Lucidly Security Review

Pashov Audit Group

Conducted by: carrotsmuggler, Dan Ogurtsov, ubermensch

April 25th 2024 - May 9th 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 - f00c45bbadc836b9eaa94717a0b3aa017e792588

fixes review commit hash - f36389e44cc99ac5a762722de3e98644f5407032

Scope

The following smart contracts were in scope of the audit:

- `Pool`
- `Staking`

7. Executive Summary

Over the course of the security review, carrotsmuggler, Dan Ogurtsov, ubermensch engaged with Lucidly to review Lucidly. In this period of time a total of **28** issues were uncovered.

Protocol Summary

Protocol Name	Lucidly
Repository	https://github.com/lucidlyfi/lucidly-core-v1
Date	April 25th 2024 - May 9th 2024
Protocol Type	multi-asset AMM

Findings Count

Severity	**Amount**
Critical	5
High	14
Medium	2
Low	7
Total Findings	**28**

Summary of Findings

ID	Title	Severity	Status
[C-01]	Incorrect variable used when calculating supply	Critical	Resolved
[C-02]	Wrong index used when updating rates	Critical	Resolved
[C-03]	Incorrect math operation leads to underprediction of stored product	Critical	Resolved
[C-04]	Missing division leads to massively deflated stored product	Critical	Resolved
[C-05]	Incorrect subtraction leads to draining of the pool	Critical	Resolved
[H-01]	Bad killed check in unpause and kill functions	High	Resolved
[H-02]	Missing negation in if clause	High	Resolved
[H-03]	Wrong out-of-bounds check in swap function	High	Resolved
[H-04]	amplification_ vs amplification in the pool constructor	High	Resolved
[H-05]	Iterating the loop length below MAXLENGTH	High	Resolved
[H-06]	Impossible to add/remove liquidity if numTokens < MAX_NUM_TOKENS	High	Resolved
[H-07]	Current pow math allows only integers	High	Resolved
[H-08]	Wrong usage of variables for _calculateSupply()	High	Resolved

[H-09]	Wrong loop area	High	Resolved
[H-10]	_packPoolVirtualBalance() reverts due to incorrect POOL_VB_MASK	High	Resolved
[H-11]	Missing _updateSupply Call in updateWeights	High	Resolved
[H-12]	Incorrect supply usage	High	Resolved
[H-13]	Incorrect supply update when adding token	High	Resolved
[H-14]	Incorrect parameters bricks certain functions	High	Resolved
[M-01]	Incorrect limit check in setWeightBands function	Medium	Resolved
[M-02]	Incorrect check in _calculateVirtualBalanceProd	Medium	Resolved
[L-01]	Same token twice not checked	Low	Acknowledged
[L-02]	Amplification should be above 1e18	Low	Resolved
[L-03]	Staking contract can be inflation attacked	Low	Acknowledged
[L-04]	Use SafeTransferLib to handle non-standard tokens	Low	Acknowledged
[L-05]	Unnecessary iterations due to incorrect bounds check	Low	Resolved
[L-06]	Missing Verification for tokenOut_ in swapExactOut	Low	Resolved
[L-07]	Incorrect slippage check in removeLiquiditySingle	Low	Resolved

8. Findings

8.1. Critical Findings

[C-01] Incorrect variable used when calculating supply

Severity

Impact: High

Likelihood: High

Description

When calculating the intermediate `_r` in the `_calculateSupply` function in the pool contract, the function uses the following logic.

```
_r = FixedPointMathLib.rawDiv(
    FixedPointMathLib.rawMul(_s, _sp), _s);
    // _r*_sp/_s
```

The comment is correct, but the implementation is incorrect. It should use `_r` instead of `s`.

