Stardusts Security Review

Pashov Audit Group

Conducted by: Said, ZeroTrust01, peanuts

December 19th 2024 - December 22nd 2024

Contents

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 **Daydream-Labs/stardusts-v1-contracts** repository was done by **Pashov Audit Group**, with a focus on the security aspects of the application's smart contracts implementation.

4. About Stardusts

Stardusts is a game-focused ERC20 token, enabling trading through a bonding curve and transitioning to Uniswap V2 pools after raising 16 ETH. The accompanying reward processor distributes player rewards and supports token buybacks.

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 - 13fe7d1e7739281c91daa4701fd94cb0432f827a

fixes review commit hash - 446f14beb6faca4115badec1cf427436a7e854d3

Scope

The following smart contracts were in scope of the audit:

- `RewardProcessor`
- `StardustToken`

7. Executive Summary

Over the course of the security review, Said, ZeroTrust01, peanuts engaged with Stardusts to review Stardusts. In this period of time a total of **12** issues were uncovered.

Protocol Summary

Protocol Name	Stardusts
Repository	https://github.com/Daydream-Labs/stardusts-v1-contracts
Date	December 19th 2024 - December 22nd 2024
Protocol Type	Bonding curve tokensale

Findings Count

Severity	Amount
Critical	1
High	1
Medium	2
Low	8
Total Findings	**12**

Summary of Findings

ID	Title	Severity	Status
[C-01]	AddLiquidityETH DoS by directly depositing WETH	Critical	Resolved
[H-01]	Incorrect fee calculation occurs on bonding curve buys	High	Resolved
[M-01]	An error in the minPayoutSize check in the sell() function	Medium	Resolved
[M-02]	An error in the minOrderSize check	Medium	Resolved
[L-01]	block.timestamp + 5 minutes does not provide any protection for the transaction timing	Low	Acknowledged
[L-02]	StardustToken can avoid the transfer fee by trading on Uniswap V3	Low	Acknowledged
[L-03]	An error in the calculation of currentEthPrice()	Low	Resolved
[L-04]	rescue allows to withdraw collected ETH rewards and boughtBackTokenBalances	Low	Resolved
[L-05]	StardustToken creation could always fail if pool is already created	Low	Resolved
[L-06]	settleRewards() can be called even when the token has not graduated	Low	Acknowledged
[L-07]	Upper limit of poolFeeBPS is too high	Low	Resolved
[L-08]	The latest bought tokens via the bonding curve are more expensive than in the pool	Low	Acknowledged

8. Findings

8.1. Critical Findings

[C-01] `AddLiquidityETH` DoS by directly depositing WETH

Severity

Impact: High

Likelihood: High

Description

When 16ETH is accumulated in the protocol, the protocol will call `_graduate()` to add 16ETH and 200M Stardust Tokens as liquidity. The minOut amount of Stardust Token is the same as the desired amount, which means the protocol doesn't expect anyone to provide liquidity before the protocol does.

```
function _graduate() internal {
        tokenState = TokenState.Trading;

        uint256 ethLiquidity = address(this).balance;

        // Mint the secondary market supply to this contract
        _mint(address(this), X1);

        // Approve the Uniswap V2 router to spend this token for adding
        // liquidity
        IERC20(address(this)).approve(swapRouter, X1);

        // Add liquidity to the Uniswap V2 pool
>       IUniswapV2Router02(swapRouter).addLiquidityETH{value: ethLiquidity}(
            address(this), X1, X1, ethLiquidity, owner
              (), block.timestamp + 5 minutes
        );

        emit Graduate(poolAddress, address(this), ethLiquidity, X1);
    }
```

While the `_update()` function prevents anyone from adding liquidity before "graduation", a malicious user can send 1 wei of WETH token into the pair address directly and call `sync()` to mess up the reserve balance.

