LAMBO.WIN
Security Review

DECEMBER, 2024

www.chaindefenders.xyz
https://x.com/ChDefendersEth

Lead Auditors

PeterSR

0x539.eth

Table of Contents

Protocol Summary

In the realm of cryptocurrency, liquidity is paramount, especially for nascent Meme projects. Liquidity refers to the ease with which assets can be bought or sold in the market without affecting their price. For cryptocurrency projects, having sufficient liquidity is crucial for several reasons.

To search for an efficient, near-zero cost liquidity solution for token launch, we propose a straightforward mechanism: the concept of virtual liquidity to meet the liquidity needs of project parties. We will establish a deep liquidity pool on

UniswapV3 to satisfy the liquidity exit for users' buying and selling activities. Essentially, this mechanism involves whales providing liquidity to retail investors. Liquidity providers can earn profits through LP fees, creating a win-win situation where whales earn LP fees and retail investors/developers resolve their liquidity issues.

We believe this mechanism can enhance the liquidity returns for DeFi whales while simultaneously addressing the liquidity challenges faced by retail investors and developers.

Disclaimer

The ChainDefenders team makes all effort to find as many vulnerabilities in the code in the given time period, but holds no responsibilities for the findings provided in this document. A security audit by the team is not an endorsement of the underlying business or product. The audit was time-boxed and the review of the code was solely on the security aspects of the Solidity implementation of the contracts.

Risk Classification

Likelihood/Impact	High	Medium	Low
High	H	H/M	M
Medium	H/M	M	M/L
Low	M	M/L	L

Audit Details

Scope

Id	Files in scope
1	LamboFactory.sol
2	LamboToken.sol
3	LamboVEthRouter.sol
4	VirtualToken.sol
5	LamboRebalanceOnUniwap.sol
6	LaunchPadUtils.sol
7	UniswapV2Library.sol

Roles

Id	Roles
1	Owner
2	LamboRouter
3	LamboFactory
4	RebalanceContract
5	User

Executive Summary

Issues found

Severity	Count	Description
High	0	Critical vulnerabilities
Medium	0	Significant risks
Low	1	Minor issues with low impact
Informational	2	Best practices or suggestions
Gas	2	Optimization opportunities

Severity	Count	Description

Findings

High

High 01 Msg.value used instead of amount

Finding description and impact

When the `cashIn` function is called with an `underlyingToken` that is not `LaunchPadUtils.NATIVE_TOKEN` (i.e., not ETH), the amount of tokens minted to the caller is incorrect. Specifically, the `_mint` function uses `msg.value` instead of the `amount` parameter, which results in an incorrect token amount being minted when the function is called with a non-ETH token.

This issue could lead to an under-minting or over-minting of tokens, causing financial loss to users or a potential inflation of the token supply. Also it can cause a loss for the user. If exploited, it could compromise the integrity of the tokenomics and result in trust issues with the protocol.

Proof of Concept

1. Assume `underlyingToken` is a non-ETH token.
2. A user calls `cashIn` with `amount = 1000` and `msg.value = 0`.
3. The `_transferAssetFromUser(amount)` function transfers the correct `amount` of the underlying token from the user.
4. However, `_mint` uses `msg.value` (which is 0), leading to no tokens being minted for the user.
5. This results in the user losing 1000 tokens without receiving any minted tokens in return.

Recommended Mitigation Steps

Use the `amount` parameter as the basis for minting when `underlyingToken` is not `LaunchPadUtils.NATIVE_TOKEN`.

Modify the `_mint` function call as follows:

```
_mint(msg.sender, underlyingToken == LaunchPadUtils.NATIVE_TOKEN ? msg.
value : amount);
```

High 01 Preview and rebalance should be executed in one transaction

Finding description and impact

Currently in `LamboRebalanceOnUniwap` `previewRebalance` and `rebalance` are two different functions. Which means if they are executed in different transaction, most probably the result that is returned by `previewRebalance` will be no more correct, when executing `rebalance`. Because `previewRebalance` is returning the input params which are passed to the `rebalance` function.

Most probably the current approach will not work correctly and it both operations should be executed in one transaction.

```
function rebalance(
    uint256 directionMask,
    uint256 amountIn,
    uint256 amountOut
)
    external
    nonReentrant
{
    uint256 balanceBefore = IERC20(weth).balanceOf(address(this));

    bytes memory data = abi.encode(directionMask, amountIn, amountOut);
    IMorpho(morphoVault).flashLoan(weth, amountIn, data);

    uint256 balanceAfter = IERC20(weth).balanceOf(address(this));
    uint256 profit = balanceAfter - balanceBefore;

    require(profit > 0, "No profit made");
}

```

```
function previewRebalance()
    public
    view
    returns (
        bool result,
        uint256 directionMask,
        uint256 amountIn,
        uint256 amountOut
    )
{
    address tokenIn;
    address tokenOut;

    (tokenIn, tokenOut, amountIn) = _getTokenInOut();
    (amountOut, directionMask) = _getQuoteAndDirection(tokenIn,
   tokenOut, amountIn);

    result = amountOut > amountIn;
}
```

Proof of Concept

Let's have the following situation:

1. Call `previewRebalance`: Execute `previewRebalance` to get the `directionMask`, `amountIn`, and `amountOut` values.
2. Swap: Another user initiates swap and changes the state of the pool.
3. Execute `rebalance`: Admin calls the `rebalance` function using the values returned from `previewRebalance`.
4. Due to the change in the step 2, rebalance will not work as expected.

