Smart Contract Security Audit

rev 1.0

Prepared for

MESHswap

Prepared by

Theori

May 6, 2022

Theori

Table of Contents

Executive Summary

Theori reviewed the MESHswap smart contracts which are based on the previously audited KlaySwap smart contracts. The biggest changes from KlaySwap are the introduction of functions for Uniswap compatibility and adjusting the contracts to work on the Polygon blockchain. During the review, we found that the new functions for Uniswap compatibility introduced new attack surfaces with critical vulnerabilities. The safety of KlaySwap and Uniswap rely on a set of different assumptions, so when combining their code together, care must be taken to ensure that the assumptions are still valid. Integrators relying on the Uniswap functions in KlaySwap must also verify that their assumptions still hold, as even if the function names are the same, the behavior may be different.

Scope

We were provided with a stable source code tree to review. We also reviewed each of the committed fixes.

Source code:

- KlaySwap/theori-audit-swap
  - b13d48f446c18e4a175e201a5c1749ce19a3d981 (initial)
  - ...
  - 63f460509da5747901a897a6c63134cb0a276ef2 (final)

Findings

These are the potential issues that may have correctness and/or security impacts.

Summary

#	Title	Severity
1	Use SafeERC20 library instead of TransferHelper library	Low
2	*getCurrentPool* is unsafe with *skim* function	Medium
3	*exchangeNeg* and *exchangePos* missing *nonReentrant* modifier	Critical
4	*initPool* missing call to *_update*	Medium
5	Missing authorization checks in *removeLiquidityWithLimit* and *removeLiquidityWithLimitETH*	Critical
6	*transferFrom* does not implement EIP-20 semantics	Critical
7	*addETHLiquidityWithLimit* has incorrect behavior	Medium
8	*addETHLiquidityWithLimit* refunds to user instead of *msg.sender*	Low
9	Possible race condition during pool initialization	Low
10	*sendReward* should ensure tokenomics are not violated	Low
11	Typo in *getAmountIn* results in incorrect behavior	Low
12	*liquidate* should have *nonReentrant* modifier	Low
13	BuybackFund may be vulnerable to sandwich attacks	Low

Issue #1: Use SafeERC20 library instead of TransferHelper library

Summary	Severity
The TransferHelper library does not check that a token address contains code which could be unsafe. Use OpenZeppelin's SafeERC20 library which has additional safety checks.	Low

While TransferHelper is safe within the context of Uniswap, it has contributed to exploits in other smart contracts as it lacks some useful safety checks, such as checking that a token address contains code. The gas benefits of using TransferHelper is minor compared to the safety benefits of using SafeERC20, especially on the Polygon blockchain where gas fees are minimal.

```
function safeTransferFrom(
    address token, address from, address to, uint256 value
) internal {
    // bytes4(keccak256(bytes('transferFrom(address,address,uint256)')));
    (bool success, bytes memory data) = token.call(abi.encodeWithSelector(0x23b872dd,
from, to, value));
    require(
        success && (data.length == 0 || abi.decode(data, (bool))),
        'TransferHelper::transferFrom: transferFrom failed'
    );
}
```

Severity is low as it is unlikely that this will cause any security impacts by itself.

Fix

Replace all instances of TransferHelper with SafeERC20 (commit c1b056a4).

Issue #2: getCurrentPool is unsafe with skim function

Summary	Severity
In Exchange.impl.sol, *getCurrentPool* retrieves the balance of each token in the pool. After the introduction of the *skim* function, this is incorrect because an attacker can transfer tokens to the pool then withdraw the tokens with *skim*. This may lead to incorrect behavior of functions that rely on *getCurrentPool*.	Medium

getCurrentPool is used internally and externally to determine the ratio of tokens in the pool as well as the value of each liquidity pool token. In its original form, *getCurrentPool* returned the balance of each token in the pool, which could be manipulated by an attacker who transfers additional tokens to the pool. The attacker could add tokens, call a contract that utilizes *getCurrentPool*, and then get the tokens back with a call to *skim*.

