Lucidly Security Review

Pashov Audit Group

Conducted by: Said, ast3ros, btk, ZanyBonzy

June 24th 2024 - July 4th 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 **lucidlyfi/lucidly-core-v1** repository was done by **Pashov Audit Group**, with a focus on the security aspects of the application's smart contracts implementation.

4. About Lucidly

Lucidly allows the creation of liquidity pools consisting of various tokens. This pool also works as an AMM for these tokens allowing exchanging between them.

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 - 2be4150ea1eecded436f3b5ee37c1cc57ff04d62

fixes review commit hash - 5bbf1fc080e47cec886240aa00ae7c4f93794a88

Scope

The following smart contracts were in scope of the audit:

- `Pool`
- `PoolToken`
- `Staking`

7. Executive Summary

Over the course of the security review, Said, ast3ros, btk, ZanyBonzy engaged with Lucidly to review Lucidly. In this period of time a total of **18** issues were uncovered.

Protocol Summary

Protocol Name	Lucidly
Repository	https://github.com/lucidlyfi/lucidly-core-v1
Date	June 24th 2024 - July 4th 2024
Protocol Type	multi-asset AMM

Findings Count

Severity	Amount
Critical	1
High	2
Medium	7
Low	8
Total Findings	**18**

Summary of Findings

ID	Title	Severity	Status
[C-01]	Fees will always be sent to address(0)	Critical	Resolved
[H-01]	Wrong _prevRatios index could be used	High	Resolved
[H-02]	The wrong amplification value is used to calculate the new supply	High	Resolved
[M-01]	Staking vault is susceptible to initial grief attack	Medium	Acknowledged
[M-02]	Owner can rescue() pool tokens in certain cases	Medium	Acknowledged
[M-03]	Staking contract is not EIP-4626 compliant	Medium	Resolved
[M-04]	Not using SafeTransferLib	Medium	Resolved
[M-05]	Unintended token minting to staking address when fee rate is 0	Medium	Acknowledged
[M-06]	Rate and weight not updated during liquidity removal	Medium	Acknowledged
[M-07]	Certain stablecoins cannot be added to Pool	Medium	Acknowledged
[L-01]	Wrong loop area in removeLiquidity.	Low	Resolved
[L-02]	_nameHash is not initialized upon deployment.	Low	Resolved
[L-03]	Risks due to centralization	Low	Resolved
[L-04]	Fees from the first set of liquidity providers can be lost	Low	Acknowledged

[L-05]	Missing sanity checks when setting tokenAddress_	Low	Resolved
[L-06]	Users can bypass fees by depositing in chunks	Low	Acknowledged
[L-07]	No deadline parameters when executing swaps	Low	Acknowledged
[L-08]	Staker can sandwich large Pool operations to get immediate profit	Low	Acknowledged

8. Findings

8.1. Critical Findings

[C-01] Fees will always be sent to `address(0)`

Severity

Impact: High

Likelihood: High

Description

The Staking contract includes a fee mechanism implemented as follows:

```
uint256 _fee = (shares_ * depositFeeBps) / 1e4;
        _mint(to_, shares_ - _fee);
        _mint(protocolFeeAddress, _fee);
```

All fees are sent to protocolFeeAddress. However, the contract does not initialize this address, nor does it provide a function to update it. As a result, fees are permanently lost because the default value of `protocolFeeAddress` is address(0).

Recommendations

Consider adding the following function to the Staking contract:

```
function setProtocolFeeAddress
      (address protocolFeeAddress_) external onlyOwner {
        if (protocolFeeAddress_ == address(0)) revert Staking__InvalidParams();
        protocolFeeAddress = protocolFeeAddress_;
    }
```