Add this test to StardustTokenTest.t.sol and run `forge test --mt test1 -vv --via-ir`. The function should fail. with `UniswapV2Library: INSUFFICIENT_LIQUIDITY`.

```
function test1() public {
        address user = address(0xaaa);
        vm.startPrank(user);
        vm.deal(user, 25 ether);
        address router = 0x4752ba5DBc23f44D87826276BF6Fd6b1C372aD24;
        IERC20 WETH1 = IERC20(IUniswapV2Router02(stardustToken.swapRouter
          ()).WETH());
        deal(address(WETH1) ,user,10 ether);
        address factory = 0x8909Dc15e40173Ff4699343b6eB8132c65e18eC6;
        IUniswapV2Pair pair = IUniswapV2Pair(IUniswapV2Factory(factory).getPair
          (address(stardustToken), address(WETH1)));
        console.log("Pair Address:", address(pair));
        console.log(WETH1.balanceOf(address(pair)));

>        WETH1.transfer(address(pair), 1);
>        pair.sync();
        (uint112 reserve2, uint112 reserve3,) = pair.getReserves();
        console.log("Reserve0:", reserve2);
        console.log("Reserve1:", reserve3);
        console.log(WETH1.balanceOf(address(pair)));
        {
            // graduate buy
            uint256 actualEthCost = stardustToken.Y1() - stardustToken.Y0();
            uint256 fee = actualEthCost * stardustToken.feeBps() / 10000;
            uint256 totalRefund = 20 ether - (actualEthCost + fee);
            uint256 boughtStar = stardustToken.getOutTokenAmountAfterFee(0);
            // graduates when > 16 ether approx
            stardustToken.buy{value: 20 ether}(boughtStar);
        }
        (uint112 reserve0, uint112 reserve1,) = pair.getReserves();
        console.log("Reserve0:", reserve0);
        console.log("Reserve1:", reserve1);
    }
```

If those two highlight lines are removed, the test will pass, and reserve0,reserve1 should be 16e18:1.8e26.

```
Reserve0: 16000000000000000000
  Reserve1: 180000000000000000000000000
```

Recommendations

Before adding liquidity, the `_graduate()` function should call `skim()` and check if the reserve is manipulated. If it is, mint an equivalent amount of StardustToken directly to the pool and call `sync()`.

```
function _graduate() internal {
        tokenState = TokenState.Trading;

        uint256 ethLiquidity = address(this).balance;

        // Mint the secondary market supply to this contract
        _mint(address(this), X1);


        // Mint to Pair itself if someone already deposited directly into pair
>       // 1. Call skim first
        IUniswapV2Pair(poolAddress).skim(address(msg.sender));
        // 2. Check if reserve is not 0
        (uint112 reserve0, uint112 reserve1,) = IUniswapV2Pair
          (poolAddress).getReserves();
        uint balance;
        // 3. Mint the equivalent amount of StardustToken to the poolAddress
        // directly
        if (reserve0 != 0) {
            balance = reserve0 * X1 / ethLiquidity;
            _mint(address(poolAddress), balance);
        } else if (reserve1 != 0) {
            balance = reserve1 * X1 / ethLiquidity;
            _mint(address(poolAddress), balance);
        }
        // 4. Call sync() to match the reserve address
        IUniswapV2Pair(poolAddress).sync();



        // Approve the Uniswap V2 router to spend this token for adding
        // liquidity
        IERC20(address(this)).approve(swapRouter, X1);

        // Add liquidity to the Uniswap V2 pool
        IUniswapV2Router02(swapRouter).addLiquidityETH{value: ethLiquidity}(
        // Note that this strict slippage may result in failure, consider
        // setting it lower.
>            address(this), X1, X1, ethLiquidity, owner
  (), block.timestamp + 5 minutes
        );

        emit Graduate(poolAddress, address(this), ethLiquidity, X1);
    }
```

If the above is added, running the test again should pass with malicious `sync()`. Reserve0 will have 16e18 ETH and 1 wei, and reserve1 will have 1.8e26 token and 1 wei equivalent of stardust tokens.

```
If 1 wei is directly deposited:

  Reserve0: 16000000000000000001
  Reserve1: 180000000000000000012500000
```

Make sure the minOut for StardustToken is not too strict. The function will still succeed if 1 WETH is deposited in the contract, but best to make is zero.

```
If 1 WETH is directly deposited:

  Reserve0: 17000000000000000000
  Reserve1: 192500000000000000000000000
```

Lastly, test the above amended code again thoroughly before deployment, it is not battle tested.

