QuillAudits

Audit Report
August, 2024

For

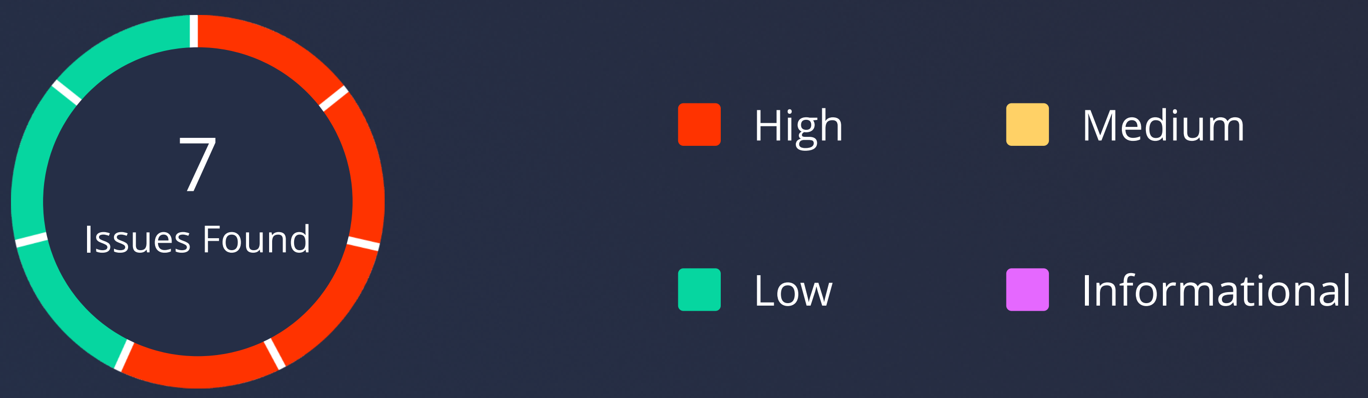
iPaza Labs

Table of Content

Executive Summary

Project Name	ISPZ - Bonding Curve
Overview	The ISPZ project implements a sophisticated token trading system using a bonding curve mechanism. The core BondingCurve contract allows users to buy and sell PAZA tokens using USDC, with prices dynamically adjusted based on the token supply. It incorporates a custom BondingCurveCalculations library for complex mathematical operations, and includes features like tax accumulation and claiming.
Timeline	3rd July 2024 - 26th August 2024
Second Review	19th July 2024 - 22nd August 2024
Method	Manual Review, Functional Testing, Automated Testing, etc. All the raised flags were manually reviewed and re-tested to identify any false positives.
Audit Scope	This audit aimed to analyze the ISPZ - Bonding Curve codebase for quality, security, and correctness. 1. BondingCurve.sol 2. BondingCurveCalculations.sol 3. Math.sol
Source Code	https://github.com/Asharma8810/PazaBondingCurve/tree/develop/contracts
Branch	Develop
Commit Hash	86256e57e134c3e27f18c4265eab19e44c243918
Fixed In	https://github.com/rahul-ray30/PazaBondingCurve/tree/c0615ed369070a24a37c3f4dab1f234bbbeebc41

Number of Security Issues per Severity



	High	Medium	Low	Informational
Open Issues	0	0	0	0
Acknowledged Issues	0	0	1	0
Partially Resolved Issues	0	0	0	0
Resolved Issues	4	0	2	0

Checked Vulnerabilities

- Re-entrancy
- Timestamp Dependence
- Gas Limit and Loops
- DoS with Block Gas Limit
- Transaction-Ordering Dependence
- Use of tx.origin
- Exception disorder
- Gasless send
- Balance equality
- Byte array
- Transfer forwards all gas
- ERC20 API violation
- Compiler version not fixed
- Redundant fallback function
- Send instead of transfer
- Style guide violation
- Unchecked external call
- Unchecked math
- Unsafe type inference
- Implicit visibility level

Techniques and Methods

Throughout the audit of smart contracts, care was taken to ensure:

- The overall quality of code.
- Use of best practices.
- Code documentation and comments match logic and expected behavior.
- Token distribution and calculations are as per the intended behavior mentioned in the whitepaper.
- Implementation of ERC-20 token standards.
- Efficient use of gas.
- Code is safe from re-entrancy and other vulnerabilities.