However, an attacker must avoid calling any function that calls *_update* as then *skim* will have no effect. Due to this restriction, we were unable to identify any definitely profitable attacks using only this issue. This is a significant and unnecessary attack surface, though.

```
function getCurrentPool() public view returns (uint, uint) {
    uint pool0 = IERC20(token0).balanceOf(address(this));
    uint pool1 = IERC20(token1).balanceOf(address(this));

    return (pool0, pool1);
}

// force balances to match reserves
function skim(address to) external nonReentrant {
    address _token0 = token0; // gas savings
    address _token1 = token1; // gas savings
    IERC20(_token0).transfer(to,
IERC20(_token0).balanceOf(address(this)).sub(reserve0));
    IERC20(_token1).transfer(to,
IERC20(_token1).balanceOf(address(this)).sub(reserve1));
}
```

Severity is medium as there are no examples of lost funds due to only this issue, however this issue introduces significant risks of vulnerabilities in other code.

Fix

Replace logic of *getCurrentPool* with *getReserves*, and remove the logic in *skim* (commit fde686a8).

Either of these changes would be sufficient to fix this issue. We believed that the *skim* function was an unnecessary risk as a user never needs to transfer tokens to the MESHswap pool directly, and there should never be a need to call *skim*. Hence, we recommended that *skim* is neutered.

Issue #3: exchangeNeg and exchangePos missing nonReentrant modifier

Summary	Severity
In Exchange.impl.sol, *exchangeNeg* and *exchangePos* are missing the *nonReentrant* modifier. This leads to a critical reentrancy bug when combined with *removeLiquidityWithLimitETH*.	Critical

exchangeNeg and *exchangePos* are missing the *nonReentrant* modifier. An attacker could call *removeLiquidityWithLimitETH* and then call into *exchangeNeg* before *sendToken* is called. As *getCurrentPool* returns the current balance, *exchangeNeg* will believe that *token0* is worth more than it should, because the *token1* assets have not yet been transferred out.

The fixes for issue #2 help mitigate the exploitability of this issue, but it is desirable to explicitly fix this issue by adding the *nonReentrant* modifier.

```
function exchangeNeg(address token, uint amount) public returns (uint) {
    // ...
}

function removeLiquidityWithLimitETH(uint amount, uint minAmount0, uint minAmount1,
address user) public nonReentrant returns (uint, uint) {
    // ...

    decreaseTotalSupply(amount);
    decreaseBalance(user, amount);

    emit Transfer(user, address(0), amount);

    if (amount0 > 0) {
        IWETH(WETH).withdraw(amount0);
        (bool success, ) = user.call.value(amount0)("");
        require(success, "transfer failed");
    }
    if (amount1 > 0) sendToken(token1, amount1, user);

    _update();
```

```
    emit RemoveLiquidity(user, token0, amount0, token1, amount1, amount);
    return (amount0, amount1);
}
```

Severity is critical as we could demonstrate a loss of funds, in pools where one of the tokens is WETH, due to this issue.

Fix

Add *nonReentrant* modifier to *exchangeNeg* and *exchangePos* (commit fde686a8). The fixes to issue #2 provide additional mitigations as well.

Issue #4: initPool missing call to _update

Summary	Severity
In Exchange.impl.sol, *initPool* is missing a call to *_update*. An attacker could immediately call skim to steal the initial assets due to missing initialization of *reserve0* and *reserve1*.	Medium

In Factory.impl.sol, *createPool* creates a new pool contract, transfers assets, and then calls *initPool* to initialize the pool's state. However, *initPool* does not initialize *reserve0* and *reserve1*, nor does it call *_update* which would initialize these variables. An attacker could immediately call *skim* which would transfer the initial assets to the attacker.

