Bullshot

Smart Contract Security Audit

Prepared by ShellBoxes

Nov 21st, 2024 - Dec 3rd, 2024

Shellboxes.com

contact@shellboxes.com

Document Properties

Client	Okratech
Version	1.0
Classification	Public

Scope

Files	MD5 Hash
bullshot-contracts/BCToken.sol	1f066e6f90e4b02bfc253a0af334e5a5
bullshot-contracts/BondingCurve.sol	d86ebc1229bd6c96cd653fe83a52bd2c
bullshot-contracts/BullshotFactory.sol	18cad8b001ba75bc43019b964fb34ebf

Re-Audit

Files	MD5 Hash
contracts/BCToken.sol	e3c6ca2b31a27ea78a0e946ba2ba3923
contracts/BondingCurve.sol	86c8a30448d52e5ff73f485560e13d80
contracts/BullshotFactory.sol	19855ccca0f2c4f135769d34fa20a13b

Contacts

COMPANY	EMAIL
ShellBoxes	contact@shellboxes.com

Contents

1 Introduction

Okratech engaged ShellBoxes to conduct a security assessment on the Bullshot beginning on Nov 21st, 2024 and ending Dec 3rd, 2024. In this report, we detail our methodical approach to evaluate potential security issues associated with the implementation of smart contracts, by exposing possible semantic discrepancies between the smart contract code and design document, and by recommending additional ideas to optimize the existing code. Our findings indicate that the current version of smart contracts can still be enhanced further due to the presence of many security and performance concerns.

This document summarizes the findings of our audit.

1.1 About Okratech

Bullshot is a meme token launcher built on the Base chain, offering a streamlined platform for users to create and deploy meme tokens efficiently. The project is designed to simplify the token creation process and provide tools for users to kickstart their meme token journey. The platform incorporates features aimed at supporting token deployment and liquidity management, with a focus on fostering accessibility and usability for creators within the ecosystem.

Issuer	Okratech
Website	`https://bullshot.org`
Type	Solidity Smart Contract
Audit Method	Whitebox

1.2 Approach & Methodology

ShellBoxes used a combination of manual and automated security testing to achieve a balance between efficiency, timeliness, practicability, and correctness within the audit's scope. While manual testing is advised for identifying problems in logic, procedure, and implementation, automated testing techniques help to expand the coverage of smart contracts and can quickly detect code that does not comply with security best practices.

1.2.1 Risk Methodology

Vulnerabilities or bugs identified by ShellBoxes are ranked using a risk assessment technique that considers both the LIKELIHOOD and IMPACT of a security incident. This framework is effective at conveying the features and consequences of technological vulnerabilities.

Its quantitative paradigm enables repeatable and precise measurement, while also revealing the underlying susceptibility characteristics that were used to calculate the Risk scores. A risk level will be assigned to each vulnerability on a scale of 5 to 1, with 5 indicating the greatest possibility or impact.

- Likelihood quantifies the probability of a certain vulnerability being discovered and exploited in the untamed.
- Impact quantifies the technical and economic costs of a successful attack.
- Severity indicates the risk's overall criticality.

Probability and impact are classified into three categories: H, M, and L, which correspond to high, medium, and low, respectively. Severity is determined by probability and impact and is categorized into four levels, namely Critical, High, Medium, and Low.

Impact			
High	Critical	High	Medium
Medium	High	Medium	Low
Low	Medium	Low	Low
	High	Medium	Low

Likelihood

2 Findings Overview

2.1 Summary

The following is a synopsis of our conclusions from our analysis of the Bullshot implementation. During the first part of our audit, we examine the smart contract source code and run the codebase via a static code analyzer. The objective here is to find known coding problems statically and then manually check (reject or confirm) issues highlighted by the tool. Additionally, we check business logics, system processes, and DeFi-related components manually to identify potential hazards and/or defects.

2.2 Key Findings

In general, these smart contracts are well-designed and constructed, but their implementation might be improved by addressing the discovered flaws, which include , 2 high-severity, 3 medium-severity, 6 low-severity, 1 informational-severity, 1 undetermined-severity vulnerabilities.