This can also be seen in the yearn contract here: link

Recommendations

Fix the line to:

```
_r = FixedPointMathLib.rawDiv(
    FixedPointMathLib.rawMul(_r, _sp), _s);
    // _r*_sp/_s
```

[C-02] Wrong index used when updating rates

Severity

Impact: High

Likelihood: High

Description

When updating the rates in the `_updateRates` function, the function takes in the input as an integer which contains a bunch of indices that represent the index in the actual token array.

```
/// @param tokens_ integer where each byte represents a token index offset
    // by one
```

Next a loop with iterator `t` is used to take out the values.

```
for (uint256 t = 0; t < MAX_NUM_TOKENS; t++) {
            uint256 token = (tokens_ >> FixedPointMathLib.rawMul(8, t)) & 255;
```

Point to note, `t` is the iterator for processing the input. `token` is the actual extracted index that needs to be used to access the array element.

But the code mistakenly uses `t` to find the rate.

```
address provider = rateProviders[token];
// ...
uint256 _rate = IRateProvider(provider).rate(tokens[t]);
```

So instead of using `tokens[token]`, the rate is called wrongly for `tokens[t]`. Thus the code ends up using the rate for a completely different token.

The correct usage can be found in the YETH contract here: link

Using the wrong rate can throw off all calculations in the system and allow attackers to extract value from it.

Recommendations

Use `rate(tokens[token])`.

[C-03] Incorrect math operation leads to underprediction of stored product

Severity

Impact: High

Likelihood: High

Description

The function `swap` is used to exchange tokens. The contract keeps track of two values: a product and a sum of the virtual balances to calculate the exchange rate. In the `swap` function, the product is updated.

```
_virtualBalanceProd = _virtualBalanceProd
            + _powUp(_prevVirtualBalanceY, _wnY) / _powDown(
              (_virtualBalanceX * PRECISION) / _prevVirtualBalanceX, _wnX);
```

The issue is that instead of updating the product with multiplication and division, it is actually updating with an addition. This completely breaks the mathematics and the update procedure, leading to wrong exchange rates.

The correct version can be found in the yETH contract here: link

Recommendations

Change the addition to a multiplication.

```
_virtualBalanceProd = _virtualBalanceProd
            * _powUp(_prevVirtualBalanceY, _wnY) / _powDown(
              (_virtualBalanceX * PRECISION) / _prevVirtualBalanceX, _wnX);
```

[C-04] Missing division leads to massively deflated stored product

Severity

Impact: High

Likelihood: High

Description

During a swap, the product of the virtual balances needs to be adjusted for the fee. This is done in the following line. (Line 286)

```
_virtualBalanceProd = (_virtualBalanceProd * PRECISION)
                / _powDown(
                    (_virtualBalanceX + _changeInVirtualBalanceTokenIn)
                    * PRECISION,
                    _wnX
                    );
```

This expression is incorrect. The power should be raised on the percentage change, while here it is raised on the total change. So instead of doing a ((x+y)/x)^a, the code does a (x+y)^a. Not only can this lead to frequent overflows, but also leads to a massive devaluing of the stored product, leading to bad exchange rates in the system.

The correct implementation can be found in the yETH contract: link

The `powDown` function's parameter needs to be scaled down by `_virtualBalanceX`.

Recommendations

Change the expression to:

```
_virtualBalanceProd = (_virtualBalanceProd * PRECISION)
                / _powDown(
                (_virtualBalanceX + _changeInVirtualBalanceTokenIn)
                * PRECISION
                / _virtualBalanceX, _wnX);
```

[C-05] Incorrect subtraction leads to draining of the pool

Severity

Impact: High

Likelihood: High

Description

The function `removeLiquiditySingle` calculates the amount of tokens to be taken out when removing liquidity.

It calculates this by calculating the change in the virtual balance before and after liquidity removal.

```
uint256 _changeInVirtualBalance = _prevVirtualBalance = _virtualBalance;
```

However, as seen above, it uses an equality `=` operator instead of the minus `-` operator. Thus instead of calculating the difference, it sets the difference as `_virtualBalance`, which is the amount of liquidity that would have remained in the pool.

Thus if a user removes 1 wei, instead of calculating the difference as 1 wei, it sets the `_changeInVirtualBalance` as the entire balance of the pool. Thus an attacker can take out all the tokens from the pool using this flaw.