The fixes to issue #2 may mitigate this issue, but it is important that all state is correctly initialized.

```
function initPool(address to) public {
    require(msg.sender == factory);

    (uint amount, ) = getCurrentPool();

    IGovernance(IFactory(factory).owner()).acceptEpoch();

    increaseTotalSupply(amount);
    increaseBalance(to, amount);

    emit Transfer(address(0), to, amount);  // mint event

    string memory symbolA = getTokenSymbol(token0);
    string memory symbolB = getTokenSymbol(token1);

    name = string(abi.encodePacked(name, " ", symbolA, "-", symbolB));

    decimals = IERC20(token0).decimals();
    WETH = IFactoryImpl(factory).WETH();
}
```

Severity is medium as the window of attack is small and only a small amount of assets are at risk, but assets are at risk. The attacker must call *skim* after the pool is created but before any other operations on the pool, and as such, only the initial assets are at risk.

Fix

Refactor *createPool* and *initPool* (commit deca7594). This includes adding a call to *_update* at the end of *initPool*.

Issue #5: Missing authorization checks in removeLiquidityWithLimit and removeLiquidityWithLimitETH

Summary	Severity
In Exchange.impl.sol, an attacker can provide an arbitrary user address to *removeLiquidityWithLimit* and *removeLiquidityWithLimitETH*, which do not have any authorization checks. This allows an attacker to withdraw any amount of liquidity from the pool for any user, potentially resulting in loss of funds due to slippage.	Critical

removeLiquidityWithLimit and *removeLiquidityWithLimitETH* allowed any caller to remove the liquidity for any user. An attacker can abuse this by manipulating the ratio of tokens in the pool, and then forcing some liquidity providers to remove their liquidity at a potential loss. Alternatively, an attacker could remove a significant amount of liquidity to make it easier for the attacker to manipulate the price of a pool. In either case, this would result in significant headaches for the liquidity providers and possible loss of assets.

These functions should either remove liquidity for only *msg.sender* or they should check that *msg.sender* is authorized to remove liquidity for *user*.

```
function removeLiquidityWithLimit(uint amount, uint minAmount0, uint minAmount1,
address user) public nonReentrant returns (uint, uint) {
    // ...

    decreaseTotalSupply(amount);
    decreaseBalance(user, amount);

    emit Transfer(user, address(0), amount);

    if (amount0 > 0) sendToken(token0, amount0, user);
    if (amount1 > 0) sendToken(token1, amount1, user);

    _update();
    emit RemoveLiquidity(user, token0, amount0, token1, amount1, amount);
    return (amount0, amount1);
}
```

```
function removeLiquidityWithLimitETH(uint amount, uint minAmount0, uint minAmount1,
address user) public nonReentrant returns (uint, uint) {
    // ...
}
```

Severity is critical as liquidity providers may lose assets. Additionally, arbitrarily removing liquidity could result in lost funds if unsupported by an integrator.

Fix

Provide *msg.sender* as the argument to *decreaseBalance* (commit fde686a8).

Issue #6: transferFrom does not implement EIP-20 semantics

Summary	Severity
In Exchange.impl.sol, the logic for *transferFrom* decreases the balance for *to* instead of *msg.sender*. This allows an attacker to move assets from a user to a contract they have approved, which may result in a loss of funds for the user.	Critical

transferFrom should check that *msg.sender* is approved to transfer the assets and then decrease the approval of *msg.sender*. However, the implementation in Exchange.impl.sol checks and modifies the approval for *_to*, which may not be *msg.sender*. This could allow an attacker to cause a loss of funds depending on which addresses users have approved. Even without the loss of funds, the current behavior is incorrect and unexpected.

```
function transferFrom(address _from, address _to, uint _value) public nonReentrant
returns (bool) {
    decreaseBalance(_from, _value);
    increaseBalance(_to, _value);

    allowance[_from][_to] = allowance[_from][_to].sub(_value);

    emit Transfer(_from, _to, _value);

    return true;
}
```

Severity is critical as it could result in lost funds, though the risk is lower than other critical issues in this report.

Fix

Subtract *_value* from the allowance of *msg.sender* (commit fde686a8).