Vulnerabilities	Severity	Status
SHB.1. Disabling Slippage Protection in buy invocation in createToken	HIGH	Fixed
SHB.2. The AmountOut Transferred to User Can Be Less Than the Minimum Amount Out	HIGH	Fixed
SHB.3. User Can Bypass buyFee	MEDIUM	Fixed
SHB.4. Approve Race Condition in BCToken Contract	MEDIUM	Fixed
SHB.5. Owner Can Renounce Ownership	MEDIUM	Fixed
SHB.6. Potential Loss of Precision in Fee Calculations	LOW	Partially Fixed
SHB.7. Missing Return Value Verification for addLiquidityETH in buy Function	LOW	Fixed

SHB.8. Potential Reentrancy in buy and sell Functions	LOW	Fixed
SHB.9. Missing Fee Percentage and Amount Verification in setFee Function	LOW	Fixed
SHB.10. Missing Address Verification	LOW	Fixed
SHB.11. Floating Pragma	LOW	Fixed
SHB.12. init Function in BondingCurve Contract Declared as payable	INFORMATIONAL	Acknowledged
SHB.13. Launch Fee Charged Multiple Times for Already Launched Tokens	UNDETERMINED	Fixed

3 Finding Details

SHB.1 Disabling Slippage Protection in buy invocation in createToken

- Severity: HIGH
- Status: Fixed
- Likelihood: 3
- Impact: 2

Description:

In the createToken function, when invoking the buy function, the amountOutMin parameter is set to 0. This effectively disables the slippage protection, meaning that there is no guarantee that the buyer will receive at least the minimum amount of tokens they expect. By setting amountOutMin to 0, the user is not protected against price slippage, which can lead to receiving a significantly lower amount of tokens than expected. This could be exploited by malicious actors, resulting in unfair or unanticipated losses for the user.

Files Affected:

SHB.1.1: BullshotFactory.sol

```
        if (initAmountIn > 0) bondingCurve.buy{ value: msg.value -
            creationFeeAmount }(initAmountIn, 0, msg.sender, block.
            timestamp);
```

Recommendation:

It is recommended to ensure that the amountOutMin parameter is always set to a non-zero value based on the user's expected amount of tokens. This would enable slippage protection and guarantee that the user receives at least the expected amount of tokens.

Updates

The team has resolved the issue by adding a new parameter, amountOutMin, to the createToken function. This parameter dynamically calculates the minimum acceptable tokens received based on the bonding curve logic. The amountOutMin is utilized in the buy invocation to enforce slippage tolerance effectively. This ensures that slippage protection is properly implemented during the buy operation.

SHB.2 The AmountOut Transferred to User Can Be Less Than the Minimum Amount Out

- Severity: HIGH
- Status: Fixed
- Likelihood: 3
- Impact: 2

Description:

In the sell function, the BondingCurve contract performs a validation to ensure that the amount the user will receive (amountOut) is greater than or equal to the specified amountOutMin. However, after this validation, a sell fee is deducted from the amountOut. This means the user could receive an amount less than the validated minimum amount out because the fee is subtracted after the validation. The issue occurs because the amountOut is validated before the fee is applied, which violates the logic that ensures users receive at least the minimum amount they expect.

Files Affected:

SHB.2.1: BondingCurve.sol

```
        virtualTokenReserve += amountIn;
        uint256 newVirtualEthReserve = correlation / virtualTokenReserve;
        amountOut = virtualEthReserve - newVirtualEthReserve;
        virtualEthReserve = newVirtualEthReserve;
```

```
        require(amountOut >= amountOutMin, "! amountOut >= amountOutMin");
        require(amountOut <= ethReserve, "! amountOut >= ethReserve");

        tokenReserve += amountIn;
        ethReserve -= amountOut;

        emit Sell(msg.sender, amountIn, amountOut);

        if (sellFeePercent > 0 && amountOut >= FEE_DENOMINATOR) {
            uint256 fee = amountOut * sellFeePercent / FEE_DENOMINATOR;
            feeRecipient.transfer(fee);
            amountOut -= fee;
        }

        payable(msg.sender).transfer(amountOut);
```

Recommendation:

To address this issue, the contract should apply the fee deduction before the validation of amountOut >= amountOutMin. This will ensure that the final amount after the fee deduction is still guaranteed to meet or exceed the specified amountOutMin.

Updates

The team has addressed the issue by reordering the logic in the sell function. They now deduct fees before performing the require checks for amountOutMin. This ensures that the validated amount takes into account any fees deducted, thus preserving the minimum token amount requirement when transferring tokens to the user.

