MoarCandy Security Review

Pashov Audit Group

Conducted by: T1MOH, SpicyMeatball, pontifex

June 27th 2024 - June 29th 2024

Contents

[L-05] Lack of checking _availableTokenBalance parameter

1. About Pashov Audit Group

Pashov Audit Group consists of multiple teams of some of the best smart contract security researchers in the space. Having a combined reported security vulnerabilities count of over 1000, the group strives to create the absolute very best audit journey possible - although 100% security can never be guaranteed, we do guarantee the best efforts of our experienced researchers for your blockchain protocol. Check our previous work here or reach out on Twitter @pashovkrum.

2. 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.

3. Introduction

A time-boxed security review of the **MoarCandy/contracts** repository was done by **Pashov Audit Group**, with a focus on the security aspects of the application's smart contracts implementation.

4. About MoarCandy

MoarCandy allows creainge new ERC20 tokens with customizable bonding curves and fees. The `ContinuosBondingERC20Token` contract is an ERC20 token with a bonding curve that handles buying and selling of tokens using ETH, implements fees, and can create liquidity pairs on decentralized exchanges like Uniswap or TraderJoe.

5. Risk Classification

Severity	Impact: High	Impact: Medium	Impact: Low
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

5.1. Impact

- High - leads to a significant material loss of assets in the protocol or significantly harms a group of users.
- Medium - only a small amount of funds can be lost (such as leakage of value) or a core functionality of the protocol is affected.
- Low - can lead to any kind of unexpected behavior with some of the protocol's functionalities that's not so critical.

5.2. Likelihood

- High - attack path is possible with reasonable assumptions that mimic on-chain conditions, and the cost of the attack is relatively low compared to the amount of funds that can be stolen or lost.
- Medium - only a conditionally incentivized attack vector, but still relatively likely.
- Low - has too many or too unlikely assumptions or requires a significant stake by the attacker with little or no incentive.

5.3. Action required for severity levels

- Critical - Must fix as soon as possible (if already deployed)
- High - Must fix (before deployment if not already deployed)
- Medium - Should fix
- Low - Could fix

6. Security Assessment Summary

review commit hash - b8428cf068b78954c44ac5d1fe9c04b411efa541

fixes review commit hash - c72d907e6ff828cd807913a5a4751392a0765f8c

Scope

The following smart contracts were in scope of the audit:

- `BondingERC20TokenFactory`
- `ContinuosBondingERC20Token`
- `AMMFormula`

7. Executive Summary

Over the course of the security review, T1MOH, SpicyMeatball, pontifex engaged with MoarCandy to review MoarCandy. In this period of time a total of **12** issues were uncovered.

Protocol Summary

Protocol Name	MoarCandy
Repository	https://github.com/MoarCandy/contracts
Date	June 27th 2024 - June 29th 2024
Protocol Type	Bonding curve tokensale

Findings Count

Severity	**Amount**
Critical	3
High	2
Medium	2
Low	5
Total Findings	**12**

Summary of Findings

ID	Title	Severity	Status
[C-01]	Creating a pair while the token is in the initial sale phase	Critical	Resolved
[C-02]	Creating a token with a poisoned router	Critical	Resolved
[C-03]	Blocking the initial liquidity seed with a 1 wei donation	Critical	Resolved
[H-01]	The bonding curve implementation can not be fully changed	High	Resolved
[H-02]	Bonding token transfers DOS after the pair was created	High	Resolved
[M-01]	Lack of isLpCreated validation	Medium	Resolved
[M-02]	Incorrect fee calculation	Medium	Resolved
[L-01]	No sanity check on creating pair for rounding issues	Low	Resolved
[L-02]	No sanity checks on setters	Low	Acknowledged
[L-03]	Incorrect check in the sellTokens	Low	Resolved
[L-04]	Incorrect slippage check placement in buyTokens	Low	Resolved
[L-05]	Lack of checking _availableTokenBalance parameter	Low	Acknowledged