The following techniques, methods, and tools were used to review all the smart contracts.

Structural Analysis

In this step, we have analyzed the design patterns and structure of smart contracts. A thorough check was done to ensure the smart contract is structured in a way that will not result in future problems.

Static Analysis

A static Analysis of Smart Contracts was done to identify contract vulnerabilities. In this step, a series of automated tools are used to test the security of smart contracts.

Code Review / Manual Analysis

Manual Analysis or review of code was done to identify new vulnerabilities or verify the vulnerabilities found during the static analysis. Contracts were completely manually analyzed, their logic was checked and compared with the one described in the whitepaper. Besides, the results of the automated analysis were manually verified.

Gas Consumption

In this step, we have checked the behavior of smart contracts in production. Checks were done to know how much gas gets consumed and the possibilities of optimization of code to reduce gas consumption.

Tools and Platforms used for Audit

Hardhat, Foundry.

Types of Severity

Every issue in this report has been assigned to a severity level. There are four levels of severity, and each of them has been explained below.

High Severity Issues

A high severity issue or vulnerability means that your smart contract can be exploited. Issues on this level are critical to the smart contract's performance or functionality, and we recommend these issues be fixed before moving to a live environment.

Medium Severity Issues

The issues marked as medium severity usually arise because of errors and deficiencies in the smart contract code. Issues on this level could potentially bring problems, and they should still be fixed.

Low Severity Issues

Low-level severity issues can cause minor impact and are just warnings that can remain unfixed for now. It would be better to fix these issues at some point in the future.

Informational

These are four severity issues that indicate an improvement request, a general question, a cosmetic or documentation error, or a request for information. There is low-to-no impact.

Types of Issues

Open

Security vulnerabilities identified that must be resolved and are currently unresolved.

Resolved

These are the issues identified in the initial audit and have been successfully fixed.

Acknowledged

Vulnerabilities which have been acknowledged but are yet to be resolved.

Partially Resolved

Considerable efforts have been invested to reduce the risk/impact of the security issue, but are not completely resolved.

High Severity Issues

1. Incorrect totalSupply variable update in the buyWithExactUsdc

Path

BondingCurve.sol

Function

```
function buyWithExactUsdc(uint256 usdcAmount) external nonReentrant {
    require(usdcAmount > 0, "Invalid USDC amount");

    // Calculate the amount of tokens to be bought
    uint256 tokenAmount = BondingCurveCalculations.calculateTokenAmountBuy(
        totalSupply,
        usdcAmount
    );
    require(tokenAmount > 0, "Invalid token amount");

    // Update the total supply
    totalSupply += tokenAmount; // @audit incorrect totalSupply update issue here.
```

Description

The function buyWithExactUsdc is responsible for handling token purchases with USDC. However, there are issues with the way totalSupply and usdcPoolBalance are updated. Specifically, the totalSupply is updated incorrectly because the tokenAmount is calculated with 18 decimals, which leads to an incorrect increase in the totalSupply.

Recommendation

- Adjust totalSupply Properly: Ensure that totalSupply is updated correctly by considering the correct decimal places for token amounts.
- Verify usdcPoolBalance Update: Ensure that usdcPoolBalance is updated accurately reflecting the correct USDC amount transferred.

Status

Resolved

2.Improper Accounting in sellWithExactUsdc Function

Path

BondingCurve.sol

Function

```
function sellWithExactUsdc(uint256 usdcAmount) external nonReentrant {
    require(
        usdcAmount > 0 && usdcAmount <= usdcPoolBalance,
        "Invalid USDC amount"
    );

    // Calculate the amount of tokens to be sold for the specified USDC amount
    uint256 tokenAmount = BondingCurveCalculations.calculateTokenAmountSell(
        totalSupply,
        usdcAmount
    ); // @audit issue in this call is that TokenAmountSell
     //returns the numbers in 10 decimals and then below
     // we divide it by 10**18 which will lead to a very small number.

    // Update the total supply (// @audit this result in returning the 0 after division)
    totalSupply -= tokenAmount / 10 ** 18;
```

Description

The sellWithExactUsdc function improperly handles token amount calculations and updates, leading to inaccurate accounting of the totalSupply. The BondingCurveCalculations.calculateTokenAmountSell function returns values with 10 decimals, but the function incorrectly attempts to divide the token amount by 10**18, which results in zero being subtracted from totalSupply. This leads to incorrect total supply updates.