SHB.3 User Can Bypass buyFee

- Severity: MEDIUM
- Status: Fixed
- Likelihood: 2
- Impact: 2

Description:

The buy function calculates the buyFee only when amountIn is greater than or equal to FEE_DENOMINATOR, using the condition if (buyFeePercent > 0 && amountIn >= FEE_DENOMINATOR). This logic allows users to bypass the buyFee by setting amountIn to a value less than FEE_DENOMINATOR, which skips the fee calculation and transfer. This can result in reduced revenue for the protocol and inconsistent behavior for different transaction sizes.

Files Affected:

SHB.3.1: BondingCurve.sol

```
        uint256 buyFee;
        if (buyFeePercent > 0 && amountIn >= FEE_DENOMINATOR) {
            buyFee = amountIn * buyFeePercent / FEE_DENOMINATOR;
            feeRecipient.transfer(buyFee);
        }
        require(msg.value == amountIn + buyFee, "Wrong value");
```

Recommendation:

Remove the amountIn >= FEE_DENOMINATOR condition from the buyFee calculation. Instead, always apply the fee when buyFeePercent > 0. To prevent abuse through extremely small transactions, consider introducing a minimum amountIn threshold or scaling the fee proportionally for smaller transactions.

Updates

The team has fixed the issue by removing the conditional check amountIn >= FEE_DENOMINATOR for fee application in the buy function. The fee is now always calculated whenever buyFeePercent > 0 for any amountIn. Additionally, a minimum transaction threshold (MIN_AMOUNT) has been introduced to prevent abuse through extremely small transactions, ensuring that the fee is applied consistently and appropriately.

SHB.4 Approve Race Condition in BCToken Contract

- Severity : MEDIUM
- Status : Fixed
- Likelihood : 1
- Impact : 3

Description:

The BCToken contract implements an approve function that allows token holders to grant or modify a spender's allowance. However, this implementation is vulnerable to a known race condition. If a spender transfers tokens using an old allowance while the token holder is updating the allowance, the spender could exploit this to transfer more tokens than intended. This vulnerability stems from overwriting the allowance directly in the approve function without considering ongoing transactions. This issue exists in the allowance mapping and is common to approval mechanisms that do not account for concurrent operations.

Files Affected:

SHB.4.1: BCToken.sol

```
    function approve(address spender, uint256 amount) public returns (
        ↪ bool) {
        allowance[msg.sender][spender] = amount;
        emit Approval(msg.sender, spender, amount);
        return true;
```

```
93        }
```

Recommendation:

To mitigate this issue:

- Adopt an increase/decrease allowance pattern: Replace the approve function with increaseAllowance and decreaseAllowance methods to incrementally adjust allowances rather than overwriting them. This prevents race conditions caused by allowance updates.
- Add safeguards for re-approval: Consider requiring the allowance to be explicitly set to zero before it can be updated to a new value, ensuring a clean reset.

Updates

The team has resolved the issue by adding the increaseAllowance and decreaseAllowance functions. Additionally, the approve function was updated to require the allowance to be explicitly set to zero before it can be updated to a new value. This change effectively prevents potential race conditions that could arise from simultaneous approval operations.

SHB.5 Owner Can Renounce Ownership

- Severity : MEDIUM
- Status : Fixed
- Likelihood : 1
- Impact : 3

Description:

The BullshotFactory contract inherits from OpenZeppelin's Ownable contract allow the owner to renounce ownership. Renouncing ownership leaves the contract without an owner, effectively disabling any functionality exclusively available to the owner.

Files Affected:

SHB.5.1: BullshotFactory.sol

```
import "@openzeppelin/contracts/access/Ownable.sol";
import "@openzeppelin/contracts/proxy/Clones.sol";
import "./BCToken.sol";
import "./BondingCurve.sol";

contract BullshotFactory is Ownable {
```

Recommendation:

It is recommended to override the renounceOwnership function in the BullshotFactory contract and disable its functionality. This ensures ownership is preserved and critical administrative controls remain intact throughout the contract's lifecycle.

Updates

The team has fixed the issue by overriding the renounceOwnership function in the BullshotFactory contract to disable its functionality. This ensures that the contract retains an owner for performing critical administrative operations, preventing the loss of ownership and maintaining control over essential contract management tasks.