8.2. High Findings

[H-01] Wrong `_prevRatios` index could be used

Severity

Impact: Medium

Likelihood: High

Description

When `addLiquidity` is called and initial liquidity has already been provided, the caller can skip providing one of the listed tokens by setting the corresponding value in `amounts_` to 0. If the amount provided is 0, the function will skip updating the token and pool virtual balance product and sum, as well as skip storing `_prevRatios`.

```
function addLiquidity
      (uint256[] calldata amounts_, uint256 minLpAmount_, address receiver_)
        external
        nonReentrant
        returns (uint256)
    {
        uint256 _numTokens = numTokens;
        if (amounts_.length != _numTokens) revert Pool__InvalidParams();

        // ...

        // update rates
        (_virtualBalanceProd, _virtualBalanceSum) = _updateRates
          (_tokens, _virtualBalanceProd, _virtualBalanceSum);
        uint256 _prevSupply = supply;

        uint256 _virtualBalanceProdFinal = _virtualBalanceProd;
        uint256 _virtualBalanceSumFinal = _virtualBalanceSum;
        uint256 _prevVirtualBalanceSum = _virtualBalanceSum;
        uint256[] memory _prevRatios = new uint256[](_numTokens);
        uint256 _virtualBalance;

        for (uint256 t = 0; t < MAX_NUM_TOKENS; t++) {
            if (t == _numTokens) break;

            uint256 __amount = amounts_[t];

            if (__amount == 0) {
                if (!(_prevSupply > 0)) {
                    revert Pool__InitialDepositAmountMustBeNonZero();
                }
                continue;
            }

            // update stored virtual balance
            (_prevVirtualBalance, _rate, _packedWeight) = _unpackVirtualBalance
              (packedVirtualBalances[t]);
            uint256 _changeInVirtualBalance = (__amount * _rate) / PRECISION;
            _virtualBalance = _prevVirtualBalance + _changeInVirtualBalance;
            packedVirtualBalances[t] = _packVirtualBalance
              (_virtualBalance, _rate, _packedWeight);

            if (_prevSupply > 0) {
>>>             _prevRatios[t] =
  (_prevVirtualBalance * PRECISION) / _prevVirtualBalanceSum;
                uint256 _weightTimesN = _unpackWeightTimesN
                  (_packedWeight, _numTokens);

                // update product and sum of virtual balances
                _virtualBalanceProdFinal = (
                    _virtualBalanceProdFinal
                        * _powUp((


                        ) / _virtualBalance, _weightTimesN
                ) / PRECISION;

                // the `D^n` factor will be updated in `_calculateSupply()`
                _virtualBalanceSumFinal += _changeInVirtualBalance;

                // remove fees from balance and recalculate sum and product
                uint256 _fee = (
                    (_changeInVirtualBalance -
                      (_prevVirtualBalance * _lowest) / PRECISION) * (swapFeeRate / 2)
                ) / PRECISION;
                _virtualBalanceProd = (
                    _virtualBalanceProd
```

```
                            * _powUp((_prevVirtualBalance * PRECISION) /
                              (_virtualBalance - _fee), _weightTimesN)
                    ) / PRECISION;
                    _virtualBalanceSum += _changeInVirtualBalance - _fee;
                }

                SafeTransferLib.safeTransferFrom(tokens[t], msg.sender, address
                  (this), __amount);
            }
            // ...
}
```

However, when checking if ratios change within a valid band, it will provide `prevRatios` at the wrong index.

```
// ...
        uint256 _supply = _prevSupply;
        if (_prevSupply == 0) {
            // initial deposit, calculate necessary variables
            (
              _virtualBalanceProd,
              _virtualBalanceSum
            ) = _calculateVirtualBalanceProdSum(
            if (!(_virtualBalanceProd > 0)) revert Pool__AmountsMustBeNonZero();
            _supply = _virtualBalanceSum;
        } else {
            // check bands
            uint256 _j = 0;
            for (uint256 t = 0; t < MAX_NUM_TOKENS; t++) {
                if (t == _numTokens) break;
                if (amounts_[t] == 0) continue;
                (_virtualBalance, _rate, _packedWeight) = _unpackVirtualBalance
                  (packedVirtualBalances[t]);
>>>             _checkBands(_prevRatios[_j],
  (_virtualBalance * PRECISION) / _virtualBalanceSumFinal, _packedWeight);
                _j = FixedPointMathLib.rawAdd(_j, 1);
            }
        }
// ...
```

Consider a scenario where there are 3 tokens (token index 0, index 1, and index 2).

when `addLiquidity` is called, skipping index 1 token, with `amounts_` at index 1 set to 0.