Recommendations

Change the second equality sign to a minus sign.

```
uint256 _changeInVirtualBalance = _prevVirtualBalance - _virtualBalance;
```

8.2. High Findings

[H-01] Bad `killed` check in `unpause` and `kill` functions

Severity

Impact: High

Likelihood: Medium

Description

Pools are designed to be paused, unpaused or killed. If killed, they cannot then be unpaused again.

Lets look at the `unpause` function.

```
if (!paused) revert Pool__NotPaused();
if (!killed) revert Pool__Killed();
paused = false;
```

The second line checks the killed status. It basically says that if the pool is not killed, the function should revert. This is a logical flaw. The function should revert IF the pool IS killed, not otherwise. The pool `killed` variable starts out as false. Thus if a pool is paused and then unpaused, the unpause will not happen since `!false=true`, so the revert will trigger.

A similar issue is present in the `kill` function.

```
if (!killed) revert Pool__Killed();
        killed = true;
```

Again, if the contract is already killed this should revert. In the current state, the contract can never be killed, since `killed` is set to false by default, so `!killed` will always be true, making this code unreachable.

Recommendations

In both cases, change the clause to `if(killed)`

[H-02] Missing negation in if clause

Severity

Impact: Medium

Likelihood: High

Description

When setting weights via the `setWeightBands` function, certain restrictions need to be followed. The weight has to be lower than the PRECISION value, which is 1e18.

This is because the sum of weights must add up to 1e18.

This check is incorrectly applied in the `setWeightBands` function.

```
if ((lower_[t] <= PRECISION && upper_[t] <= PRECISION)) {
                revert Pool__BandsOutOfBounds();
            }
```

As seen above, the contract will revert if both the weights are below PRECISION. Thus the function forces the setting of the weights to be above the 1e18 cap, which will break other parts of the system. The code in fact is missing a negation `!`, so the contract should revert only if the criteria is NOT satisfied, not if it is satisfied.

Recommendations

Change the if statement to:

```
if (!(lower_[t] <= PRECISION && upper_[t] <= PRECISION)) {
```

[H-03] Wrong out-of-bounds check in `swap` function

Severity

Impact: Medium

Likelihood: High

Description

The `swap` function makes sure the passed-in indices are not out of bounds.

```
if (tokenIn_ > _numTokens - 1 && tokenOut_ > _numTokens - 1) {
            revert Pool__IndexOutOfBounds();
        }
```

The issue is that in the above snippet, the contract reverts if BOTH the checks pass, so if BOTH the indices are out of bounds. If only a single one is out of bounds, the revert does not happen. It will instead use 0 values and lead to unpredictable behavior since the arrays have a size larger than the max number of tokens in the pool, so it won't naturally revert.

Recommendations

Change the `&&` to an `||`.

[H-04] `amplification_` vs `amplification` in the pool constructor

Severity

Impact: Medium

Likelihood: High

Description

The mistake made during the input validation:

```
if (amplification == 0) {
            revert Pool__MustBeInitiatedWithAGreaterThanZero();
        }

        amplification = amplification_;
```

link

Here the storage value checks when it was never written (equal to 0). As a result, every pool deployment will fail.

Recommendations

```
-         if (amplification == 0) {
+         if (amplification_ == 0) {
```

[H-05] Iterating the loop length below MAXLENGTH

Severity

Impact: Medium

Likelihood: High

Description

During the pool deployment, the transaction reverts for a pool with the length of tokens below `MAXLENGHT` which is `32`:

- link

It reverts with the out-of-bound error, so it is required to provide a long length of token of 32, which is not always the case.