SHB.6 Potential Loss of Precision in Fee Calculations

- Severity : LOW
- Status : Partially Fixed
- Likelihood : 1
- Impact : 2

Description:

In the current implementation, the fee calculations for transactions (buy, sell, etc.) use a small FEE_DENOMINATOR (e.g., 1000).

This can lead to a loss of precision, particularly when calculating small fees for low transaction amounts. The result is that fees may be incorrectly rounded down to zero, which could lead to unexpected behavior or loss of fee collection.

Files Affected:

SHB.6.1: BondingCurve.sol

```
            uint256 fee = amountOut * sellFeePercent / FEE_DENOMINATOR;
            feeRecipient.transfer(fee);
```

SHB.6.2: BondingCurve.sol

```
                uint256 fee = ethAmount * launchFeePercent /
                    ↪ FEE_DENOMINATOR;
```

SHB.6.3: BondingCurve.sol

```
            buyFee = amountIn * buyFeePercent / FEE_DENOMINATOR;
```

SHB.6.4: BullshotFactory.sol

```
        require(msg.value == initAmountIn + (initAmountIn * buyFeePercent
            ↪  / FEE_DENOMINATOR) + creationFeeAmount, "Wrong value");
```

Recommendation:

To mitigate the loss of precision in fee calculations, it is recommended to increase the value of FEE_DENOMINATOR to a larger value. This will allow for greater granularity in the fee calculations, ensuring that even small transaction amounts contribute appropriately to the fee pool. Additionally, to avoid scenarios where very small transaction amounts could lead to unintended behavior due to truncation, introduce a minimum threshold check for transaction amounts.

Updates

The team has partially addressed the issue by adding the logic for a minimum fee threshold (MIN_AMOUNT) in the buy function.

SHB.7 Missing Return Value Verification for addLiquidityETH in buy Function

- Severity : LOW
- Status : Fixed
- Likelihood : 1
- Impact : 2

Description:

In the buy function of the BondingCurve contract, the addLiquidityETH function is invoked to add liquidity to Uniswap. However, the return values (amountToken, amountETH, and liquidity) from this call are not checked. Failing to verify these values can lead to undetected errors, such as insufficient liquidity being added or mismatches in the token and ETH amounts. This could result in unexpected behavior, such as incorrect liquidity provisioning or wasted gas, without the contract taking corrective measures.

Files Affected:

SHB.7.1: BondingCurve.sol

```
        uniswapV2Router.addLiquidityETH{ value: ethAmount }(
            address(token),
            tokenAmount,
            tokenAmount,
            ethAmount,
            address(0),
            block.timestamp
        );

        emit Launch(tokenAmount, ethAmount);
```

Recommendation:

Always validate the return values of the addLiquidityETH function to ensure that liquidity is added as expected. Implement checks to verify that:

1. The amountToken and amountETH match the expected values.
2. The liquidity amount is non-zero and within acceptable bounds.

Updates

The team has fixed the issue by capturing and validating the return values (amountToken, amountETH, liquidity) from the addLiquidityETH call in the buy function.

SHB.8 Potential Reentrancy in buy and sell Functions

- Severity : LOW
- Status : Fixed
- Likelihood : 1
- Impact : 2

Description:

The buy and sell functions in the BondingCurve contract lack protections against reentrancy attacks. These functions modify critical state variables such as virtualEthReserve, virtualTokenReserve, ethReserve, and tokenReserve, and also involve external calls such as transferring fees to feeRecipient. Without proper reentrancy protection, a malicious contract could exploit this vulnerability by repeatedly calling these functions before the state changes are finalized. This could lead to double-spending, bypassing fee deductions, or draining reserves.

Files Affected:

SHB.8.1: BondingCurve.sol

```
        feeRecipient.transfer(buyFee);
```

SHB.8.2: BondingCurve.sol

```
        if (sellFeePercent > 0 && amountOut >= FEE_DENOMINATOR) {
            uint256 fee = amountOut * sellFeePercent / FEE_DENOMINATOR;
            feeRecipient.transfer(fee);
            amountOut -= fee;
        }

        payable(msg.sender).transfer(amountOut);
```

Recommendation:

To mitigate the risk of reentrancy:

- Introduce a reentrancy guard by utilizing OpenZeppelin's ReentrancyGuard contract and applying the nonReentrant modifier to the buy and sell functions.
- Ensure that all state variable updates occur before any external calls are made (e.g., feeRecipient.transfer).