When `_checkBands` is called for the token at index 2, it will provide `_prevRatios` at index `_j` equal to 1 (since token index 1 skipped at not incrementing `_j`). This is incorrect because when storing `_prevRatios`, the actual index (`t`) is used regardless of whether there is a skipped token or not.

This could cause the valid `addLiquidity` operation to revert due to using `_prevRatios` at the wrong index.

Recommendations

Use `t` index instead of `_j` :

```
// ...
        uint256 _supply = _prevSupply;
        if (_prevSupply == 0) {
            // initial deposit, calculate necessary variables
            (
              _virtualBalanceProd,
              _virtualBalanceSum
            ) = _calculateVirtualBalanceProdSum(
            if (!(_virtualBalanceProd > 0)) revert Pool__AmountsMustBeNonZero();
            _supply = _virtualBalanceSum;
        } else {
            // check bands
-            uint256 _j = 0;
            for (uint256 t = 0; t < MAX_NUM_TOKENS; t++) {
                if (t == _numTokens) break;
                if (amounts_[t] == 0) continue;
                (_virtualBalance, _rate, _packedWeight) = _unpackVirtualBalance
                  (packedVirtualBalances[t]);
-                _checkBands(_prevRatios[_j],
- (_virtualBalance * PRECISION) / _virtualBalanceSumFinal, _packedWeight);
+                _checkBands(_prevRatios[t],
+ (_virtualBalance * PRECISION) / _virtualBalanceSumFinal, _packedWeight);
-                _j = FixedPointMathLib.rawAdd(_j, 1);
            }
        }
// ...
```

[H-02] The wrong amplification value is used to calculate the new supply

Severity

Impact: High

Likelihood: Medium

Description

When `addToken` is called to add a new token to the pool, one of the parameters that are provided are the new pool amplification factor for calculating supply and virtual balance prod. However, when `_calculateSupply` is called, it provides the previous `amplification` instead of the new `amplification_` value.

```
function addToken(
        address token_,
        address rateProvider_,
        uint256 weight_,
        uint256 lower_,
        uint256 upper_,
        uint256 amount_,
        uint256 amplification_,
        uint256 minLpAmount_,
        address receiver_
    ) external onlyOwner {
        // ...

        _virtualBalance = (amount_ * _rate) / PRECISION;
        _packedWeight = _packWeight(weight_, weight_, _lower, _upper);

        // set parameters for new token
        numTokens = _numTokens;
        tokens[_prevNumTokens] = token_;
        rateProviders[_prevNumTokens] = rateProvider_;
        packedVirtualBalances[_prevNumTokens] = _packVirtualBalance
          (_virtualBalance, _rate, _packedWeight);

        // recalculate variables
        (
          uint256_virtualBalanceProd,
          uint256_virtualBalanceSum
        ) = _calculateVirtualBalanceProdSum(

        // update supply
        uint256 _prevSupply = supply;
        uint256 __supply;
        // @audit - should provide new amplification here
        (__supply, _virtualBalanceProd) = _calculateSupply(
>>>
           _numTokens, _virtualBalanceSum, amplification, _virtualBalanceProd, _virtua
        );

        amplification = amplification_;
        supply = __supply;
        packedPoolVirtualBalance = _packPoolVirtualBalance
          (_virtualBalanceProd, _virtualBalanceSum);

        SafeTransferLib.safeTransferFrom(token_, msg.sender, address
          (this), amount_);
        if (__supply <= _prevSupply) revert Pool__InvalidParams();
        uint256 _lpAmount = FixedPointMathLib.rawSub(__supply, _prevSupply);
        if (_lpAmount < minLpAmount_) revert Pool__InvalidParams();
        PoolToken(tokenAddress).mint(receiver_, _lpAmount);
        emit AddToken
          (_prevNumTokens, token_, rateProvider_, _rate, weight_, amount_);
    }
```

This will cause the calculation of the supply and the virtual balance product of the pool to be incorrect, and the provided amplification will not immediately affect the supply and virtual balance product as expected.