8. Findings

8.1. Critical Findings

[C-01] Creating a pair while the token is in the initial sale phase

Severity

Impact: High

Likelihood: High

Description

As the initial sale approaches its end, a scenario can occur where a user buys more tokens than are available for sale. In this situation, the user will receive the maximum available token amount and any excess ETH will be refunded to them:

```
function buyTokens
      (uint256 minExpectedAmount) external payable nonReentrant returns (uint256) {
        if (liquidityGoalReached()) revert LiquidityGoalReached();
        if (msg.value == 0) revert NeedToSendETH();

        uint256 ethAmount = msg.value;
        uint256 feeAmount = (ethAmount * buyFee) / PERCENTAGE_DENOMINATOR;
        uint256 remainingAmount = ethAmount - feeAmount;

        uint256 tokenReserveBalance = getReserve();
>>      uint256 maxTokenToReceive = tokenReserveBalance -
  (MAX_TOTAL_SUPPLY - availableTokenBalance);

        uint256 tokensToReceive =
            bondingCurve.calculatePurchaseReturn
              (remainingAmount, ethBalance, tokenReserveBalance, bytes(""));
        if
          (tokensToReceive < minExpectedAmount) revert InSufficientAmountReceived();
        uint256 ethReceivedAmount = remainingAmount;
        uint256 refund;
>>      if (tokensToReceive > maxTokenToReceive) {
            tokensToReceive = maxTokenToReceive;
            ethReceivedAmount = getOutputPrice
              (tokensToReceive, ethBalance, tokenReserveBalance);
            feeAmount = (ethReceivedAmount * buyFee) / PERCENTAGE_DENOMINATOR;
            if (msg.value < (feeAmount + ethReceivedAmount)) {
                revert InsufficientETH();
            }
            refund = msg.value - (feeAmount + ethReceivedAmount);
        }
        ethBalance += ethReceivedAmount;
        treasuryClaimableEth += feeAmount;

        _transfer(address(this), msg.sender, tokensToReceive);

>>      (bool sent,) = msg.sender.call{ value: refund }("");
        if (!sent) {
            revert FailedToSendETH();
        }
        ---SNIP---
```

This ETH transfer can be exploited. A malicious contract can create a Token/WETH pair within the receive callback. Since the `liquidityGoalReached()` condition is met, the transfer to the router will not revert. After creating the pair, the attacker can send some amount of bonding tokens to the ERC20 contract to disable `liquidityGoalReached()`, preventing the pair from being created a second time and ensuring the attack does not revert. A pair can then be created with incorrect proportions.

As a result, when the sale finally reaches its cap, the `_createPair` transaction will fail, causing the bonding token to remain stuck in the sale phase and trapping the collected ETH and minted tokens.

Coded POC for `ContinuosBondingERC20TokenTest.t.sol`:

Malicious contract:

```
contract Refundee {
    address internal router = 0x60aE616a2155Ee3d9A68541Ba4544862310933d4; //
    // trader joe router
    receive() external payable {
        console2.log("HUH");
        ContinuosBondingERC20Token(msg.sender).approve(router, 1 ether);
        address wNative = IJoeRouter02(router).WAVAX();
        console2.log("SUCC: ", address(this).balance);
        IJoeRouter02(router).addLiquidityAVAX{ value: 1 ether }(
            msg.sender, 1 ether, 1, 1, address(this), block.timestamp
        );
        ContinuosBondingERC20Token(msg.sender).transfer
          (msg.sender, 200_000_001 ether);
    }
}
```

Exploit:

```
function testPooled() public {
        address attacker = address(new Refundee());
        vm.deal(attacker, 10_000 ether);
        vm.deal(user, 10_000 ether);
        vm.startPrank(attacker);

        bondingERC20Token.buyTokens{ value: 202.2 ether }(0);
        // This call will revert and the token will be stuck in the initial sale
        // phase
        vm.startPrank(user);
        vm.expectRevert();
        bondingERC20Token.buyTokens{ value: 200 ether }(0);
    }
```