Updates

The team has resolved the issue by adding OpenZeppelin's ReentrancyGuard to the BondingCurve contract and applying the nonReentrant modifier to both the buy and sell functions.

SHB.9 Missing Fee Percentage and Amount Verification in setFee Function

- Severity: LOW
- Status: Fixed
- Likelihood: 1
- Impact: 2

Description:

The setFee function in the BullshotFactory contract allows the owner to update several fee-related parameters: creationFeeAmount_, buyFeePercent_, sellFeePercent_, and launchFeePercent_. However, there are no checks to validate that the provided values fall within reasonable and secure limits. For example, fee percentages could be set higher than the FEE_DENOMINATOR, which would break the logic of fee calculations. Similarly, missing checks for non-zero and appropriate ranges can lead to undesirable or malicious configurations that harm users or the contract's financial stability.

Files Affected:

SHB.9.1: BullshotFactory.sol

```
function setFee(uint256 creationFeeAmount_, uint8 buyFeePercent_, uint8 sellFeePercent_, uint8 launchFeePercent_, address payable feeRecipient_) public onlyOwner {
    creationFeeAmount = creationFeeAmount_;
    buyFeePercent = buyFeePercent_;
    sellFeePercent = sellFeePercent_;
    launchFeePercent = launchFeePercent_;
    feeRecipient = feeRecipient_;
}
```

Recommendation:

It is recommended to validate the input values in the setFee function:

- Ensure buyFeePercent_, sellFeePercent_, and launchFeePercent_ are within a specific range (e.g., between 0 and FEE_DENOMINATOR to represent 0% to 100%).
- Validate creationFeeAmount_ to be greater than zero and within an appropriate range.

These validations should be implemented using require statements to ensure only logical and secure values are set for the fees. This will help maintain the contract's functionality.

Updates

The team has addressed the issue by adding require checks in the setFee function. These checks validate that the fee percentages are within the range [0, FEE_DENOMINATOR] and ensure that the creationFeeAmount is greater than zero. This prevents invalid fee values from being set, ensuring proper fee configuration in the contract.

SHB.10 Missing Address Verification

- Severity : LOW
- Status : Fixed
- Likelihood : 1
- Impact : 2

Description:

The BullshotFactory contract's constructor takes several address parameters (uniswapV2Factory, uniswapV2Router, and feeRecipient_), but none of these are verified to ensure that they point to valid, deployed contract addresses or non-zero addresses. This introduces a security risk as the contract can be initialized with invalid or malicious addresses, leading to unexpected behaviors or vulnerabilities. Specifically, the contract could interact with non-existent or malicious contracts, and fees could be sent to a zero address, potentially causing financial losses or failing to properly process fees.

Files Affected:

SHB.10.1: BullshotFactory.sol

```
    constructor(
        address uniswapV2Factory_,
        address uniswapV2Router_,
        uint256 creationFeeAmount_,
        uint8 buyFeePercent_,
        uint8 sellFeePercent_,
        uint8 launchFeePercent_,
        address payable feeRecipient_
    ) {
        setFee(creationFeeAmount_, buyFeePercent_, sellFeePercent_,
            ↪ launchFeePercent_, feeRecipient_);
        uniswapV2Factory = uniswapV2Factory_;
        uniswapV2Router = uniswapV2Router_;
```

Recommendation:

It is recommended to validate all addresses passed to the constructor, specifically ensuring that:

- uniswapV2Factory and uniswapV2Router point to valid contract addresses (using Address.isContract or a similar check).
- feeRecipient_ is a valid, non-zero address.

This can be done with require statements to ensure the addresses are not the zero address and that they are contracts where applicable.

Updates

The team has resolved the issue by adding require checks in the constructor of the BullshotFactory contract. These checks ensure that uniswapV2Factory and uniswapV2Router are valid contract addresses, and that feeRecipient is a non-zero address. This prevents the use of invalid addresses and ensures proper contract initialization.

SHB.11 Floating Pragma

- Severity : LOW
- Status : Fixed
- Likelihood : 1
- Impact : 2

Description:

All the contracts use a floating Solidity pragma of 0.8.19, indicating that they can be compiled with any compiler version from 0.8.19 (inclusive) up to, but not including, version 0.9.0.This flexibility could potentially introduce unexpected behavior if the contracts are compiled with a newer compiler version that includes breaking changes.