8.2. High Findings

[H-01] Incorrect fee calculation occurs on bonding curve buys

Severity

Impact: Medium

Likelihood: High

Description

When users perform bonding curve buy and the `trueOrderSize` is greater than `maxRemainingTokens`, it will calculate `fee` based on `ethNeeded`.

```
function _validateBondingCurveBuy(uint256 minOrderSize)
        internal
        returns (
          uint256totalCost,
          uint256trueOrderSize,
          uint256fee,
          uint256refund,
          boolshouldGraduate
        )
    {
        // Set the total cost to the amount of ETH sent
        totalCost = msg.value;

        // Calculate the fee
>>>     fee = (totalCost * feeBps) / 10000;

        // Calculate the amount of ETH remaining for the order
        uint256 remainingEth = totalCost - fee;

        // ...

        // If the order size is greater than the maximum number of remaining
        // tokens:
        if (trueOrderSize > maxRemainingTokens) {
            // Reset the order size to the number of remaining tokens
            trueOrderSize = maxRemainingTokens;

            // Calculate the amount of ETH needed to buy the true order
            uint256 ethNeeded = Y1 - virtualEthLiquidity;

            // Recalculate the fee with the updated order size
>>>         fee = (ethNeeded * feeBps) / 10000;

            // Recalculate the total cost with the updated order size and fee
            totalCost = ethNeeded + fee;

            // Refund any excess ETH
            if (msg.value > totalCost) {
                refund = msg.value - totalCost;
            }

            shouldGraduate = true;
        }

    }
```

It can be observed that when `trueOrderSize` is greater than `maxRemainingTokens`, the `fee` is based on `totalCost`, not the ETH needed to buy. This leads to an underestimation of the fee when `trueOrderSize` exceeds `maxRemainingTokens`.

Recommendations

Change the fee calculation as follows :

```diff
if (trueOrderSize > maxRemainingTokens) {
            // Reset the order size to the number of remaining tokens
            trueOrderSize = maxRemainingTokens;

            // Calculate the amount of ETH needed to buy the true order
            uint256 ethNeeded = Y1 - virtualEthLiquidity;

            // Recalculate the fee with the updated order size
+            totalCost = (10000 * ethNeeded ) / (10000 - feeBps);
-            fee = (ethNeeded * feeBps) / 10000;
+            fee = totalCost - ethNeeded;

            // Recalculate the total cost with the updated order size and fee
-            totalCost = ethNeeded + fee;

            // Refund any excess ETH
            if (msg.value > totalCost) {
                refund = msg.value - totalCost;
            }

            shouldGraduate = true;
        }
```

8.3. Medium Findings

[M-01] An error in the `minPayoutSize` check in the `sell()` function

Severity

Impact: Medium

Likelihood: Medium

Description

```
function sell(
   uint256tokensToSell,
   uint256minPayoutSize
 ) external nonReentrant returns (uint256
        // Ensure the sender has enough liquidity to sell
        if (tokensToSell > balanceOf(msg.sender)) revert InsufficientBalance();

        // Initialize the true payout size
        uint256 truePayoutSize;

        if (tokenState == TokenState.Trading) {
            truePayoutSize = _handleExchangeSell(tokensToSell, minPayoutSize);
        } else if (tokenState == TokenState.Raising) {
@>            truePayoutSize = _handleBondingCurveSell
  (tokensToSell, minPayoutSize);
            // Calculate the fee
            uint256 fee = (truePayoutSize * feeBps) / 10000;

            // Calculate the payout after the fee
@>             uint256 payoutAfterFee = truePayoutSize - fee;

            // Update the virtual ETH liquidity
            virtualEthLiquidity -= payoutAfterFee;

            // Handle the fees
            _distributeFees(fee);

            // Send the payout to the recipient
            Address.sendValue(payable(msg.sender), payoutAfterFee);
        }

        // truePayoutSize will be 0 if the sell is done through uniswap v2 pool
        // need to fetch that info from the pool events
        emit Trade(
          msg.sender,
          false,
          tokensToSell,
          truePayoutSize,
          currentEthPrice
        ), block.timestamp
        return truePayoutSize;
    }
```

`minPayoutSize` is a slippage protection parameter set by the user, meaning it guarantees that the user will receive at least minPayoutSize ETH in the end. Checking truePayoutSize does not achieve this protection, because payoutAfterFee could still be smaller than minPayoutSize.