Recommendations

Provide `amplification_` instead of `amplification` to `_calculateSupply`.

```
// ...
        (__supply, _virtualBalanceProd) = _calculateSupply(
-
-            _numTokens, _virtualBalanceSum, amplification, _virtualBalanceProd, _virt
+
+            _numTokens, _virtualBalanceSum, amplification_, _virtualBalanceProd, _vir
        );
// ...
```

8.3. Medium Findings

[M-01] `Staking` vault is susceptible to initial grief attack

Severity

Impact: High

Likelihood: Low

Description

The initial deposit attack is largely mitigated due to the usage of virtual shares. However, the initial depositor grief attack is still possible, causing the first depositor to lose assets at a relatively low cost for the griefer.

```
contract PoolSwap is Test {

    Staking staking;
    MockToken public token;
    address alice = makeAddr("alice");
    address bob = makeAddr("bob");

    function setUp() public {
        token = new MockToken("token", "t", 18);

        // deploy staking contract
        staking = new Staking(address
          (token), "XYZ Mastervault Token", "XYZ-MVT", true, alice);
    }

        function test_initial_deposit_grief() public {

        vm.startPrank(alice);
        token.mint(alice,11e18 + 10);

        uint256 initialAssetBalance = token.balanceOf(alice);
        console.log("attacker balance before : ");
        console.log(initialAssetBalance);

        token.approve(address(staking), 1e18);

        staking.mint(10, alice);

        token.transfer(address(staking), 11e18);

        vm.stopPrank();

        vm.startPrank(bob);

        token.mint(bob,10e18 + 10);
        token.approve(address(staking), 1e18);
        staking.deposit(1e18, bob);
        vm.stopPrank();

        uint256 bobShares = staking.balanceOf(bob);
        console.log("bob shares : ");
        console.log(bobShares);

        vm.stopPrank();

        vm.startPrank(alice);

        staking.redeem(staking.balanceOf(alice), alice, alice);
        uint256 afterAssetBalance = token.balanceOf(alice);
        console.log("attacker balance after : ");
        console.log(afterAssetBalance);
        vm.stopPrank();

    }
}
```

Run the test :

```
forge test --match-test test_initial_deposit_grief -vvv
```

Log output :

```
Logs:
  attacker balance before :
  1100000000000000010
  bob shares :
  0
  attacker balance after :
  10909090909090909100
```

It can be observed that alice can lock `1 ETH` of bob's asset at the cost of `~ 0.1 ETH`.

Recommendations

Consider mitigating this with an initial deposit of a small amount

[M-02] Owner can `rescue()` pool tokens in certain cases

Severity

Impact: High

Likelihood: Low

Description

The pool contract includes a `rescue()` function designed to recover non-pool tokens from the contract:

```
function rescue(address token_, address receiver_) external onlyOwner {
        uint256 _numTokens = numTokens;
        for (uint256 t = 0; t < MAX_NUM_TOKENS; t++) {
            if (t == _numTokens) break;
            if (!(token_ != tokens[t])) revert Pool__CannotRescuePoolToken();
        }
        uint256 _amount = ERC20(token_).balanceOf(address(this));
        ERC20(token_).transfer(receiver_, _amount);
    }
```

When the owner attempts to rescue a pool token, the function throws a `Pool__CannotRescuePoolToken()` error. However, the owner can bypass this check if a token has multiple entry points. For example, this was the case with the TUSD token when its secondary entry point was still active. This vulnerability remains a risk as other tokens with multiple entry points may exist.

Recommendations

While there is no direct fix for this issue, it is important to document that the owner can withdraw all tokens in cases where tokens have multiple entry points.