Recommendations

```
-        (bool sent,) = msg.sender.call{ value: refund }("");
-        if (!sent) {
-            revert FailedToSendETH();
-        }

         if (liquidityGoalReached()) {
             _createPair();
         }

+        (bool sent,) = msg.sender.call{ value: refund }("");
+        if (!sent) {
+            revert FailedToSendETH();
+        }
```

[C-02] Creating a token with a poisoned router

Severity

Impact: High

Likelihood: High

Description

The `BondingERC20TokenFactory.deployBondingERC20Token` function does not validate the user's input. So it is possible to create a bonding token with a fake router to steal all ETH from the token sales.

```
function deployBondingERC20Token(
>>      address _router,
        string memory _name,
        string memory _symbol,
        LP_POOL _poolType
    )
        public
        returns (address)
    {

>>          _router,
            _name,
            _symbol,
            treasury,
            buyFee,
            sellFee,
            bondingCurve,
            initialTokenBalance,
            availableTokenBalance,
            _poolType
        );
        emit TokenDeployed(address(_bondingERC20Token), msg.sender);

        return address(_bondingERC20Token);
    }
```

ContinuosBondingERC20Token.sol

```
constructor(
>>      address _router,
        string memory _name,
        string memory _symbol,
        address _treasury,
        uint256 _buyFee,
        uint256 _sellFee,
        IBondingCurve _bondingCurve,
        uint256 _initialTokenBalance,
        uint256 _availableTokenBalance,
        LP_POOL _poolType
    )
        ERC20(_name, _symbol)
    {
>>      router = _router;
```

Recommendations

Consider checking the router address with the values from the DEXes official documentation: https://support.traderjoexyz.com/en/articles/6807983-contracts-api https://docs.uniswap.org/contracts/v2/reference/smart-contracts/v2-deployments

[C-03] Blocking the initial liquidity seed with a 1 wei donation

Severity

Impact: High

Likelihood: High

Description

Consider the following scenario:

1. While the bonding token (TOK) is still in the initial sale phase, the attacker creates an empty TOK/WETH pair on the Uniswap
2. He then donates 1 wei of WETH to the pair and calls the `sync` function:

```
///
// https://github.com/Uniswap/v2-core/blob/master/contracts/UniswapV2Pair.sol#L198
    function sync() external lock {
        _update(IERC20(token0).balanceOf(address(this)), IERC20
          (token1).balanceOf(address(this)), reserve0, reserve1);
    }
```

As a result, one of the reserves will be non-zero. 3. When the sale is over, the protocol will attempt to create a pair via the Uniswap router, but the call will fail:

```
function _addLiquidity(
        address tokenA,
        address tokenB,
        uint amountADesired,
        uint amountBDesired,
        uint amountAMin,
        uint amountBMin
    ) internal virtual returns (uint amountA, uint amountB) {
        // create the pair if it doesn't exist yet
        if (IUniswapV2Factory(factory).getPair(tokenA, tokenB) == address(0)) {
            IUniswapV2Factory(factory).createPair(tokenA, tokenB);
        }
>>      (uint reserveA, uint reserveB) = UniswapV2Library.getReserves
  (factory, tokenA, tokenB);
        if (reserveA == 0 && reserveB == 0) {
            (amountA, amountB) = (amountADesired, amountBDesired);
        } else {
>>          uint amountBOptimal = UniswapV2Library.quote
  (amountADesired, reserveA, reserveB);
            if (amountBOptimal <= amountBDesired) {
                require(
                  amountBOptimal>=amountBMin,
                  'UniswapV2Router:INSUFFICIENT_B_AMOUNT'
                );
                (amountA, amountB) = (amountADesired, amountBOptimal);
            } else {
>>              uint amountAOptimal = UniswapV2Library.quote
  (amountBDesired, reserveB, reserveA);
                assert(amountAOptimal <= amountADesired);
                require(
                  amountAOptimal>=amountAMin,
                  'UniswapV2Router:INSUFFICIENT_A_AMOUNT'
                );
                (amountA, amountB) = (amountAOptimal, amountBDesired);
            }
        }
    }
```

