PRISMA SHIELD

ARBIDEX

DEEP LOGIC AUDIT REPORT

ArbiDex: Arbitrage, ArbiDexRouter, and ArbDexFactory

APR 08 2023

PRISMA SHIELD

Table of Contents

Overview

This audit only covers the Arbitrage, ArbiDexRouter, and ArbDexFactory contracts. It does not cover any other contracts built by ArbiDex.

What is a Deep Logic Audit?

A deep logic smart contract audit is a human-driven code review that checks all of the code business logic for bugs, mathematical errors, and security risks. The audit verifies that the code honors the whitepaper. In addition, this service includes mainnet testing and proactive communication with the project owners to ensure full comprehension of the project to provide the best possible code review quality.

Overview

Findings Summary

	Total Findings	Resolved	Acknowledged
Total Findings	**12**	**12**	**0**
High Security Findings	0	0	0
Medium Security Findings	0	0	0
High Logical Findings	5	5	0
Medium Logical Findings	3	3	0
Informational Findings	4	4	0

ID	Section	Type	Severity	Page	Status
ABG-01	Arbitrage	Logical	High	07	Resolved
ABG-02	Arbitrage	Logical	High	08	Resolved
ABG-03	Arbitrage	Logical	High	09	Resolved
ABG-04	Arbitrage	Logical	High	10	Resolved
ABG-05	Arbitrage	Logical	Medium	11	Resolved
ABG-06	Arbitrage	Logical	Medium	12	Resolved
ABG-07	Arbitrage	Logical	Informational	13	Resolved

Overview

Findings Summary

ID	Section	Type	Severity	Page	Status
ABG-08	Arbitrage	Logical	Informational	14	Resolved
ABG-09	Arbitrage	Logical	Informational	15	Resolved
ABG-10	Arbitrage	Logical	Informational	16	Resolved
DXR-01	ArbiDexRouter	Logical	High	18	Resolved
DXR-02	ArbiDexRouter	Logical	Medium	19	Resolved

Contract Addresses

Arbitrage
https://arbiscan.io/address/0x1e837Ea6F3C1ee918AEFA8db7a2221D4EAAe1c21#code

ArbiDexRouter
https://arbiscan.io/address/0x7238FB45146BD8FcB2c463Dc119A53494be57Aac#code

ArbDexFactory
https://arbiscan.io/address/0x1c6e968f2e6c9dec61db874e28589fd5ce3e1f2c#code

Audit Findings

Arbitrage

ABG-01 - Logical High Severity

generateApproval only approves a limited amount of tokens to the router. If more than the specified amount of tokens are transferred, this will result in the failure of any functions requiring the router to transfer said tokens from the contract. The conductArbitrage function does not currently re-add approval due the checks if (!approvedTokens[tokenA]) and if (!approvedTokens[tokenB]). This is important because the "unlimited" approval mentioned earlier might not actually be truly unlimited in some cases, like for USDC on Arbitrum, which does not have unlimited approval logic.

Recommendation

Grant unlimited approval to the router by instead calling IERC20(_token).approve(router, type(uint256).max); Also change generateApproval to be a public function to allow re-adding approval for USDC if necessary. Finally, calling generateApproval in conductArbitrage is not required. This is because the router only ever transfers USDC from the Arbitrage contract. So approvedTokens can also be removed. This will help save a small amount of gas.

Resolution

The team has implemented the recommendation.

Audit Findings

Arbitrage

ABG-02 - Logical High Severity

tryArbitrage contains a loop without a specified hard limit. If the loop grows too large, the function may face an out-of-gas error, preventing the function from being called and locking out the contract (and the ArbiDexRouter contract which relies on it).

Recommendation

Add an admin function to remove pairs from arbPairs to limit the loop size if necessary.

Resolution

The team has implemented the recommendation.

Audit Findings

Arbitrage

ABG-03 - Logical High Severity

Similar to ABG-02, the recursive nature of computeProfit may result in out-of-gas errors if too many recursive calls happen.

Recommendation

Limit the number of recursive calls that can be done:

```
function computeProfit(uint256 amountIn) internal {
  if (computeProfitCalls == computeProfitCallsLimit) {
    return;
  }
  computeProfitCalls += 1;
  ...
}

function conductArbitrage(address tokenA, address tokenB) internal {
  computeProfitCalls = 0;
  ...
}

function setComputeProfitCallsLimit(uint256 limit) external onlyOwner {
  computeProfitCallsLimit = limit;
}
```

Resolution

The team has implemented the recommendation.

Audit Findings

Arbitrage

ABG-04 - Logical High Severity

In conductArbitrage, the require(profit > 0, "Not profitable"); statement will result in token swaps failing if there is no profit to be made.

Recommendation

Replace the require statement with an if-condition.

Resolution

The team has implemented the recommendation.

Audit Findings

Arbitrage

ABG-05 - Logical Medium Severity

conductArbitrage does not check that the treasury has the 10 USDC, which could result in token swaps failing.

Recommendation

Similar to computeProfit, the following check should be added in conductArbitrage: if (amountIn > IERC20(USDC).balanceOf(treasury)) {return;}

Resolution

The team has implemented the recommendation.

Audit Findings

Arbitrage

ABG-06 - Logical Medium Severity

removePair fails the edge case of the provided _pairAddress not existing in arbPairIndices/arbPairs. It will result in the pair in index 0 of arbPairs being removed even though it might not be the same _pairAddress.