[M-03] Staking contract is not EIP-4626 compliant

Severity

Impact: Medium

Likelihood: Medium

Description

The current staking implementation charges fees on assets deposited into the vault, as shown below:

```
function _deposit(
        addressby_,
        addressto_,
        uint256tokens_,
        uint256shares_
    ) internal virtual override {
        SafeTransferLib.safeTransferFrom(asset(), by_, address(this), tokens_);
        uint256 _fee = (shares_ * depositFeeBps) / 1e4;
        _mint(to_, shares_ - _fee);
        _mint(protocolFeeAddress, _fee);

        emit Deposit(by_, to_, tokens_, shares_);

        _afterDeposit(tokens_, shares_);
    }
```

According to EIP4626, `previewDeposit()` and `previewMint()` must be inclusive of deposit fees:

> MUST be inclusive of deposit fees. Integrators should be aware of the existence of deposit fees.

However, the current staking implementation (i.e. OpenZeppelin impl) does not account for these fees when converting from assets to shares. This omission can lead to unexpected behavior and integration issues in the future.

Recommendations

Consider overriding the `previewDeposit()` and `previewMint()` functions to include fees.

[M-04] Not using `SafeTransferLib`

Severity

Impact: Medium

Likelihood: Medium

Description

Most of the token transfer operations within the `Pool` already utilize `SafeTransferLib`. However, several operations, including those in `removeLiquidity`, `rescue`, and `skim`, still use the unsafe transfer method. Consider adjusting these transfer operations to also utilize `SafeTransferLib`.

The rescue function is designed to withdraw all tokens mistakenly sent to the contract, excluding the supported tokens listed in the tokens array. However, the function fails for non-standard ERC20 tokens that do not return a boolean value on transfer (e.g., USDT). This causes the contract to revert, preventing the rescue of those tokens.

```
function rescue(address token_, address receiver_) external onlyOwner {
        uint256 _numTokens = numTokens;
        for (uint256 t = 0; t < MAX_NUM_TOKENS; t++) {
            if (t == _numTokens) break;
            if (!(token_ != tokens[t])) revert Pool__CannotRescuePoolToken();
        }
        uint256 _amount = ERC20(token_).balanceOf(address(this));
        ERC20(token_).transfer(receiver_, _amount);
    }
```

In addition, the current implementation of `Pool.removeLiquidity()` does not support tokens that do not return a boolean on successful transfers. The issue lies in the following code:

```
if (!(ERC20(tokens[t]).transfer
  (receiver_, amount))) revert Pool__TransferFailed();
```

Recommendations

Consider using `SafeTransferLib.safeTransfer()` to handle such tokens.

[M-05] Unintended token minting to staking address when fee rate is 0

Severity

Impact: Medium

Likelihood: Medium

Description

When adding liquidity to the pool, the contract calls `_calculateSupply` twice. The difference between the two results is minted to the staking address as mint fees. However, when the fee rate is set to 0, pool tokens are still minted to the staking address, resulting in an unexpected loss for liquidity providers.

This issue arises because the first call to `_calculateSupply` rounds down, while the second call rounds up. Consequently, each time the `addLiquidity` function is invoked with `_prevSupply > 0` and `mint fee = 0`, the staking address receives a small number of pool tokens.

```
function addLiquidity
      (uint256[] calldata amounts_, uint256 minLpAmount_, address receiver_)
        external
        nonReentrant
        returns (uint256)
    {
        ...
        // mint LP tokens
        (_supply, _virtualBalanceProd) = _calculateSupply(
            _numTokens, _supply, amplification, _virtualBalanceProd,
            // _virtualBalanceSum, _prevSupply == 0 // @audit first call - rounding do
        );
        uint256 _toMint = _supply - _prevSupply;

        if (!(_toMint > 0 && _toMint >= minLpAmount_)) {
            revert Pool__SlippageLimitExceeded();
        }
        PoolToken(tokenAddress).mint(receiver_, _toMint);
        emit AddLiquidity(msg.sender, receiver_, amounts_, _toMint);

        uint256 _supplyFinal = _supply;
        if (_prevSupply > 0) {
            // mint fees
            (_supplyFinal, _virtualBalanceProdFinal) = _calculateSupply(
                _numTokens, _prevSupply, amplification,
                // _virtualBalanceProdFinal, _virtualBalanceSumFinal, true  // @audit
            );
            PoolToken(tokenAddress).mint
              (stakingAddress, _supplyFinal - _supply);
        ...
```

