & Co with Mahdi as the Security Researcher.

Disclaimer

A smart contract security review can never verify the complete absence of vulnerabilities. This is a time, resource, and expertise-bound effort where we try to find as many vulnerabilities as possible. We can not guarantee 100% security after the review or even if the review will find any vulnerabilities. Subsequent security reviews, bug bounty programs, and on-chain monitoring are recommended.

Severity Definitions

Severity	Impact: High	Impact: Medium	Impact: Low
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

Impact - The technical, economic, and reputation damage from a successful attack

Likelihood - The chance that a particular vulnerability gets discovered and exploited

Severity - The overall criticality of the risk

Informational - Findings in this category are recommended changes for improving the structure, usability, and overall effectiveness of the system.

Findings Summary

ID	Title	Severity	Status
[H-01]	Buy Operations Revert Due to Incorrect Fee Handling	High	Fixed
[H-02]	Denial of Service via Unbounded `_queue` Growth in `executePaymentQueue()`	High	Fixed

HIGH

[H-01] Buy Operations Revert Due to Incorrect Fee Handling

Description

The buy operations in the bonding curve implementation are reverting due to an incorrect order of operations in fee handling. The issue occurs in the `_buyOrder` function of `BondingCurveBase_v1.sol`

where the protocol fee transfer is attempted after all funds have already been transferred to the project treasury.

Impact

High - This vulnerability results in:

1. Buy operations failing to execute
2. Protocol fees cannot be collected
3. Users cannot purchase tokens through the bonding curve
4. Core functionality of the bonding curve is compromised

Proof of Concept

The issue stems from two conflicting operations in the buy flow:

```
// Handle collateral tokens before buy
_handleCollateralTokensBeforeBuy(_msgSender(), _depositAmount);

// Process protocol fee on incoming collateral tokens
_processProtocolFeeViaTransfer(
    collateralTreasury, collateralToken, collateralProtocolFeeAmount
);
```

Since all funds are transferred to the project treasury first, there are no funds left in the funding manager to cover the protocol fee transfer, causing the operation to revert.

Recommendation

The order of operations should be reversed to ensure protocol fees are collected before transferring the remaining funds to the project treasury:

1. First calculate and transfer protocol fees
2. Then transfer the remaining amount to the project treasury

This ensures that protocol fees are properly collected before the funds are moved to their final destination.

Resolution

Fixed.

[H-02] Denial of Service via Unbounded `_queue` Growth in `executePaymentQueue()`

Description

The `executePaymentQueue()` function in `PP_Queue_ManualExecution_v1` is vulnerable to a Denial-of-Service (DoS) attack due to unbounded growth of the `_queue` linked list. A malicious actor can repeatedly submit redeem requests with small or dust amounts, resulting in numerous entries being added

to `_queue`. Because `executePaymentQueue()` attempts to iterate over every item in `_queue` and execute the order, the function's gas consumption grows linearly with the number of items in the `_queue`.

Impact

High - This vulnerability results in:

1. Queue execution becomes impossible when too many orders are present
2. Legitimate users cannot have their orders processed
3. System becomes unusable as orders accumulate without being processed
4. Gas costs increase linearly with queue size

Proof of Concept

The following test demonstrates the vulnerability:

```
function test_DoSOrderExecution() public {
    _init();

    // Setup oracle price
    uint issuanceAndRedemptionPrice = 1e6;
    vm.prank(priceSetter);
    permissionedOracle.setIssuanceAndRedemptionPrice(
        issuanceAndRedemptionPrice, issuanceAndRedemptionPrice
    );

    // Prepare buy conditions
    uint buyAmount = 1000e6;
    _prepareBuyConditions(whitelistedUser, buyAmount);

    // Get expected issuance tokens in return
    uint expectedIssuedTokens =
fundingManager.calculatePurchaseReturn(buyAmount);

    // Execute buy
    vm.startPrank(whitelistedUser);
    fundingManager.buy(buyAmount, expectedIssuedTokens);
    vm.stopPrank();

    // Sell Tokens
    uint sellAmount = expectedIssuedTokens / 1805;

    // Execute sell multiple times
    vm.startPrank(whitelistedUser);
    uint256 i;
    while (i < 1805) {
        fundingManager.sell(sellAmount, 1);
        i++;
    }
    vm.stopPrank();

    // Attempt to execute queue
```

```
        vm.startPrank(queueExecutor);
        fundingManager.executeRedemptionQueue();
    }
```

The test shows that the queue execution reverts when the number of orders reaches 1,805 due to out of gas.

Recommendation

Implement the following solutions:

1. Bound the iteration in `executePaymentQueue()`:
    - Process up to a maximum number of orders per call
    - Add a parameter to specify the number of orders to execute
2. Implement rate limiting or filtering:
    - Reject extremely small or dust-value orders
    - Implement minimum thresholds per token or per user

Resolution

Fixed.