Recommendations

Remove the minPayoutSize check from the _handleBondingCurveSell() function and move it to the sell() function, as follows:

```
// Calculate the payout after the fee
             uint256 payoutAfterFee = truePayoutSize - fee;
+             if (payoutAfterFee < minPayoutSize) revert SlippageTooHigh();
```

[M-02] An error in the `minOrderSize` check

Severity

Impact: Medium

Likelihood: Medium

Description

```
function _validateBondingCurveBuy(uint256 minOrderSize)
        internal
        returns (
          uint256totalCost,
          uint256trueOrderSize,
          uint256fee,
          uint256refund,
          boolshouldGraduate
        )
    {
        // Set the total cost to the amount of ETH sent
        totalCost = msg.value;

        // Calculate the fee
        fee = (totalCost * feeBps) / 10000;

        // Calculate the amount of ETH remaining for the order
        uint256 remainingEth = totalCost - fee;

        // Get quote for the number of tokens that can be bought with the amount
        // of ETH remaining
        trueOrderSize = getOutTokenAmount(remainingEth);

        // Ensure the order size is greater than the minimum order size
@>        if (trueOrderSize < minOrderSize) revert SlippageTooHigh();

        // Calculate the maximum number of tokens that can be bought on the
        // bonding curve
        uint256 maxRemainingTokens = (X0 - X1) - totalSupply();

        // Start the market if the order size equals the number of remaining
        // tokens
        if (trueOrderSize == maxRemainingTokens) {
            shouldGraduate = true;
        }

        // If the order size is greater than the maximum number of remaining
        // tokens:
        if (trueOrderSize > maxRemainingTokens) {
            // Reset the order size to the number of remaining tokens
@>            trueOrderSize = maxRemainingTokens;

            // Calculate the amount of ETH needed to buy the true order
            uint256 ethNeeded = Y1 - virtualEthLiquidity;

            // Recalculate the fee with the updated order size
            fee = (ethNeeded * feeBps) / 10000;

            // Recalculate the total cost with the updated order size and fee
            totalCost = ethNeeded + fee;

            // Refund any excess ETH
            if (msg.value > totalCost) {
                refund = msg.value - totalCost;
            }

            shouldGraduate = true;
        }
    }
```

It can be observed that under the condition trueOrderSize > maxRemainingTokens, trueOrderSize is reassigned, but there is no check on minOrderSize. This can lead to situations where a transaction is frontrun, causing the user’s minOrderSize to not provide the intended slippage protection.

Recommendations

Moving the check if (trueOrderSize < minOrderSize) revert SlippageTooHigh(); to the end of the _validateBondingCurveBuy() function to ensure that the minOrderSize check is applied after all other conditions

8.4. Low Findings

[L-01] `block.timestamp + 5 minutes` does not provide any protection for the transaction timing

```
IUniswapV2Router02
  (swapRouter).swapExactETHForTokensSupportingFeeOnTransferTokens{value: msg.value}(
                minAmountOut, path, msg.sender, block.timestamp + 5 minutes
            );
```

When trading on UniswapV2, use the parameter block.timestamp + 5 minutes is entirely equivalent to block.timestamp, because block.timestamp reflects the timestamp of the block in which the transaction is included. For example, if the transaction is initiated at T0, it could still be included in a block at T0 + 1 hour.

If the intent is to ensure the transaction is included in a block within 5 minutes of initiation and discarded otherwise, the correct approach would be to use the specific current system time as a reference rather than the blockchain variable block.timestamp.

[L-02] StardustToken can avoid the transfer fee by trading on Uniswap V3

```
if (tokenState == TokenState.Trading && isTradingFromDex && !isPrizePool) {
                uint256 feeAmount = (value * feeBps) / 10000;
                uint256 protocolFee = feeAmount * protocolFeeBps / 10000;
                uint256 prizePoolFee = feeAmount - protocolFee;
                uint256 transferAmount = value - feeAmount;

                // Transfer fee to treasury and prize pool
                super._update(from, treasury, protocolFee);
                super._update(from, prizePool, prizePoolFee);

                // Transfer remainder to user
                super._update(from, to, transferAmount);
            }
```

In the _update() function, the transfer fee is only applied to tokens traded on the Uniswap V2 pool. As a result, users can place buy or sell orders on

Uniswap V3 at prices close to those on Uniswap V2, allowing them to trade without paying any transfer fees. This may be a known issue.