In the `_calculateSupply` function, the first calculation rounds down, yielding `_sp - delta`. The second calculation for mint fees rounds up, resulting in `_sp + delta`. The difference between the two calculations, which is `2 * delta`, is minted to the staking address as mint fees, even when the fee rate is 0.

```
function _calculateSupply(
        uint256 numTokens_,
        uint256 supply_,
        uint256 amplification_,
        uint256 virtualBalanceProd_,
        uint256 virtualBalanceSum_,
        bool up_
    ) internal pure returns (uint256, uint256) {
        ...
            if (FixedPointMathLib.rawDiv(FixedPointMathLib.rawMul
              (_delta, PRECISION), _s) <= MAX_POW_REL_ERR) {
                _delta = FixedPointMathLib.rawDiv(FixedPointMathLib.rawMul
                  (_sp, MAX_POW_REL_ERR), PRECISION);
                if (up_) {
                    _sp += _delta;
                } else {
                    _sp -= _delta;
                }
                return (_sp, _r);
            }

            _s = _sp;
        }
        ...
    }
```

Liquidity providers experience a minor but continuous loss of value, as a portion of their provided liquidity is unintentionally transferred to the staking address. Over time and with increased transaction volume, this could lead to a significant accumulation of tokens in the staking address, reducing the overall value for liquidity providers.

Recommendations

Check if the `swapFeeRate` is 0, and skip the calculation and minting of mint fees.

[M-06] Rate and weight not updated during liquidity removal

Severity

Impact: High

Likelihood: Low

Description

When users remove liquidity, the amount of tokens they receive will be calculated based on each token's `_rate`. Additionally, `virtualBalanceProd` will be recalculated using each token's weight.

```
function removeLiquidity
      (uint256 lpAmount_, uint256[] calldata minAmounts_, address receiver_)
        external
        nonReentrant
    {
        uint256 _numTokens = numTokens;

        if
          (minAmounts_.length != _numTokens || minAmounts_.length > MAX_NUM_TOKENS) re
        // update supply
        uint256 _prevSupply = supply;
        uint256 _supply = _prevSupply - lpAmount_;
        supply = _supply;
        PoolToken(tokenAddress).burn(msg.sender, lpAmount_);
        emit RemoveLiquidity(msg.sender, receiver_, lpAmount_);

        // update variables and transfer tokens
        uint256 _virtualBalanceProd = PRECISION;
        uint256 _virtualBalanceSum = 0;

        uint256 _prevVirtualBalance;
        uint256 _rate;
        uint256 _packedWeight;
        for (uint256 t = 0; t <= MAX_NUM_TOKENS; t++) {
            if (t == _numTokens) break;

            (_prevVirtualBalance, _rate, _packedWeight) = _unpackVirtualBalance
              (packedVirtualBalances[t]);

            uint256 __weight = _unpackWeightTimesN(_packedWeight, 1);

            uint256 dVb = (_prevVirtualBalance * lpAmount_) / _prevSupply;
            uint256 vb = _prevVirtualBalance - dVb;
            packedVirtualBalances[t] = _packVirtualBalance
              (vb, _rate, _packedWeight);
            // @audit - compare this calculation with other place
>>>         _virtualBalanceProd = FixedPointMathLib.rawDiv(
                FixedPointMathLib.rawMul(
                    _virtualBalanceProd,
                    _powDown(
                        FixedPointMathLib.rawDiv(FixedPointMathLib.rawMul
                          (_supply, __weight), vb),
                        FixedPointMathLib.rawMul(__weight, _numTokens)
                    )
                ),
                PRECISION
            );
            _virtualBalanceSum = FixedPointMathLib.rawAdd
              (_virtualBalanceSum, vb);

>>>         uint256 amount = (dVb * PRECISION) / _rate;
            if (amount < minAmounts_[t]) revert Pool__SlippageLimitExceeded();
            if (!(ERC20(tokens[t]).transfer
              (receiver_, amount))) revert Pool__TransferFailed();
        }

        packedPoolVirtualBalance = _packPoolVirtualBalance
          (_virtualBalanceProd, _virtualBalanceSum);
    }
```

However, it can be observed that weights and rates are not updated first. This can cause issues in scenarios where there is a sudden drop in the value of one of the tokens, requiring the rates to be readjusted. As a result, the tokens transferred to users may not reflect the current condition of the pool's rates and weights.