Issue #7: addETHLiquidityWithLimit has incorrect behavior

Summary	Severity
In Exchange.impl.sol, the logic for *addETHLiquidityWithLimit* is incorrect and may result in a loss of funds for the user if they have assets in WETH and have approved the contract to transfer these assets.	Medium

addETHLiquidityWithLimit deposits the provided native coins in the WETH contract, so the WETH tokens are in the pool's account. However, *addLiquidity* will call *grabToken* which will always transfer the tokens from the *msg.sender* account. If *msg.sender* does not have sufficient WETH tokens or has not approved the pool contract to transfer WETH tokens, then the call will always fail. Otherwise, *msg.sender* will lose assets as they are paying double.

```
function grabToken(address token, uint amount) private {
    uint userBefore = IERC20(token).balanceOf(msg.sender);
    uint thisBefore = IERC20(token).balanceOf(address(this));

    require(IERC20(token).transferFrom(msg.sender, address(this), amount), "grabToken
failed");

    uint userAfter = IERC20(token).balanceOf(msg.sender);
    uint thisAfter = IERC20(token).balanceOf(address(this));

    require(userAfter.add(amount) == userBefore);
    require(thisAfter == thisBefore.add(amount));
}

function addETHLiquidityWithLimit(uint amount, uint minAmount0, uint minAmount1,
address user) public payable nonReentrant returns (uint real0, uint real1, uint
amountLP) {
    require(token0 == WETH);
    IWETH(WETH).deposit.value(msg.value)();
    (real0, real1, amountLP) = addLiquidity(msg.value, amount, user);
    if (real0 != msg.value) {
        IWETH(WETH).withdraw(msg.value.sub(real0));
        (bool success, ) = user.call.value(msg.value.sub(real0))("");
        require(success, "Transfer failed.");
    }
    require(real0 >= minAmount0, "minAmount0 is not satisfied");
```

```
    require(real1 >= minAmount1, "minAmount1 is not satisfied");
}
```

Severity is medium as it could theoretically result in lost funds, but it cannot be used by an attacker and usually the code will simply fail.

Fix

Remove *addETHLiquidityWithLimit* function (commit fde686a8). MESHswapRouter.impl.sol already has a helper function, *addLiquidityETH*, that allows a caller to add liquidity using a native coin.

Issue #8: addETHLiquidityWithLimit refunds to user instead of msg.sender

Summary	Severity
In Exchange.impl.sol, *addETHLiquidityWithLimit* refunds extra coins to the destination user instead of *msg.sender*. While this behavior may be intentional, it is undocumented and unintuitive. This also applies to functions in MESHswapRouter.impl.sol: *addLiquidity* and *addLiquidityETH*.	Low

The functions to add liquidity to a pool will refund extra tokens. This is different from Uniswap where those extra tokens would be lost. However, it is not documented where the tokens should be refunded. One option is to refund the tokens to *msg.sender* as they are taken from *msg.sender*. The other option is to refund them to *to* which is the address that will receive the minted liquidity pool tokens.

We do not assert that either option is the correct option, but it should be documented so that callers know the behavior of these functions. Callers must not make any assumptions about the undocumented behavior of functions.

```
function addLiquidity(
    address token0, address token1, uint amountADesired, uint amountBDesired,
    uint amountAMin, uint amountBMin, address to, uint deadline
) external ensure(deadline) nonReentrant returns (uint amount0, uint amount1, uint
liquidity) {
    address pair = IFactory(factory).tokenToPool(token0, token1);
    TransferHelper.safeTransferFrom(
       token0, msg.sender, address(this), amountADesired);
    TransferHelper.safeTransferFrom(
       token1, msg.sender, address(this), amountBDesired);
    (amount0, amount1, liquidity) = IExchange(pair).addTokenLiquidityWithLimit(
       amountADesired, amountBDesired, amountAMin, amountBMin, to);
    if (amount0 < amountADesired)
       TransferHelper.safeTransfer(token0, to, amountADesired.sub(amount0));
    if (amount1 < amountBDesired)
       TransferHelper.safeTransfer(token1, to, amountBDesired.sub(amount1));
}
```

Severity is low as this is a documentation and specification problem. Callers should not rely on any behavior that is not documented.