Recommendations

The loop should iterate up to `_numTokens`. Or the loop should use this pattern from other functions when the loop reaches the final iteration:

```
if (t == _numTokens) {
                    break;
                }
```

[H-06] Impossible to add/remove liquidity if numTokens < MAX_NUM_TOKENS

Severity

Impact: Medium

Likelihood: High

Description

```
function addLiquidity(
        uint256[MAX_NUM_TOKENS] calldata amounts_,
        uint256 minLpAmount_,
        address receiver_
        )
        external
        nonReentrant
        returns (uint256)
    {
```

Here it required _amount to be provided with the length of 32. On the other hand, later it requires a different thing:

```
if (amounts_.length != _numTokens) revert Pool__InvalidParams();
```

So, these two requirements conflict when we have _numTokens below 32. Thus, it is impossible to provide liquidity for such pools.

Same thing for `removeLiquidity()`.

Recommendations

Consider changing to:

```
+ function addLiquidity(
        uint256[] calldata amounts_,
        uint256 minLpAmount_,
        address receiver_
        )
        external
        nonReentrant
        returns (uint256)
    {
```

The same thing should be fixed for `removeLiquidity()`.

[H-07] Current pow math allows only integers

Severity

Impact: Medium

Likelihood: High

Description

Pow math uses `FixedPointMathLib.rpow` which is not enough because it accepts only integers as a second param. link

But the expected pow calculations should allow calculations like 0.5 ^0.5 for instance (as it works with weights).

As a result, all current calculations using `FixedPointMathLib.rpow` will revert due to overflow.

Recommendations

Consider using alternative math libraries.

[H-08] Wrong usage of variables for _calculateSupply()

Severity

Impact: Medium

Likelihood: High

Description

`addLiquidity()` uses `_calculateSupply()` twice

```
(_supply, _virtualBalanceProd) = _calculateSupply(
            _numTokens,
            _prevSupply,
            amplification,
            _virtualBalanceProdFinal,
            _virtualBalanceSumFinal,
            true
        );
```

link

There are a few mistakes:

1. `_prevSupply` is used, while `_supply` is correct Otherwise, it reverts on the first liquidity provision because of zero requirement here link
2. `_virtualBalanceProdFinal` is used, while `_virtualBalanceProd` is correct
3. `_virtualBalanceSumFinal` is used, while `_virtualBalanceSum` is correct
4. `true` is used as the last argument while it is true when `_prevSupply == 0`. So it should be replaced with `_prevSupply == 0`.

Recommendations

Apply necessary adjustments in the first usage of `_calculateSupply()`. The second usage after that is correct.

[H-09] Wrong loop area

Severity

Impact: Medium

Likelihood: High

Description

The loop here does not end correctly according to the desired formula calculation. link

Compare to this calculation in yETH: link

The loop ends before `delta = 0`.

Recommendations

These lines should be included in the loop only

```
for (uint256 t = 0; t < MAX_NUM_TOKENS; t++) {
                if (t == numTokens_) break;
                _r = FixedPointMathLib.rawDiv(FixedPointMathLib.rawMul
                  (_s, _sp), _s);
+               }
                uint256 _delta = 0;
                ...
```

And change the following brackets accordingly.

[H-10] `_packPoolVirtualBalance()` reverts due to incorrect `POOL_VB_MASK`

Severity

Impact: Medium

Likelihood: High

Description

All calculations with POOL_VB_MASK always revert due to a wrong value stored. E.g. reverts in `_packPoolVirtualBalance()` during the liquidity provision.

Recommendations

`POOL_VB_MASK` value should be fixed:

```
- uint256 public constant POOL_VB_MASK = 2 * 128 - 1;
+ uint256 public constant POOL_VB_MASK = 2 ** 128 - 1;
```

[H-11] Missing `_updateSupply` Call in `updateWeights`

Severity

Impact: Medium

Likelihood: High

Description

The `updateWeights` function updates the virtual balance product (`vb_prod`) but does not include a call to `_updateSupply`. This inconsistency can result in the use of a stale `supply`, affecting the accuracy of swaps and liquidity adjustments within the pool.