[L-03] An error in the calculation of `currentEthPrice()`

```
function currentEthPrice() public view returns (uint256) {
        uint256 ethLiquidity = address(this).balance < Y0 ? Y0 : address
          (this).balance;
        if (tokenState == TokenState.Raising) {
            uint256 virtualTokenLiquidity = X0 - totalSupply();
            return ethLiquidity * 1e18 / virtualTokenLiquidity;
        } else if (tokenState == TokenState.Trading) {
            (uint112 reserve0, uint112 reserve1,) = IUniswapV2Pair
              (poolAddress).getReserves();
            address token0 = IUniswapV2Pair(poolAddress).token0();
            (uint112 reserveToken, uint112 reserveETH) =
                address(this) == token0 ? (reserve0, reserve1) :
                  (reserve1, reserve0);

            return (uint256(reserveETH) * 1e18) / uint256(reserveToken);
        }

        return 0;
    }
```

Based on the bonding curve characteristics, the token price relative to ETH is determined by virtualEthLiquidity, not by address(this).balance.

Therefore:

```
-        uint256 ethLiquidity = address(this).balance < Y0 ? Y0 : address
- (this).balance;
+        uint256 ethLiquidity = virtualEthLiquidity
```

[L-04] `rescue` allows to withdraw collected ETH rewards and `boughtBackTokenBalances`

`rescue` does not restrict the amount of ETH that can be withdrawn by the caller and does not verify whether the token to be withdrawn has `boughtBackTokenBalances`. Consider adding these verifications to prevent `rescue` from causing issues within `RewardProcessor`.

```
function rescue
      (address token, uint256 amount, address to) external onlyOwner {
        // rescue eth
        if (token == address(0)) {
+           if (amount > address
+ (this).balance - totalPrizeBalance) revert InvalidInput();
            Address.sendValue(payable(to), amount);
        } else {
+           if (amount > IERC20(token).balanceOf(address
+ (this)) - boughtBackTokenBalances[token]) revert InvalidInput();
            IERC20(token).transfer(to, amount);
        }
    }
```

[L-05] `StardustToken` creation could always fail if pool is already created

When the `StardustToken` is first created, it will attempt to create a V2 pool with WETH and the `StardustToken`.

```
function _createPool() internal returns (address) {
        // Get the factory and WETH address from the router
        address factory = IUniswapV2Router02(swapRouter).factory();
        address wethAddress = IUniswapV2Router02(swapRouter).WETH();
>>>     address pair = IUniswapV2Factory(factory).createPair(address
  (this), wethAddress);
        return pair;
    }
```

However, it is possible that an attacker that can predict the address of `StardustToken`, front runs the creation of the pool and causes the `createPool` to revert because the pool already exists. Consider trigger `createPair` only if a pool does not yet exist.

```
function _createPool() internal returns (address) {
        // Get the factory and WETH address from the router
        address factory = IUniswapV2Router02(swapRouter).factory();
        address wethAddress = IUniswapV2Router02(swapRouter).WETH();
+       address pair = IUniswapV2Factory(factory).getPair(address
+ (this), wethAddress);
+       (if pair == address(0)) {
+         pair = IUniswapV2Factory(factory).createPair(address
+ (this), wethAddress);
+       }
-        address pair = IUniswapV2Factory(factory).createPair(address
- (this), wethAddress);
        return pair;
    }
```

[L-06] `settleRewards()` can be called even when the token has not graduated

In the `settleReward()` function, the idea is to swap part of the `settleAmount` (in ETH) to stardust tokens and distribute the rest to users. Even if the token has not graduated, the function still can be called, skipping the swap and directly giving out ETH to users.

```
function settleRewards(
        address tokenToBuyBack,
        uint256 amountOutMin,
        uint256 settleAmount,
        address[] calldata users,
        uint256[] calldata rewardBps
    ) public onlyOperator {
        if
          (users.length != rewardBps.length || settleAmount > totalPrizeBalance) {
        ...
        uint256 ethForBuyBack = settleAmount * buyBackBpsIfGraduated / 10000;
        if (IStardustToken(tokenToBuyBack).tokenState
          () == IStardustToken.TokenState.Trading && ethForBuyBack > 0) {
            settleAmount -= ethForBuyBack;
            totalPrizeBalance -= ethForBuyBack;
            uint256 tokensReceived;

            ...
            emit BuyBack(tokenToBuyBack, ethForBuyBack, tokensReceived);
        }
>       // Update user rewards
        uint256 totalBps;
        for (uint256 i = 0; i < users.length; i++) {
```