Because one of the reserves has a non-zero value the `quote` function will be invoked:

```
function quote(
      uintamountA,
      uintreserveA,
      uintreserveB
    ) internal pure returns (uint amountB
        require(amountA > 0, 'UniswapV2Library: INSUFFICIENT_AMOUNT');
>>      require
  (reserveA > 0 && reserveB > 0, 'UniswapV2Library: INSUFFICIENT_LIQUIDITY');
        amountB = amountA.mul(reserveB) / reserveA;
    }
```

And since one of the reserves is empty, `addLiquidity` will fail to trap bonding tokens and collected ETH in the contract.

Recommendations

To address this issue, it is suggested to interact directly with the Uniswap pair instead of relying on a router:

```
if (poolType == LP_POOL.Uniswap) {
            wNative = IUniswapV2Router02(router).WETH();
+           IFactory factory = IFactory(IUniswapV2Router02(router).factory());
+           address pair = factory.getPair(address(this), wNative);
+           if(pair == address(0)) pair = factory.createPair(address
+ (this), wNative));
+           IWETH(wNative).deposit{value: currentEth}();
+           IWETH(wNative).transfer(pair, currentEth);
+           _transfer(address(this), pair, currentTokenBalance);
+           uint256 liquidity = IPair(pair).mint(address(this));
        } else if (poolType == LP_POOL.TraderJoe) {
            wNative = IJoeRouter02(router).WAVAX();
+           IFactory factory = IFactory(IJoeRouter02(router).factory());
+           address pair = factory.getPair(address(this), wNative);
+           if(pair == address(0)) pair = factory.createPair(address
+ (this), wNative));
+           IWETH(wNative).deposit{value: currentEth}();
+           IWETH(wNative).transfer(pair, currentEth);
+           _transfer(address(this), pair, currentTokenBalance);
+           uint256 liquidity = IPair(pair).mint(address(this));
```

8.2. High Findings

[H-01] The bonding curve implementation can not be fully changed

Severity

Impact: High

Likelihood: Medium

Description

The `BondingERC20TokenFactory.updateBondingCurve` function should update the bonding curve implementation, but it does not change the `getOutputPrice` function, which also calculates the token price since the function is hardcoded in the `ContinuosBondingERC20Token` contract.

```
function getOutputPrice(
        uint256 outputAmount,
        uint256 inputReserve,
        uint256 outputReserve
    )
        public
        pure
        returns (uint256)
    {
        require(
          inputReserve>0&&outputReserve>0,
          "Reservesmustbegreaterthan0"
        );
        uint256 numerator = inputReserve * outputAmount;
        uint256 denominator = (outputReserve - outputAmount);
        return numerator / denominator + 1;
    }
```

This can cause an incorrect token price calculation at the end of the curve where the price usually is the highest.

Recommendations

Consider implementing the `getOutputPrice` function in the `AMMFormula` contract instead of the `ContinuosBondingERC20Token` contract.

[H-02] Bonding token transfers DOS after the pair was created

Severity

Impact: High

Likelihood: Medium

Description

Users are not permitted to transfer bonding tokens before the initial sale is concluded and the Token/WETH pair is created:

```
function _update
      (address from, address to, uint256 value) internal virtual override {
        // will revert for normal transfer till goal not reached
>>      if (
            !liquidityGoalReached() && from != address(0) && to != address
              (0) && from != address(this)
                && to != address(this)
        ) {
            revert TransferNotAllowedUntilLiquidityGoalReached();
        }
        super._update(from, to, value);
    }
```

Unfortunately, this protection can be exploited to perform DoS on token transfers after the pair with initial liquidity has been created. An attacker can donate `MAX_TOTAL_SUPPLY - availableTokenBalance + 1` tokens and reset the `liquidityGoalReached()`:

```
function liquidityGoalReached() public view returns (bool) {
        return getReserve() <= (MAX_TOTAL_SUPPLY - availableTokenBalance);
    }
```