Fix

The fix for issue #7 removed *addETHLiquidityWithLimit* (commit fde686a8). Refund *msg.sender* instead of *to* in *addLiquidity* and *addLiquidityETH* (commit 07e65bc5).

We found during the preparation of the report that the fix was incomplete as *addLiquidityETH* will now refund ETH to *to* and the other token to *msg.sender*. We reported this issue to MESHswap.

Issue #9: Possible race condition during pool initialization

Summary	Severity
In Factory.impl.sol, *createPool* calls the tokens' *transferFrom* functions before calling *initPool*. If a token implements EIP-777, an attacker could call into the partially initialized pool which may result in unintended behaviors.	Low

Some tokens implement EIP-777 which provides for hooks when transferring tokens. These tokens have been a source of reentrancy issues in several attacks. One potential issue we identified is during the creation of a pool. An attacker could create a pool for a EIP-777 token and call into the pool during the hooks, before the call to *initPool*. As the pool state is uninitialized before the call to *initPool*, this could result in unintended behavior. It is unknown whether this could result in any future profit for the attacker.

To eliminate the possible race condition, there should be no calls to external functions before the call to *initPool*. Or all of the functions in the pool contract should check that the state has been initialized.

```
function createPool(address token0, uint amount0, address token1, uint amount1, uint
fee, bool isETH) private {
    // ...

    Exchange exc = new Exchange(token0, token1, fee);

    // ...

    userBefore = IERC20(token1).balanceOf(msg.sender);
    require(IERC20(token1).transferFrom(msg.sender, address(exc), amount1));
    userAfter = IERC20(token1).balanceOf(msg.sender);
    require(userAfter == userBefore - amount1);
    require(IERC20(token1).balanceOf(address(exc)) == amount1);

    poolExist[address(exc)] = true;
    IExchange(address(exc)).initPool(msg.sender);
    pools.push(address(exc));

    tokenToPool[token0][token1] = address(exc);
```

```
    tokenToPool[token1][token0] = address(exc);

    IRouter(router).approvePair(address(exc), token0, token1);

    emit CreatePool(token0, amount0, token1, amount1, fee, address(exc), pools.length
- 1);
}
```

Severity is low as it cannot result in any immediate profit for an attacker, and it is not clear if there is any benefit to exploiting this issue.

Fix

Refactor *createPool* and *initPool* (commit deca7594). Moves *transferFrom* calls to after call to *initPool*, eliminating possible race condition.

Issue #10: sendReward should ensure tokenomics are not violated

Summary	Severity
In MESH.sol, *sendReward* insufficiently limits the amount of newly minted tokens. This may result in a violation of the tokenomics if there is a misconfiguration or bug in an approved caller.	Low

The logic to calculate the amount of MESH reward tokens is spread across several contracts. There is no single place that checks that the amount of reward tokens being minted is correct, so all of the contracts that are allowed to call *sendReward* must be trusted. As a mitigation against possible issues in these trusted contracts, we suggest that *sendReward* ensures there is a limit to the number of tokens that can be minted. Specifically, the number of tokens minted by *sendReward*, in total, should never be larger than the amount *mined*. This prevents an attacker from minting an infinite supply of MESH tokens.

```
function sendReward(address user, uint amount) public {
    require(msg.sender == owner ||
IFactory(IGovernance(owner).factory()).poolExist(msg.sender));
    require(amount <= mined());

    balanceOf[user] = balanceOf[user].add(amount);
    totalSupply = totalSupply.add(amount);

    emit Transfer(address(0), user, amount);
}
```

Severity is low as there is no exploitable issue provided that the allowed callers are properly configured and implemented.