Files Affected:

SHB.11.1: BCToken.sol

```
pragma solidity ^0.8.19;
```

SHB.11.2: BondingCurve.sol

```
pragma solidity ^0.8.19;
```

SHB.11.3: BullshotFactory.sol

```
pragma solidity ^0.8.19;
```

Recommendation:

It is generally recommended to lock the pragma statement to a specific Solidity compiler version to ensure consistent behavior across different compiler versions. To achieve this, consider removing the caret (^) from the pragma statement and specifying a fixed version, such as pragma solidity 0.8.19;.

Updates

The team has fixed the issue by updating the pragma statements in all contracts to lock the Solidity version to 0.8.19. This ensures that the contracts will not be affected by any future compiler updates, preventing potential unexpected behavior and maintaining consistent contract functionality.

SHB.12 init Function in BondingCurve Contract Declared as payable

- Severity: INFORMATIONAL
- Status: Acknowledged
- Likelihood : 3
- Impact : 0

Description:

The init function of the BondingCurve contract is declared as payable, which means it is expected to receive Ether when it is called. However, when the createToken function is executed, no Ether is being passed to the init function during initialization. This creates an inconsistency between the contract's function signature and its actual usage.

Files Affected:

SHB.12.1: BondingCurve.sol

```
    function init(
        address factory_,
        address uniswapV2Factory_,
        address uniswapV2Router_,
        BCToken token_,
        uint8 buyFeePercent_,
        uint8 sellFeePercent_,
        uint8 launchFeePercent_,
```

```
        address payable feeRecipient_
    ) public payable returns (address) {
```

SHB.12.2: BullshotFactory.sol

```
        address pair = bondingCurve.init(
            address(this),
            uniswapV2Factory,
            uniswapV2Router,
            token,
            buyFeePercent,
            sellFeePercent,
            launchFeePercent,
            feeRecipient
        );
```

Recommendation:

If the init function is meant to receive Ether during initialization, ensure that the appropriate value is passed to it when calling createToken. Alternatively, if no Ether is required, consider removing the payable modifier from the init function to avoid confusion and prevent unnecessary complexity.

Updates

The team has acknowledged the risk, stating that the payable modifier on the init function is a deliberate design choice. This allows flexibility for developers who may want to provide initial liquidity or bootstrap the bonding curve with an initial buy.

SHB.13 Launch Fee Charged Multiple Times for Already Launched Tokens

- Severity : UNDETERMINED
- Status : Fixed
- Likelihood : 3
- Impact : -

Description:

In the buy function, the launch process is triggered when the ethReserve exceeds the launchThreshold. This includes transferring a launchFeePercent to the feeRecipient and adding liquidity to the Uniswap pool. However, the launch function in the BCToken contract does not validate if the token has already been launched, allowing the fee to be charged multiple times for tokens that are already live. This design flaw can result in unnecessary deductions from user funds under specific scenarios where the ethReserve threshold is met again, despite the token already being in circulation. This may lead to user dissatisfaction and loss of trust in the system.

Files Affected:

SHB.13.1: BondingCurve.sol

```
        if (ethReserve >= launchThreshold) {
            uint256 tokenAmount = token.balanceOf(address(this));

            token.launch();
            ethReserve = 0;

            token.approve(address(uniswapV2Router), tokenAmount);

            if (launchFeePercent > 0) {
                uint256 fee = ethAmount * launchFeePercent /
                    ↪ FEE_DENOMINATOR;
```

```
            feeRecipient.transfer(fee);
            ethAmount -= fee;
        }
```

Recommendation:

1. Implement a condition in the buy function to check the token's launch status (using BCToken.launched) before triggering the launch process and charging the launchFeePercent.
2. Update documentation to explicitly explain the launch process to end users to prevent confusion.

Updates

The team has resolved the issue by adding a check in the buy function to verify the token's launch status using token.launched(). This ensures that the launch process is only triggered once, preventing the launch fee from being charged multiple times for tokens that have already been launched.

4 Best Practices

BP.1 Store Only Token Addresses in tokens Array in BullshotFactory Contract

Description:

Instead of storing the entire BCToken objects in the tokens array, store only the token addresses. This reduces the gas costs, as storing addresses is more efficient than storing entire contract objects. It also simplifies the code and enhances the contract's performance by reducing unnecessary state variables.