Recommendation

To fix this issue, ensure that the token amount calculations are handled consistently with their decimal format. Specifically, avoid unnecessary conversions that lead to incorrect values.

Status

Resolved

3. Users will not be able to buy pazatokens due to wrong assumptions of decimals.

Path

BondingCurveCalculation.sol

Function

calculateUsdcAmountBuy(), calculateUsdcAmountSell()

Description

There are two functions calculateUsdcAmountBuy and calculateUsdcAmountSell to calculate the amount of usdc based on the input value of tokens provided by these functions in BondingCurve.sol

- buyWithExactToken
- calculateTaxPercent

Both of these functions rely on an assumption that the input token amount will be 18 decimal places.

This can be confirmed by the implementation of calculateUsdcAmountBuy where tokenNumber is calculated by dividing the input token amount with 10**18

```
function calculateUsdcAmountBuy(
    uint256 supply,
    uint256 tokenAmount
) internal pure returns (uint256) {
    // Difference between the supplied tokens and the base value (B)

    uint256 tokenNumber = tokenAmount / 10 ** 18;
    // Difference between the supplied tokens and the base value (B)
    uint256 x = B > supply ? B - supply : supply - B;
```

This means as long as the input token amount has some arbitrary value that does not account for decimal places, tokenNumber will round down to zero resulting in output usdc amount as zero.

This same problem can be seen in calculateUsdcAmountSell also where the calculation of tokenNumber assumes an input amount with 18 decimal places.

```
ftrace | funcSig
function calculateUsdcAmountSell(
    uint256 supply,
    uint256 tokenAmount
) internal pure returns (uint256) {
    uint256 tokenNumber = tokenAmount / (10 ** 18);
    uint256 pb_s = poolBalancePbsSell(supply - tokenNumber); // Pool balance Pbs at given supply
    uint256 pb_s_n = poolBalancePbsSell(supply); // Pool balance Pbs + n at given supply
    uint256 usdcGiven = (pb_s_n - pb_s) / 10 ** 11; // Calculate the USDC amount given the pool balance changes

    // Return the calculated amount of USDC provided
    return usdcGiven;
}
```

2 scenarios can happen here:

1. User tries to call buyWithExactToken with a value (let's say 2000). The function call goes to calculateUsdcAmountSell resulting in tokenNumber = 0 and then usdcGiven = 0. This triggers the require statement

   require(usdcAmount > 0, "Invalid USDC amount");

   and users are not able to buy pazatokens.

2. calculateTaxPercent is called with an amount (let's say 2000), which internally calls calculateUsdcAmountBuy and calculateUsdcAmountSell. There's no require statement here, so the resultant tax = buyPriceUsdc - sellPriceUsdc will be zero

Recommendation

Either change the library implementation to correctly handle decimal places or add a require statement to all the the functions that are using calculateUsdcAmountBuy and calculateUsdcAmountSell like this:

```
require(
    (tokenAmount / 10 ** 18) > 0 &&
        (tokenAmount / 10 ** 18) <= totalSupply,
    "Invalid token amount"
);
```

Status

Resolved

4. Collected Tax Can’t Be Claimed from the Bonding Curve Contract

Path

BondingCurve.sol

Description

The current implementation of the _transfer function sends taxes to the bondingCurve contract. However, the bondingCurve contract lacks a mechanism to withdraw or reclaim these collected taxes, resulting in potential permanent loss of the funds.