Fix

Track amount of MESH mined from *sendReward* and ensure that the total remains below *mined* (commit a52019e8).

```
function sendReward(address user, uint amount) public {
    require(msg.sender == owner ||
IFactory(IGovernance(owner).factory()).poolExist(msg.sender));
    require(amount.add(rewarded) <= mined());

    rewarded = rewarded.add(amount);
    balanceOf[user] = balanceOf[user].add(amount);
    totalSupply = totalSupply.add(amount);

    emit Transfer(address(0), user, amount);
}
```

Issue #11: Typo in getAmountIn results in incorrect behavior

Summary	Severity
In MESHswapRouter.impl.sol, the *getAmountIn* view function is incorrect because it calls the *getAmountOut* library function.	Low

getAmountIn has a simple typo that produces incorrect results as it calls the *getAmountOut* library function instead of the *getAmountIn* library function.

```
function getAmountIn(uint amountOut, uint reserveIn, uint reserveOut) public pure
returns (uint amountIn) {
    return MESHswapLibrary.getAmountOut(amountOut, reserveIn, reserveOut);
}
```

Severity is low as it is more likely to result in failure than an exploitable condition. Additionally, callers should avoid using this function entirely as it only supports pools whose fee is exactly 0.3%.

Fix

Replace with call to *MESHswapLibrary.getAmountIn* (commit 179ae46d).

Issue #12: liquidate should have nonReentrant modifier

Summary	Severity
In PlusPool.impl.sol, liquidate is lacking the *nonReentrant* modifier. While liquidators are currently whitelisted, *nonReentrant* modifier should be added in an abundance of caution.	Low

liquidate is a complex function and the only one in PlusPool.impl.sol that does not have the *nonReentrant* modifier. While callers are currently trusted, adding *nonReentrant* will not break any functionality and it may prevent issues in the future when anyone can call *liquidate*.

```
function liquidate(address user) public onlyLiquidator {
    // ...
}
```

Severity is low as this function already has a whitelist of valid callers.

Fix

Add *nonReentrant* modifier to *liquidate* (commit 0af2ca79).

Issue #13: BuybackFund may be vulnerable to sandwich attacks

Summary	Severity
In BuybackFund.impl.sol, the token swaps specify a minimum output of 1 token. A sandwich attack could result in a loss of funds in the buyback pool.	Low

Sandwich opportunities are actively exploited on Polygon where it results in a profit. For example, in block 27054844, a bot successfully executed a sandwich attack that resulted in a profit of ~0.2 MATIC:

```
0x1c1c095a6ff09c62bbb6e63a05451218486a6d2889261afe59ec5eb5dc340859
155 Gwei
Swap 21.326 MATIC for 111.023 PAPER

0x5799c32e3c5d07199c021648d7ede526cbf5c5eb43087a51a2ed4597c1bacb03
55 Gwei (victim transaction)
Swap 100 USDT for 70.234 MATIC, then swap for 361.883 PAPER

0xc287ce842cf8a5d77d51fa9baafaaacbcfdd859f28743ebdb78e411fbbf32cfc
55 Gwei
Swap 111.023 PAPER for 21.536 MATIC
```

In order to prevent exploitation, swaps must provide a reasonable minimum output amount, which can be provided by the transaction sender or a price oracle. In BuybackFund.impl.sol, for example, there is a vulnerable swap call where a fixed output amount of 1 token is provided:

```
function exchange(address token, uint256 amount) private returns (uint256) {
    if(token == mesh || amount == 0) return amount;

    uint256 bal = token == address(0) ? (address(this)).balance :
IERC20(token).balanceOf(address(this));
    if(bal < amount) amount = bal;

    IERC20 mesh = IERC20(mesh);
    uint256 diff = mesh.balanceOf(address(this));
    IRouter(router).swapExactTokensForTokens(amount, 1, paths[token], address(this),
block.timestamp + 3600);

    diff = (mesh.balanceOf(address(this))).sub(diff);
```

```
    return diff;
}
```

Severity is low as an attack is not risk free for the attacker and relies on a large swap relative to the liquidity of the pools. It is unknown whether a sandwich attack would be profitable in this example.