Files Affected:

BP.1.1: BullshotFactory.sol

```
    BCToken[] public tokens;
```

Status - Acknowledged

BP.2 Pass deadline from buy Function to addLiquidityETH Call

Description:

Instead of hardcoding block.timestamp in the addLiquidityETH function call, pass the deadline from the buy function's parameter. This ensures consistency and allows the caller to specify the exact expiration time for transactions, providing more control over the contract's execution.

Files Affected:

BP.2.1: BondingCurve.sol

```
function buy(uint256 amountIn, uint256 amountOutMin, address to,
    ↪ uint256 deadline) external payable checkDeadline(deadline)
    ↪ returns (uint256 amountOut) {
```

BP.2.2: BondingCurve.sol

```
        uniswapV2Router.addLiquidityETH{ value: ethAmount }(
            address(token),
            tokenAmount,
            tokenAmount,
            ethAmount,
            address(0),
            block.timestamp
        );
```

Status - Fixed

BP.3 Remove Unused Factory Address Variable in BondingCurve Contract

Description:

The factory address variable in the BondingCurve contract is not being used anywhere, consider removing it to clean up the code. Unused variables increase the complexity of the contract and could potentially lead to confusion or errors in the future.

Files Affected:

BP.3.1: BondingCurve.sol

```
address public factory;
```

Status - Fixed

BP.4 Write Clear Error Messages

Description:

For all require statements, ensure that the error messages are clear, concise, and descriptive. This helps improve the readability and maintainability of the code, making it easier to understand why a specific condition failed. This is particularly important for debugging and contract interaction.

Status - Fixed

BP.5 Remove Hardhat Console Comment

Description:

Remove any commented-out hardhat/console.sol lines before deploying the BCToken contract to production. These are typically used for debugging during development and can unnecessarily increase the size of the contract or introduce unwanted dependencies in production.

Files Affected:

BP.5.1: BCToken.sol

```
//import "hardhat/console.sol";
```

Status - Fixed

BP.6 Public Functions Can Be Declared as External

Description:

Consider declaring functions as external instead of public in Solidity to reduce gas costs. External functions are more restricted, as they cannot be called internally and can only be called by other contracts and externally-owned accounts. This restriction allows the compiler to optimize the function's bytecode, leading to lower gas costs. Review the project contracts to identify public functions that do not need to be called internally and change their visibility to external to benefit from potential gas savings.

Status - Acknowledged

5 Tests

Results:

→ TOKEN - BullshotFactory

✓ buy_full_test

→ TOKEN - Slippage Buy

✓ slippage_buy

→ TOKEN - Slippage Sell

✓ slippage_sell

✓ slippage_sell_all

→ TOKEN - Slippage Sell All

✓ slippage_sell_all

Coverage:

The code coverage results were obtained by running npx hardhat coverage in the Bullshot project. We found the following results :

- Statements Coverage : 75.4%
- Branches Coverage : 47.73%
- Functions Coverage : 70.97%
- Lines Coverage : 77.27%

6 Conclusion

In this audit, we examined the design and implementation of Bullshot contract and discovered several issues of varying severity. Okratech team addressed 11 issues raised in the initial report and implemented the necessary fixes, while classifying the rest as a risk with low-probability of occurrence. Shellboxes' auditors advised Okratech Team to maintain a high level of vigilance and to keep those findings in mind in order to avoid any future complications.

7 Disclaimer

Shellboxes reports should not be construed as "endorsements" or "disapprovals" of particular teams or projects. These reports do not reflect the economics or value of any "product" or "asset" produced by any team or project that engages Shellboxes to do a security evaluation, nor should they be regarded as such. Shellboxes Reports do not provide any warranty or guarantee regarding the absolute bug-free nature of the examined technology, nor do they provide any indication of the technology's proprietors, business model, business or legal compliance. Shellboxes Reports should not be used in any way to decide whether to invest in or take part in a certain project. These reports don't offer any kind of investing advice and shouldn't be used that way. Shellboxes Reports are the result of a thorough auditing process designed to assist our clients in improving the quality of their code while lowering the significant risk posed by blockchain technology. According to Shellboxes, each business and person is in charge of their own due diligence and ongoing security. Shellboxes does not guarantee the security or functionality of the technology we agree to research; instead, our purpose is to assist in limiting the attack vectors and the high degree of variation associated with using new and evolving technologies.

SHELLBOXES

For a Contract Audit, contact us at contact@shellboxes.com