Recommended mitigation steps

Refactor the contract to execute both the `previewRebalance` and `rebalance` logic within a single transaction to maintain state consistency.

Medium

Mid 01 Rebalance is not calculate correctly

Finding description and impact

The `rebalance` function fails to operate correctly because it relies on the `previewBalance` function, which calculates `amountIn` based solely on the token balances. This approach is incompatible with Uniswap V3's mechanics.

Code Snippet

```
function _getTokenInOut()
    internal
    view
    returns (address tokenIn, address tokenOut, uint256 amountIn)
{
    (uint256 wethBalance, uint256 vethBalance) = _getTokenBalances();
    uint256 targetBalance = (wethBalance + vethBalance) / 2;

    if (vethBalance > targetBalance) {
        amountIn = vethBalance - targetBalance;
        tokenIn = weth;
        tokenOut = veth;
    } else {
        amountIn = wethBalance - targetBalance;
        tokenIn = veth;
        tokenOut = weth;
    }

    require(amountIn > 0, "amountIn must be greater than zero");
}
```

The primary issue is the calculation of `amountIn` based on token balances (`wethBalance` and `vethBalance`). In Uniswap V3, this approach is fundamentally flawed due to the following reasons:

Impact of Fees:

- The token balances (`balanceOf`) include accrued fees from trades, which are not reinvested into the pool's liquidity. Consequently, these balances do not accurately reflect the pool's price or liquidity distributio

Proof of Concept

Consider the following scenario:

1. The pool is a VETH/WETH pair, where the token balances are `balanceOf(VETH) = Q1` and `balanceOf(WETH) = Q2`.
2. These balances (`Q1` and `Q2`) include fees collected from trades, which are not used to calculate the pool's price.
3. As a result, the `rebalance` function derives an incorrect `amountIn` value, leading to erroneous swap amounts.

Recommended mitigation steps

To ensure accurate calculations:

1. Utilize the pool's `slot0()` function to retrieve the current square root price ratio of the two tokens.
2. Use the retrieved price to compute the appropriate swap amount for rebalancing.
3. Avoid relying on `balanceOf` values for determining the pool's price or liquidity state.

Mid 02 DOS of `createLaunchPadAndInitialBuy`

Finding description and impact

Currently the function `createLaunchPadAndInitialBuy` is doing the following things:

1. Creates a lamboToken and a uniswap v2 pool for the pair of `lamboToken` and `VETH`.
2. Takes a loan for the pool of VETH tokens.
3. Then all shares that are minted for this pool are send to address(0).
4. Creator initiates a buy quote token, swapping eth/veth for lamboToken.

The problem here is that the VETH loan function has a limit per block. Which means that, when the limit is reached(300 * 1e18), no other user can't create a new lamboToken and pool for this token in this block. The problem is that malicious user can frontrun all request and set 300 * 1e18 as virtualLiquidityAmount. This will prevent any other loan to be created in this block. This could be executed with small amount of eth, becuase for the `buyAmount` can be used 1.

```
function createLaunchPadAndInitialBuy(
    address lamboFactory,
    string memory name,
    string memory tickname,
    uint256 virtualLiquidityAmount,
    uint256 buyAmount
)
    public
    payable
    nonReentrant
    returns (address quoteToken, address pool, uint256 amountYOut)
{
    require(
        VirtualToken(vETH).isValidFactory(lamboFactory),
        "only ValidFactory"
    );

    (quoteToken, pool) = LamboFactory(lamboFactory).createLaunchPad(
        name,
        tickname,
        virtualLiquidityAmount,
        address(vETH)
    );

    amountYOut = _buyQuote(quoteToken, buyAmount, 0);
}
```

Proof of Concept

Let's have the following scenario:

1. Malicious user frontruns all other request for `createLaunchPadAndInitialBuy` using the max value for `virtualLiquidityAmount`, which is accepted as loan amount per block.
2. Malicious user sets buyAmount = 1. So the only money that he will pay is for gas prices.
3. No other user can create a lambo token for this block.

Recommended mitigation steps

Consider creating a whitelist of which user can create new lambo tokens and pools.