Coded POC for the `ContinuosBondingERC20TokenTest.t.sol`:

```
function testTokenDown() public {
        vm.deal(user, 10_000 ether);
        vm.startPrank(user);

        bondingERC20Token.buyTokens{ value: 202.2 ether }(0);

        vm.expectRevert();
        bondingERC20Token.buyTokens{ value: 1 wei }(0);

        // pair is created
        assertEq(bondingERC20Token.isLpCreated(), true);
        assertEq(bondingERC20Token.getReserve(), 0);

        // donate tokens to the ERC20 contract
        bondingERC20Token.transfer(address
          (bondingERC20Token), 200_000_001 ether);

        // revert TransferNotAllowedUntilLiquidityGoalReached()
        vm.expectRevert();
        bondingERC20Token.transfer(address(this), 10 ether);
    }
```

Recommendations

```
function _update
      (address from, address to, uint256 value) internal virtual override {
        // will revert for normal transfer till goal not reached
       if (
            !liquidityGoalReached() && from != address(0) && to != address
              (0) && from != address(this)
+               && to != address(this) && !isPairCreated
        ) {
```

8.3. Medium Findings

[M-01] Lack of `isLpCreated` validation

Severity

Impact: Medium

Likelihood: Medium

Description

The lack of `isLpCreated` variable validation in the `ContinuosBondingERC20Token` contract functions makes it possible to return the bonding token to the purchase phase. This can cause unexpected behavior such as secondary involving `_createPair` or withdrawing treasury fees but it is almost always economically inefficient. The `isLpCreated` variable is set to `true` in the `_createPair` function but is never checked.

```
function _createPair() internal {
        uint256 currentTokenBalance = getReserve();
        uint256 currentEth = ethBalance - initialTokenBalance;
>>      isLpCreated = true;
```

Recommendations

Consider checking `isLpCreated` variable in `buyTokens` and `sellTokens`

```
if (liquidityGoalReached() || isLpCreated) revert LiquidityGoalReached
          ();
```

[M-02] Incorrect fee calculation

Severity

Impact: Medium

Likelihood: Medium

Description

The `ContinuosBondingERC20Token.buyTokens` charges less fees for the case when `tokensToReceive > maxTokenToReceive` than for the common flow.

```
function buyTokens
      (uint256 minExpectedAmount) external payable nonReentrant returns (uint256) {
        if (liquidityGoalReached()) revert LiquidityGoalReached();
        if (msg.value == 0) revert NeedToSendETH();

>>      uint256 ethAmount = msg.value;
>>      uint256 feeAmount = (ethAmount * buyFee) / PERCENTAGE_DENOMINATOR;
        uint256 remainingAmount = ethAmount - feeAmount;

        uint256 tokenReserveBalance = getReserve();
        uint256 maxTokenToReceive = tokenReserveBalance -
          (MAX_TOTAL_SUPPLY - availableTokenBalance);

        uint256 tokensToReceive =
            bondingCurve.calculatePurchaseReturn
              (remainingAmount, ethBalance, tokenReserveBalance, bytes(""));
        if
          (tokensToReceive < minExpectedAmount) revert InSufficientAmountReceived();
        uint256 ethReceivedAmount = remainingAmount;
        uint256 refund;
        if (tokensToReceive > maxTokenToReceive) {
            tokensToReceive = maxTokenToReceive;
>>          ethReceivedAmount = getOutputPrice
  (tokensToReceive, ethBalance, tokenReserveBalance);
>>          feeAmount = (ethReceivedAmount * buyFee) / PERCENTAGE_DENOMINATOR;
            if (msg.value < (feeAmount + ethReceivedAmount)) {
                revert InsufficientETH();
            }
>>          refund = msg.value - (feeAmount + ethReceivedAmount);
        }
```

Recommendations

Consider using a different way for fee calculation when `tokensToReceive > maxTokenToReceive`:

```
feeAmount = ethReceivedAmount * PERCENTAGE_DENOMINATOR /
  (PERCENTAGE_DENOMINATOR -  buyFee) - ethReceivedAmount;
```