The issue is that this function does not update the supply. Since the weights and balances are updated, the supply also needs to be updated to sync up the state of the pool. The supply update is present in the yETH pool here: link

Recommendations

Incorporate a call to `_updateSupply` within the `updateWeights` function to ensure that the supply state is synchronized:

```
function updateWeights() external returns (bool) {
    _checkIfPaused();
    bool _updated = false;
    (
      uint256 _virtualBalanceProd,
      uint256 _virtualBalanceSum
    ) = _unpackPoolVirtualBalance(packedPoolVirtualBalance
    (_virtualBalanceProd, _updated) = _updateWeights(_virtualBalanceProd);
    if (_updated && _virtualBalanceSum > 0) {
+       uint256 _supply;
+       (_supply, _virtualBalanceProd) = _updateSupply
+ (supply, _virtualBalanceProd, _virtualBalanceSum);
        packedPoolVirtualBalance = _packPoolVirtualBalance
          (_virtualBalanceProd, _virtualBalanceSum);
    }
    return _updated;
}
```

[H-12] Incorrect supply usage

Severity

Impact: Medium

Likelihood: High

Description

The contract uses the global variable `supply` in some places, and the function value `totalSupply` in other places. The issue is that these two values are not the same.

`totalSupply` is the total minted out LP tokens in circulation.

`supply` is a value calculated using Newton-Raphson iterations based on the weights, sums and products of the tokens composing the pool.

These two values are only in sync after calling the `_calculateSupply` followed by `_updateSupply`, since the latter function burns or mints the LP token supply in order to match the two quantities.

However this is not respected in the codebase, and `totalSupply` is used in places where `supply` should be used.

One example is in the `removeLiquidity` function. Let's assume the two values `supply` and `totalSupply` are in sync before this function call. In this function, first, the LP tokens are burnt.

```
uint256 _prevSupply = totalSupply();
_burn(msg.sender, lpAmount_);
```

This will immediately make the two values different, since `totalSupply` just went down, but `supply` isn't updated. Then when the product is being calculated, the `totalSupply` is used, which is now different from the `supply` value.

```
_virtualBalanceProd = FixedPointMathLib.rawDiv(
                FixedPointMathLib.rawMul(
                    _virtualBalanceProd,
                    _powDown(
                        FixedPointMathLib.rawDiv(FixedPointMathLib.rawMul
                        //(totalSupply(), __weight), vb),  //@audit totalSupply is use
                        FixedPointMathLib.rawMul(__weight, _numTokens)
                    )
                ),
                PRECISION
            );
```

This is different from the calculations in the whitepaper. It should instead use `supply`. This is also shown in the yETH code base here: link

Due to this issue, there will be a discrepancy in the product calculated. This can lead to bad exchange rates down the line.

Recommendations

Replace every instance of `totalSupply` usage with `supply` or its equivalent as in the yETH contract. `totalSupply` can only be used if the two values have been synced first.

[H-13] Incorrect supply update when adding token

Severity

Impact: High

Likelihood: Medium

Description

The function `add_token` calculates the supply, and then mints to the caller the appropriate number of LP tokens. The function also takes in `amplification_`, the new pool state's amplification factor.

To calculate the new supply, the pool must use this new amplification value, since it is the new pool state.

```
function addToken(
        uint256 amplification_,
        // ...
    ) external onlyOwner {
        (__supply, _virtualBalanceProd) = _calculateSupply(
            _numTokens,
            _virtualBalanceSum,
            amplification,
            _virtualBalanceProd,
            _virtualBalanceSum,
            true
        );
```

However, as seen above, the function stores the new amplification in `aplification_`, but uses the old `amplification` to calculate the supply. This is incorrect and leads to bad values being calculated since the change in amplification has not been accounted for.

The correct usage can be found in the yETH contract: link

The new amplification value should be used to calculate the supply.

Recommendations

Change `amplification` to `amplification_`

[H-14] Incorrect parameters bricks certain functions

Severity

Impact: Medium

Likelihood: High

Description

In solidity, if a function parameter list has an array of defined length, it forces the user to pass an array of that length. The call reverts if the passed array is of an unexpected length.

