CD SECURITY

AUDIT REPORT

AirPuff

March 2024

Introduction

A time-boxed security review of the **AirPuff** protocol was done by **CD Security**, with a focus on the security aspects of the application's implementation.

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 problems with your smart contracts. Subsequent security reviews, bug bounty programs, and on-chain monitoring are strongly recommended.

About **AirPuff**

AirPuff is an ecosystem which enables users to choose different restaking strategies and to engage in leveraged positions in various assets for LRT(Liquid Restaking Token) exposure. It integrates with external protocols to present different options to its users.

Severity classification

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 of a successful attack

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

Severity - the overall criticality of the risk

Security Assessment Summary

***review commit hash* - 3d9458e18caf47a7835f44097ff2385e5eb06a66**

Scope

The following smart contracts were in scope of the audit:

- `contracts/*`

The following number of issues were found, categorized by their severity:

- Critical & High: 0 issues
- Medium: 2 issues
- Low: 3 issues
- Informational: 3 issues

Findings Summary

ID	Title	Severity
[M-01]	Insufficient oracle validation	Medium
[M-02]	Deadline check is not sufficient	Medium
[L-01]	ETH can be stuck in the contracts forever	Low
[L-02]	Lack of two-step role transfer	Low
[L-03]	Use `call()` instead of `transfer()`	Low
[I-01]	Redundant code	Informational
[I-02]	Solidity safe pragma best practices are not used	Informational
[I-03]	Prefer Solidity Custom Errors over `require` statements	Informational

Detailed Findings

[M-01] Insufficient oracle validation

Severity

Impact: High

Likelihood: Low

Description

The `AirPuffHandler::getLatestData` function calls the Chainlink price feed aggregator's `latestRoundData` method, but it does no validation on it - the `answer` is not checked if it is actually a positive number and also the `timestamp` or `answeredInRound` property is not checked if it isn't too old.

```
    function getLatestData(address _token) public view returns (uint256) {
        ...

        (, /* uint80 roundID */ int answer /*uint startedAt*/ /*uint timeStamp*/ /*uint80 answeredInRound*/, , , ) = AggregatorV3Interface(
            chainlinkOracle[_token]
        ).latestRoundData(); //in 1e8
        uint256 decimalPrice;
        if (_token == swapHandlerAddresses.wstETH) {
            decimalPrice = uint256(answer);
        } else {
            decimalPrice = uint256(answer) * 1e10;
        }

        return decimalPrice;
```

This means the contract can operate with old and stale price and lead to significant errors.

Recommendations

Consider validating the data feed:

```
+       require(answeredInRound >= roundID, "Stale price");
+       require(timestamp != 0,"Round not complete");
+       require(answer > 0,"Chainlink answer reporting 0");
```

Client

Fixed

[M-02] Deadline check is not sufficient

Severity

Impact: Medium

Likelihood: Medium

Description

The `swapBalancer` function calls the `batchSwap` method on the Balancer vault to perform swaps of assets. The problem is that the passed deadline parameter is hardcoded to `block.timestamp`.

```
    int256[] memory assetDeltas =
IBalancerVault(swapHandlerAddresses.BalancerVault).batchSwap(
        IBalancerVault.SwapKind.GIVEN_IN,
        swaps,
        assets,
        funds,
        limits,
        block.timestamp // deadline
    );
```

The deadline parameter enforces a time limit by which the transaction must be executed otherwise it will revert. If we take a look at the `batSwap` source code, we can see the following validation:

```
_require(block.timestamp <= deadline, Errors.SWAP_DEADLINE);
```

Now when the deadline is hardcoded as `block.timestamp`, the transaction will not revert because the require statement will always be fulfilled by `block.timestamp == block.timestamp`.

If the provided transaction fee that is too low for miners to be interested in including the transaction in a block, the transaction stays pending in the mempool for extended periods, which could be hours, days, weeks, or even longer.

This could lead to users getting a worse price because a validator can just hold onto the transaction.

Recommendations

Use a user-supplied deadline instead of `block.timestamp`.

Client

Fixed

[L-01] ETH can be stuck in the contracts forever

There are `receive` method implemented which allows the contracts to receive ETH. However, there is no method to withdraw it and the funds will be stuck inside the contracts forever. Consider adding a withdraw method or remove the `receive` method.

Client

Acknowledged

[L-02] Lack of two-step role transfer

All of the contracts in scope have imported the `OwnableUpgradeable.sol` contract forked from OZ which means they lack two-step role transfer. The ownership transfer should be done with great care and two-step role transfer should be preferable.

Use Ownable2StepUpgradeable by OpenZeppelin.

Client

Acknowledged

[L-03] Use `call()` instead of `transfer()` when sending ETH

Couple of functions in the contract are using the `transfer` method to send ETH. These addresses are possible to be a smart contract that have a `receive` or `fallback` function that takes up more than the 2300 gas which is the limit of `transfer`. Examples are some smart contract wallets or multi-sig wallets, so usage of `transfer` is discouraged.

Use .call with value but make sure the `nonReentrant` modifier is present in these functions as well.

Client

Fixed

[I-01] Redundant code

The below events are not used anywhere and can be removed:

```
    event OpenRequest(address indexed user, uint256 amountAfterFee);
        event AddedToPosition(address indexed user, uint256 indexed positionID, uint256 amount,address asset);
```

Client

Fixed

[I-02] Solidity safe pragma best practices are not used

Always use a stable pragma to be certain that you deterministically compile the Solidity code to the same bytecode every time. All of the contracts are currently using a floatable version.

Client

Acknowledged

[I-03] Prefer Solidity Custom Errors over `require` statements with strings

Using Solidity Custom Errors has the benefits of less gas spent in reverted transactions, better interoperability of the protocol as clients of it can catch the errors easily on-chain, as well as you can give descriptive names of the errors without having a bigger bytecode or transaction gas spending, which will result in a better UX as well. Consider replacing the `require` statements with custom errors.

Client

Acknowledged