Recommendation

Implement a withdrawal mechanism in the bondingCurve contract to allow authorized entities to reclaim the collected taxes. This mechanism should include proper access controls to prevent unauthorized withdrawals.

Status

Resolved

Low Severity Issues

5. Lack of address(0) check in the constructor is not present.

Path

BondingCurve.sol

Description

The constructor logic fails to validate incoming arguments, so the caller can’t accidentally set important state variables to the zero address.

Recommendation

Add a zero address check.

Status

Acknowledged

6. Use OpenZeppelin’s Math.sol instead of custom Math Library

Path

Math.sol

Description

It is recommended to use Math Library provided by OpenZeppelin’s rather than custom implementation to prevent any edgecases associated with incorrect math

OpenZeppelin’s Math library is widely used and is battle tested.

https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/utils/math/Math.sol

Recommendation

Use OpenZeppelin’s Math.sol instead

Status

Resolved

7. Not Following the Checks-Effects-Interactions (CEI) Pattern

Path

BondingCurve.sol

Description

The sellWithExactToken function does not adhere to the Checks-Effects-Interactions (CEI) pattern. The CEI pattern is a best practice in smart contract development, designed to prevent reentrancy attacks by ensuring that all state changes (effects) are made before any external calls (interactions). In this function, the external token transfer is made before the state variables (totalSupply and usdcPoolBalance) are updated. This can potentially lead to vulnerabilities, especially if the external token contract is compromised or behaves unexpectedly.

Recommendation

To follow the CEI pattern, the code should be reorganized so that the state variables are updated (effects) before any external calls (interactions) are made.

Status

Resolved

Functional Tests Cases

Some of the tests performed are mentioned below:

- Should initialize Bonding Curve with correct state variables
- Should be able to sell tokens
- Should be able to buy tokens
- Should update totalSupply correctly when calling buyWithExactUsdc
- Should calculate correct tax percentage when calling calculateTaxPercent

Automated Tests

No major issues were found. Some false positive errors were reported by the tools. All the other issues have been categorized above according to their level of severity.

Closing Summary

In this report, we have considered the security of the ISPZ - Bonding Curve contract. We performed our audit according to the procedure described above.

Some issues of High, Medium, Low severity were found, Some suggestions and best practices are also provided in order to improve the code quality and security posture.In the End, ISPZ Team Resolved almost all Issues.

Disclaimer

QuillAudits Smart contract security audit provides services to help identify and mitigate potential security risks in ISPZ - Bonding Curve smart contracts. However, it is important to understand that no security audit can guarantee complete protection against all possible security threats. QuillAudits audit reports are based on the information provided to us at the time of the audit, and we cannot guarantee the accuracy or completeness of this information. Additionally, the security landscape is constantly evolving, and new security threats may emerge after the audit has been completed.

Therefore, it is recommended that multiple audits and bug bounty programs be conducted to ensure the ongoing security of ISPZ - Bonding Curve smart contracts. One audit is not enough to guarantee complete protection against all possible security threats. It is important to implement proper risk management strategies and stay vigilant in monitoring your smart contracts for potential security risks.

QuillAudits cannot be held liable for any security breaches or losses that may occur subsequent to and despite using our audit services. It is the responsibility of the ISPZ Team to implement the recommendations provided in our audit reports and to take appropriate steps to mitigate potential security risks.

About QuillAudits

QuillAudits is a leading name in Web3 security, offering top-notch solutions to safeguard projects across DeFi, GameFi, NFT gaming, and all blockchain layers. With six years of expertise, we've secured over 1000 projects globally, averting over $30 billion in losses. Our specialists rigorously audit smart contracts and ensure DApp safety on major platforms like Ethereum, BSC, Arbitrum, Algorand, Tron, Polygon, Polkadot, Fantom, NEAR, Solana, and others, guaranteeing your project's security with cutting-edge practices.

1000+
Audits Completed

$30B
Secured

1M+
Lines of Code Audited

Follow Our Journey

Audit Report
August, 2024

For

iPaza Labs

QuillAudits

Canada, India, Singapore, UAE, UK

www.quillaudits.com

audits@quillhash.com