Recommendation

Update the code as such:

```
struct Index {
  uint256 index;
  bool exists;
}

mapping(address => Index) public arbPairIndices;

function addPair(address _pairAddress) external onlyOwner {
  arbPairs.push(Pair(_pairAddress, [IArbDexPair(_pairAddress).token0(),
IArbDexPair(_pairAddress).token1()]));
  arbPairIndices[_pairAddress] = Index(arbPairs.length - 1, true);
}

function removePair(address _pairAddress) external onlyOwner {
  Index memory index = arbPairIndices[_pairAddress];
  require(index.exists);
  ...
}
```

Resolution

The team has implemented the recommendation.

Audit Findings

Arbitrage

ABG-07 - Logical Informational Severity

The setMultiplier function does not check that the provided value is in a reasonable range.

Recommendation

Add minimum and maximum values for the multiplier value.

Resolution

The team has implemented the recommendation.

Audit Findings

Arbitrage

ABG-08 - Logical Informational Severity

minimumTokensOut and requiredTokens should also probably be set in the if-conditions in conductArbitrage in case computeProfit is not able to generate any extra profit.

Recommendation

Set the values of minimumTokensOut and requiredTokens in conductArbitrage.

Resolution

The team has implemented the recommendation.

Audit Findings

Arbitrage

ABG-09 - Logical Informational Severity

To save a small amount of gas, using (block.timestamp + 120) is not necessary in the swapExactTokensForTokens function call.

Recommendation

Simply use block.timestamp in the swapExactTokensForTokens function call.

Resolution

The team has implemented the recommendation.

Audit Findings

Arbitrage

ABG-10 - Logical Informational Severity

In conductArbitrage, one of the two if-conditions should have > changed to >= (amounts1[amounts1.length-1] >= amounts2[amounts2.length-1] in the first if-condition or amounts2[amounts2.length-1] >= amounts1[amounts1.length-1] in the second if-condition). That is to ensure profit is taken if both paths produce the same amount which is greater than the expectedAmount.

Recommendation

Change one of the two if-conditions to be >= instead of >.

Resolution

The team has implemented the recommendation.

Overview

Arbitrage

- This contract is used to conduct arbitrage on specified ArbiDex token pairs when token swaps occur. It tries to swap as much USDC as possible through 3 different token pairs to generate USDC profit, taking into account fees that are generated from the token swaps, which is done throught the tryArbitrage function that is called by the ArbiDexRouter in the token swap functions.

Audit Findings

ArbiDexRouter

DXR-01 - Logical High Severity

swapExactTokensForTokens depends on Arbitrage::tryArbitrage, which in turn depends on swapExactTokensForTokens, forming a circular dependency. This could result in a call to either of those functions to revert if there is an arbitrage opportunity created by the swap.

Recommendation

An if-condition should be added in swapExactTokensForTokens to prevent Arbitrage::tryArbitrage from being called if msg.sender is the Arbitrage contract.

Resolution

The team has implemented the recommendation.

Audit Findings

ArbiDexRouter

DXR-02 - Logcal Medium Severity

The supportingFeeOnTransferTokens functions should also call IArbitrage(arbitrage).tryArbitrage(); to capture arbitrage opportunities. Moreover, _swapSupportingFeeOnTransferTokens should not call IArbitrage(arbitrage).tryArbitrage(); directly if its calling functions are already calling it.

Recommendation

Call IArbitrage(arbitrage).tryArbitrage(); in the supportingFeeOnTransferTokens functions, but not in _swapSupportingFeeOnTransferTokens.

Resolution

The team has implemented the recommendation.

Overview

ArbiDexRouter

- This contract is a fork of the PancakeSwap RouterV2 contract, which is slightly modified to call IArbitrage(arbitrage).tryArbitrage(); to capture arbitrage opportunities.

Overview

ArbDexFactory

- This contract is a fork of the PancakeSwap Factory contract, with the LP mint fees being 20/25 of the growth of the root of K, instead of PancakeSwap's 8/25 of the growth of the root of K.

How to Interpret Findings

Security - High Severity

Indicates that users' funds are at risk or that there is a high probability of exploitation.

Security - Medium Severity

No risk to the protocol or those who interact with it, however it should be highlighted nonetheless.

Logical - High Severity

Indicates that the errors puts users' funds at risk, or can result in significant functional failure in the code.

Logical - Medium Severity

Indicates some functional failure or discrepancy in the code.

Logical - Informational

Minor discrepancy between the intended functionality of the code and the implementation, which does not result in functional failure, or a recommendation to improve the logic.

Yellow Text

Indicates centralization of control and admin powers.

Red Text

An important warning to take note of.

Disclaimer

The information in this deep logic audit report objectively describes the smart contracts being audited, and points out logical and mathematical errors, security risks and vulnerabilities, and optimization opportunities in the audited code. This deep logic audit does not ensure the correctness or authenticity of any software or dApp that interacts with or claims to interact with any smart contract.

This audit report does not constitute any advice whatsoever. You are solely responsible for conducting your own due diligence and consulting your financial advisor before making any investment decisions. Trust in project owners is required to invest in this protocol as a Prisma Shield audit does not ensure the fulfillment of roadmap deliverables and allocation of funds. While our deep logic audits raise the level of security, reliability, mathematical accuracy, and logical soundness of the smart contracts reviewed, they do not amount to any form of warranty or guarantee that the reviewed smart contracts are void of any weaknesses, vulnerabilities, or bugs. Prisma Shield and its founders, employees, owners, and associates are not liable for any damage or loss of funds. Please ensure trust in the team prior to investing as this deep logic audit does not guarantee the promised use of your funds.

PRISMA SHIELD

Introducing Deep Logic Smart Contract Auditing to Web3

prismashield.com

prismashield@gmail.com

PrismaShield

PrismaShield