8.4. Low Findings

[L-01] No sanity check on creating pair for rounding issues

When the liquidity goal is reached, the token creates UniV2 pair and supplies Token + ETH. As you can see, the contract forwards `currentEth`, i.e. expects that it has sufficient balance:

```
function _createPair() internal {
        ...
@>      uint256 currentEth = ethBalance - initialTokenBalance;
        ...
        if (poolType == LP_POOL.Uniswap) {
            wNative = IUniswapV2Router02(router).WETH();
@>          (,, liquidity) = IUniswapV2Router02
  (router).addLiquidityETH{ value: currentEth }(
                address(
                  this
                ), currentTokenBalance, currentTokenBalance, currentEth, address(this
            );
        } else if (poolType == LP_POOL.TraderJoe) {
            wNative = IJoeRouter02(router).WAVAX();
@>          (,, liquidity) = IJoeRouter02
  (router).addLiquidityAVAX{ value: currentEth }(
                address(
                  this
                ), currentTokenBalance, currentTokenBalance, currentEth, address(this
            );
        }
        ...
    }
```

Consider adding a sanity check to ensure it will never revert because of insufficient balance. It may occur due to some other potential issue.

```
uint256 currentEth = ethBalance - initialTokenBalance;
+       currentEth = currentEth > address(this).balance ? address
+ (this).balance : currentEth;
```

[L-02] No sanity checks on setters

Currently, there are no safe bounds for new values, consider adding them to catch early admin mistakes:

```
function updateBuyFee(uint256 _newBuyFee) public onlyOwner {
        emit BuyFeeUpdated(_newBuyFee, buyFee);
        buyFee = _newBuyFee;
    }

    function updateSellFee(uint256 _newSellFee) public onlyOwner {
        emit SellFeeUpdated(_newSellFee, sellFee);
        sellFee = _newSellFee;
    }

    function updateTreasury(address _newTreasury) public onlyOwner {
        emit TreasuryUpdated(_newTreasury, treasury);
        treasury = _newTreasury;
    }

    function updateAvailableTokenBalance
      (uint256 _newAvailableTokenBalance) public onlyOwner {
        emit AvailableTokenUpdated
          (_newAvailableTokenBalance, availableTokenBalance);
        availableTokenBalance = _newAvailableTokenBalance;
    }

    function updateInitialTokenBalance
      (uint256 _newInitialTokenBalance) public onlyOwner {
        emit InitialTokenBalanceUpdated
          (_newInitialTokenBalance, initialTokenBalance);
        initialTokenBalance = _newInitialTokenBalance;
    }

    function updateBondingCurve
      (IBondingCurve _newBondingCurve) public onlyOwner {
        emit BondingCurveUpdated(address(_newBondingCurve), address
          (bondingCurve));
        bondingCurve = _newBondingCurve;
    }
```

Describe the finding and your recommendation here

[L-03] Incorrect check in the `sellTokens`

The ETH balance sufficiency check in the `ContinuosBondingERC20Token.sellTokens` does not count the `treasuryClaimableEth` variable but should.

```
function sellTokens(
      uint256tokenAmount,
      uint256minExpectedEth
    ) external nonReentrant returns (uint256
        if (liquidityGoalReached()) revert LiquidityGoalReached();
        if (tokenAmount == 0) revert NeedToSellTokens();

        uint256 tokenReserveBalance = getReserve();
        uint256 reimburseAmount =
            bondingCurve.calculateSaleReturn
              (tokenAmount, tokenReserveBalance, ethBalance, bytes(""));

        uint256 feeAmount =
          (reimburseAmount * sellFee) / PERCENTAGE_DENOMINATOR;
        ethBalance -= reimburseAmount;
        reimburseAmount -= feeAmount;
>>      treasuryClaimableEth += feeAmount;

        if (reimburseAmount < minExpectedEth) revert InSufficientAmountReceived
          ();
>>      if (address
  (this).balance < reimburseAmount) revert ContractNotEnoughETH();

        _transfer(msg.sender, address(this), tokenAmount);
        (bool sent,) = msg.sender.call{ value: reimburseAmount }("");
        if (!sent) revert FailedToSendETH();

        if (treasuryClaimableEth >= 0.1 ether) {
            (bool sent,) = TREASURY_ADDRESS.call{ value: treasuryClaimableEth }
              ("");
            treasuryClaimableEth = 0;
            if (!sent) {
                revert FailedToSendETH();
            }
        }

        emit TokensSold(msg.sender, tokenAmount, reimburseAmount, feeAmount);
    }
```