Fix

Transaction sender provides minimum output amounts that are passed to the swap function (commit 3c9dce1b).

```
function buybackRange(uint256 si, uint256 ei, uint256[] memory minAmount0, uint256[] memory minAmount1) public nonReentrant onlyOperator {
    ...
}

function exchange(address token, uint256 amount, uint256 minAmount) private returns (uint256) {
    if(token == mesh || amount == 0) return amount;

    uint256 bal = token == address(0) ? (address(this)).balance : IERC20(token).balanceOf(address(this));
    if(bal < amount) amount = bal;

    IERC20 mesh = IERC20(mesh);
    uint256 diff = mesh.balanceOf(address(this));
    IRouter(router).swapExactTokensForTokens(amount, minAmount, paths[token], address(this), block.timestamp + 3600);

    diff = (mesh.balanceOf(address(this))).sub(diff);

    return diff;
}
```

Recommendations

These are suggestions to improve code maintainability, readability, and/or resilience.

Summary

#	Title
1	Use *block.timestamp* instead of *block.number*
2	Fix various comments, typos, and variable names
3	Check caller is EOA in *setMiningRate*
4	Remove duplicate logics in several contracts

Recommendation #1: Use block.timestamp instead of block.number

The time to produce a block on Polygon could potentially change in the future. This would result in the distribution of MESH reward tokens also changing as it depends on *block.number*. This issue is described in the Consensys's "Development Recommendations" (https://consensys.github.io/smart-contract-best-practices/development-recommendations/solidity-specific/timestamp-dependence/). According to Consensys, as long as manipulating the timestamp within a 15 second window does not produce much profit, then it is acceptable to use *block.timestamp*.

Polygon provides for even stricter limits on how much *block.timestamp* can deviate from the consensus view of real time, i.e., a few seconds, and with these limits it would be safe to rely on *block.timestamp* for the purposes of distributing rewards.

Fix

MESHswap decided that they will continue to use *block.number*.

Recommendation #2: Fix various comments, typos, and variable names

During the source code review, we found some outdated or incorrect comments and variable names as well as typos. These should be fixed. We also recommend adding additional documentation as most contracts and functions are not currently documented.

Fix

MESHswap fixed the trivial issues we identified (commit 179ae46d and 0af2ca79).

Recommendation #3: Check caller is EOA in setMiningRate

In Governance.impl.sol, *setMiningRate* is responsible for incrementing the current epoch which affects the rewards for liquidity providers. This function can be called by anyone, but it will only have an effect if it is time for the next epoch. Our concern is that by calling *setMiningRate* from an unexpected context it may be possible for an attacker to manipulate the distribution of rewards. We do not have any scenarios where this is profitable for an attacker, but we recommend that the risk is mitigated.

One possible mitigation is to whitelist allowed callers, but this is overkill without proof of an issue. Instead, we recommend that *setMiningRate* require *tx.origin == msg.sender*, which prevents it from being called in an unexpected context. The downsides of this change would be minimal.

Fix

MESHswap applied the recommendation (commit 0af2ca79).

Recommendation #4: Remove duplicate logics in several contracts

Duplicate code and logic are a challenge for maintainability and readability. Whenever a fix is made to one instance of the duplicated code, it must also be applied to every other instance of the duplicated code. Otherwise, the fix will be incomplete. Additionally, duplicate code makes source code review more difficult as reviewers must check the correctness of each instance of the duplicated code. As such, we recommend that duplicated code and logic is eliminated, when possible, provided it does not significantly harm the understandability of the code.

We found several examples of duplicated logic:

- In Exchange.impl.sol, *getMiningIndex*, *updateMiningIndex* and *giveReward*
- In Exchange.impl.sol, *exchangePos* and *exchangeNeg*
- In SinglePool.impl.sol, *getMiningIndex* and *updateMiningIndex*

Fix

MESHswap eliminated *getMiningIndex* and instead call *updateMiningIndex* (commit 31eba42c). MESHswap decided to keep *exchangePos* and *exchangeNeg* as-is.