Recommendations

Consider updating rates and weights inside remove liquidity operation, before calculating the virtualBalanceProd.

[M-07] Certain stablecoins cannot be added to Pool

Severity

Impact: Medium

Likelihood: Medium

Description

The protocol aims to work with standard tokens, stablecoins, and LRTs, but in the constructor and the `addToken` function, there's a check ensuring that token decimals equal 18.

From constructor:

```
if (ERC20(tokens_[t]).decimals() != 18) {
                revert Pool__InvalidDecimals();
            }
```

From `addToken`:

```
if (ERC20(token_).decimals() != 18) revert Pool__InvalidParams();
```

Most popular stablecoins like USDC and USDT, do not have 18 decimals but 6, and as such cannot be added to the pool.

Recommendations

Recommend removing the check for decimals being 18.

8.4. Low Findings

[L-01] Wrong loop area in `removeLiquidity`.

The `removeLiquidity` function uses <= operator instead of <. Not much serious impact due to the function exiting the loop immediately t equals 32 but an extra iteration is attempted. Consider using the < operator instead.

```
for (uint256 t = 0; t <= MAX_NUM_TOKENS; t++) {
            if (t == _numTokens) break;
```

[L-02] `_nameHash` is not initialized upon deployment.

I believe this is worth pointing out as it depends on the solc that will be used to compile the contracts upon deployment.

The foundry config file uses 0.8.24, so here, it's not much of a problem, the permitted functionality will simply create a new `nameHash` if current `nameHash` is 0.

However, solc versions <= 0.8.20 will not allow deployment without initializations of all immutable state variables, and as the `_nameHash` is not initialized in the constructor, PoolToken.sol will not be compilable, consequently undeployable.

```
...........
    bytes32 internal immutable _nameHash;
...........
    constructor(
      stringmemoryname_,
      stringmemorysymbol_,
      uint8decimals_,
      addressowner_
    ) {
        _name = name_;
        _symbol = symbol_;
        _decimals = decimals_;
        _setOwner(owner_);
    }
```

I'd recommend just removing it since the permit function will create a `nameHash` anyway. If the protocol plans to use this instead, in the permit function, then it can be declared, and the `_constantNameHash` function created to return the it when queried.

[L-03] Risks due to centralization

A number of functions are protected by the `onlyOwner` modifier and are all vulnerable to the actions of a malicious or compromised admin. Some functions are however worth taking a look at and having protections in place to protect users from the potential ramifications.

1. `setDepositFeeBps` can be set very high to grief users, as high as just less than 100%. Introduce a more reasonable cap on the fee bps. Maybe 20%/25% cap.

```
function setDepositFeeBps(uint256 depositFeeBps_) external onlyOwner {
        if (depositFeeBps >= 1e4) revert Staking__InvalidParams();
        depositFeeBps = depositFeeBps_;
    }
```

2. `setPool` can be set to a zero address to dos poolToken minting/burning. Malicious owners can also set pool addresses to an externally controlled address to arbitrarily mint and burn tokens from anyone. A potential fix for this is to renounce ownership immediately after setting pool address and adding a zero address check.

```
function setPool(address poolAddress_) public onlyOwner {
        poolAddress = poolAddress_;

        emit PoolAddressSet(poolAddress);
    }
```

[L-04] Fees from the first set of liquidity providers can be lost

When users call the `addLiquidity` function, fees are minted to the `stakingAddress`. The staking address is not set in the constructor unlike other parameters, therefore within the period between deployment and admin calling the `setStaking` function, calls to the `addLiquidity` function risk minting fees to zero address.

```
function addLiquidity
      (uint256[] calldata amounts_, uint256 minLpAmount_, address receiver_)
        external
        nonReentrant
        returns (uint256)
    {
..........
        if (_prevSupply > 0) {
            // mint fees
            (_supplyFinal, _virtualBalanceProdFinal) = _calculateSupply(

                                    _numTokens, _prevSupply, amplification, _virtualBalan
            );
            PoolToken(tokenAddress).mint
              (stakingAddress, _supplyFinal - _supply);
..........
    }
```

The recommendation is to call the `setStaking` function in the constructor or skip minting if stakingAddress is 0.