[L-04] Incorrect slippage check placement in `buyTokens`

The bonding token sale includes a slippage check that allows users to specify the minimum amount of tokens they expect to receive in a buy transaction. If this condition is not met, the transaction will revert:

```
function buyTokens
     (uint256 minExpectedAmount) external payable nonReentrant returns (uint256) {
        if (liquidityGoalReached()) revert LiquidityGoalReached();
        if (msg.value == 0) revert NeedToSendETH();

        uint256 ethAmount = msg.value;
        uint256 feeAmount = (ethAmount * buyFee) / PERCENTAGE_DENOMINATOR;
        uint256 remainingAmount = ethAmount - feeAmount;

        uint256 tokenReserveBalance = getReserve();
        uint256 maxTokenToReceive = tokenReserveBalance -
          (MAX_TOTAL_SUPPLY - availableTokenBalance);

        uint256 tokensToReceive =
            bondingCurve.calculatePurchaseReturn
              (remainingAmount, ethBalance, tokenReserveBalance, bytes(""));
>>      if
  (tokensToReceive < minExpectedAmount) revert InSufficientAmountReceived();
```

However, this check is ineffective in a particular situation. If the purchased amount exceeds the available amount for the sale, the received amount will be capped. As a result, the user may receive fewer tokens than specified in `minExpectedAmount`, causing the transaction to proceed despite the slippage check.

Consider the following fix:

```
function buyTokens
      (uint256 minExpectedAmount) external payable nonReentrant returns (uint256) {
        ---SNIP---
        if (tokensToReceive > maxTokenToReceive) {
            tokensToReceive = maxTokenToReceive;
            ethReceivedAmount = getOutputPrice
              (tokensToReceive, ethBalance, tokenReserveBalance);
            feeAmount = (ethReceivedAmount * buyFee) / PERCENTAGE_DENOMINATOR;
            if (msg.value < (feeAmount + ethReceivedAmount)) {
                revert InsufficientETH();
            }
            refund = msg.value - (feeAmount + ethReceivedAmount);
        }
        ethBalance += ethReceivedAmount;
        treasuryClaimableEth += feeAmount;
+       if
+ (tokensToReceive < minExpectedAmount) revert InSufficientAmountReceived();
        _transfer(address(this), msg.sender, tokensToReceive);
```

[L-05] Lack of checking `_availableTokenBalance` parameter

The `availableTokenBalance` variable has a significant meaning for the token starting price on DEXes which should correspond with the price of the token at the end of the bonding curve to prevent arbitrage.

```
constructor(
        address _owner,
        IBondingCurve _bondingCurve,
        address _treasury,
        uint256 _initialTokenBalance,
>>      uint256 _availableTokenBalance,
        uint256 _buyFee,
        uint256 _sellFee
    )
        Ownable(_owner)
    {
        bondingCurve = _bondingCurve;
        treasury = _treasury;
        initialTokenBalance = _initialTokenBalance;
>>      availableTokenBalance = _availableTokenBalance;
        buyFee = _buyFee;
        sellFee = _sellFee;
    }
<...>
    function updateAvailableTokenBalance
      (uint256 _newAvailableTokenBalance) public onlyOwner {
        emit AvailableTokenUpdated
          (_newAvailableTokenBalance, availableTokenBalance);
>>      availableTokenBalance = _newAvailableTokenBalance;
    }
```

Consider implementing the corresponding check and changing `availableTokenBalance` variable and bonding curve in the same function to prevent any manipulations.