If not intended, ensure that the function only runs if `IStardustToken(tokenToBuyBack).tokenState() == IStardustToken.TokenState.Trading`.

```
if (users.length != rewardBps.length || settleAmount > totalPrizeBalance) {
            revert InvalidInput();
        }
>    if (IStardustToken(tokenToBuyBack).tokenState
  () != IStardustToken.TokenState.Trading){
            revert InvalidInput();
        }
```

[L-07] Upper limit of `poolFeeBPS` is too high

In the StardustToken contract, the owner can set the `feeBps` to 100% (10000). Either the owner can set it to 100% to gain all the tokens when the user swaps their token in the liquidity pool, or the user unknowingly pays more fees than intended if they did not know that the fees will be changing.

Set a max cap of <100% (eg 20%) to reduce centralization risk.

```
function setFeeBps(uint256 newFeeBps) public onlyOwner {
        if (newFeeBps >= 10000) {
            revert InvalidConfig();
        }
        feeBps = newFeeBps;
        emit SetFeeBps(newFeeBps);
    }
```

Set a max fee of 10% (1000) instead.

```
function setFeeBps(uint256 newFeeBps) public onlyOwner {
        if (newFeeBps >= 1000) {
            revert InvalidConfig();
        }
        feeBps = newFeeBps;
        emit SetFeeBps(newFeeBps);
    }
```

[L-08] The latest bought tokens via the bonding curve are more expensive than in the pool

As the `totalSupply` (bought tokens) approaches the max buyable tokens (`x0 - x1`), the price of tokens in the bonding curve increases. This means the last tokens purchased are more expensive than the first tokens bought in the pool.

Since tokens and ETH will be added to the pool as liquidity once all the buyable tokens are purchased, there is an incentive to wait for other users to buy the last tokens in the bonding curve and then purchase them in the pool at a lower price.

```
function testBuy_Price_Issue() public {
        address user = address(0xaaa);
        address buyer = address(0xbbb);
        address weth = IUniswapV2Router02(stardustToken.swapRouter()).WETH();
        uint256 buyAmount =  5_000_000 * 10 ** 18;

        vm.startPrank(user);
        vm.deal(user, 20 ether);
        uint256 maxTokenBuy = stardustToken.X0() - stardustToken.X1
          () - buyAmount;
        uint256 allBuyEthRequired = (stardustToken.K() / (stardustToken.X0
          () - maxTokenBuy)) - stardustToken.virtualEthLiquidity();
        uint256 buyEthWithFee = (10000 * allBuyEthRequired ) /
          (10000 - stardustToken.feeBps());
        stardustToken.buy{value: buyEthWithFee}(0);
        vm.stopPrank();

        vm.startPrank(buyer);
        vm.deal(buyer, 5 ether);
        uint256 balanceBefore = buyer.balance;
        stardustToken.buy{value: 5 ether}(0);
        console.log("buyer cost (bonding curve) : ");
        console.log(balanceBefore - buyer.balance);
        console.log("buyer balance (Token) after (bonding curve): ");
        console.log(stardustToken.balanceOf(buyer));

        // buy exact token
        balanceBefore = buyer.balance;
        // update buy amount to make it identical with previous received
        buyAmount = stardustToken.balanceOf(buyer);
        address[] memory path = new address[](2);
        path[0] = weth;
        path[1] = address(stardustToken);
        uint256 buyTokenWithFee = (10000 * buyAmount) /
          (10000 - stardustToken.feeBps());
        IUniswapV2Router02(stardustToken.swapRouter(
          stardustToken.swapRouter

        )
        console.log("buyer cost (trading on pool) : ");
        console.log(balanceBefore - buyer.balance);
        console.log("buyer balance (Token) after (buy on pool): ");
        console.log(stardustToken.balanceOf(buyer));
        vm.stopPrank();
    }
```

Output :

```
Logs:
  buyer cost (bonding curve) :
  5365853658536585365
  buyer balance (Token) after (bonding curve):
  5000000000000000002050000

  buyer cost (trading on pool) :
  5110874023343917111
  buyer balance (Token) after (buy on pool):
  10000000000000000004100000
```

Adjust the initial liquidity proportion so that the last tokens bought in the bonding curve are not more expensive than the first tokens bought in the pool.