[L-05] Missing sanity checks when setting `tokenAddress_`

The constructor of the Pool contract lacks sanity checks to ensure that the tokenAddress_ provided is valid. For reference, Yearn Finance includes such validation:

```
assert _token != empty(address)
```

To prevent errors during deployment, consider adding similar checks:

```
if (tokenAddress_ == address(0)) revert Pool__InvalidParams();
```

[L-06] Users can bypass fees by depositing in chunks

When users deposit tokens into the Staking contract, a fee is taken and minted to the `protocolFeeAddress`:

```
uint256 _fee = (shares_ * depositFeeBps) / 1e4;
        _mint(to_, shares_ - _fee);
        _mint(protocolFeeAddress, _fee);
```

Users can exploit this calculation by making small, fragmented deposits, causing the fee to round down to zero. This issue is particularly problematic for tokens with low decimal precision, such as GUSD, which has 2 decimals. Consider updating the deposit function as follows:

```
uint256 feeBps = depositFeeBps;
        require((shares_ * feeBps) > 1e4);
        uint256 _fee = (shares_ * feeBps) / 1e4;
        _mint(to_, shares_ - _fee);
        _mint(protocolFeeAddress, _fee);
    }
```

[L-07] No deadline parameters when executing swaps

Functions `addLiquidity`, `swap`, `removeLiquiditySingle`, and `removeLiquidity` are missing deadline check.

As front-running is a key aspect of AMM design, the deadline is a useful tool to ensure that users' transactions cannot be "saved for later" by miners or stay longer than needed in the mempool. The longer transactions stay in the mempool, the more likely it is that MEV bots can steal positive slippage from the transaction.

However, it's not a big problem, since the functions have `minAmountOut` parameter that offers slippage protection, so users are always going to get what they set. They would just miss out on more positive slippage if the exchange rate becomes favorable when the transaction is included in a block.

Consider introducing a `deadline` parameter in all the pointed-out functions.

[L-08] Staker can sandwich large `Pool` operations to get immediate profit

`Staking` is ERC4626 that tracks shares and uses the underlying token balance to determine deposit and withdrawal amounts. When `Pool` operations result in a fee that is minted to the `Staking` contract, it immediately impacts the calculation of deposit and withdrawal token amounts.

```
function swap(
        uint256 tokenIn_,
        uint256 tokenOut_,
        uint256 tokenInAmount_,
        uint256 minTokenOutAmount_,
        address receiver_
    ) external nonReentrant returns (uint256) {
        // ...

        // mint fees
        if (_tokenInFee > 0) {
            uint256 _supply;
>>>         (_supply, _virtualBalanceProd) = _updateSupply
  (supply, _virtualBalanceProd, _virtualBalanceSum);
        }

        packedPoolVirtualBalance = _packPoolVirtualBalance
          (_virtualBalanceProd, _virtualBalanceSum);

        // transfer tokens
        SafeTransferLib.safeTransferFrom(tokens[tokenIn_], msg.sender, address
          (this), tokenInAmount_);
        SafeTransferLib.safeTransfer
          (tokens[tokenOut_], receiver_, _tokenOutAmount);
        emit Swap(
          msg.sender,
          receiver_,
          tokenIn_,
          tokenOut_,
          tokenInAmount_,
          _tokenOutAmount
        );

        return _tokenOutAmount;
    }
```

If there is an operation that results in a considerable amount of tokens being sent to the `Staking` contract, such as `addLiquidity`, `swap`, or rate updates, an attacker can sandwich the operation by depositing tokens into the `Staking` contract before the operation, then immediately withdrawing to obtain the tokens. This is profitable as long as the amount withdrawn is greater than the staking fee.

Consider implementing a time-based reward rate distribution mechanism to prevent such scenarios.