This is the case in the `setWeightBands` and `setRamp` functions.

```
function setWeightBands(
        uint256[MAX_NUM_TOKENS] calldata tokens_,
        uint256[MAX_NUM_TOKENS] calldata lower_,
        uint256[MAX_NUM_TOKENS] calldata upper_
    )
```

Since `MAX_NUM_TOKENS=32`, this forces the caller to pass in an array with 32 elements. Any lower and this call reverts. This is also present in the `setRamp` function.

```
function setRamp(
        uint256 amplification_,
        uint256[MAX_NUM_TOKENS] calldata weights_,
        uint256 duration_,
        uint256 start_
    )
```

However, these functions expect arrays only of the size `numTokens`. In fact, if a larger array is passed, it will revert due to another line.

```
for (uint256 t = 0; t < MAX_NUM_TOKENS; t++) {
            //...
             if (t >= _numTokens) revert Pool__IndexOutOfBounds();
```

So users are forced to pass in 32-length arrays or the call reverts, and if they do so, the call will still revert since `_numTokens` will be exceeded if the pool does not actually use 32 tokens.

So all calls to both these functions will revert. Thus these functions are unusable.

Recommendations

Remove the size requirement from the function parameters.

```
function setWeightBands(
        uint256[] calldata tokens_,
        uint256[] calldata lower_,
        uint256[] calldata upper_
    )
```

```
function setRamp(
        uint256 amplification_,
        uint256[] calldata weights_,
        uint256 duration_,
        uint256 start_
    )
```

8.3. Medium Findings

[M-01] Incorrect limit check in `setWeightBands` function

Severity

Impact: Medium

Likelihood: Medium

Description

The `setWeightBands` function is used to set the weight bands of tokens. It takes in a `tokens` array, which contains the list of indices of the tokens to operate on.

So to check the limits, the value of `tokens[i]` should be checked.

```
for (uint256 t = 0; t < MAX_NUM_TOKENS; t++) {
            //...
            if (t >= _numTokens) revert Pool__IndexOutOfBounds();
```

In the code, `t` is the iterator which is used to take out the indices from `tokens` array. The issue is that the limit check is done on the iterator `t`, instead of doing it on the actual index, which is `tokens_[t]`. This leads to incorrect parsing of the error.

Since the arrays in the contract are oversized and of the size `MAX_NUM_TOKENS`, it won't naturally revert when accessing an out-of-size element. Instead, the weight of an unused token slot will be set.

Recommendations

Change the `Pool__IndexOutOfBounds` check to

```
if (tokens_[t] >= _numTokens) revert Pool__IndexOutOfBounds();
```

[M-02] Incorrect check in `_calculateVirtualBalanceProd`

Severity

Impact: Medium

Likelihood: Medium

Description

The `_calculateVirtualBalanceProd` function should revert if the `weight` is 0 or if the `virtualBalance` of the token is 0.

However, in the implementation, it only breaks if both are 0.

```
if (_weight <= 0 && _virtualBalance <= 0) {
                break;
            }
```

This has two problems:

1. If either 1 of the values is zero, the function doesn't revert and the calculation gets processed like a normal transaction
2. The transaction isn't reverted. Instead, due to the break statement, the returned value is just equal to PRECISION. This can lead to incorrect results.

The correct implementation can be found in the Yeth contract here: link

Recommendations

Change the `&&` to an `||`, and the `break` to a revert.

8.4. Low Findings

[L-01] Same token twice not checked

Pool constructor does not check that a token is provided twice in the length of tokens. Actually, the whole pool can work given this mistake. But it would allow an attack using `skim()` based on the difference between the virtual balance and the actual balance - the actual balance will include the balance for two virtual balances and it can be withdrawn more than expected.

Consider checking that the same token is not provided twice on deployment.

[L-02] Amplification should be above `1e18`

Amplification is not checked for being above `1e18` on the pool deployment. If it is below any liquidity provision would fail here: link

Consider reverting in the pool constructor when the provided Amplification is below

[L-03] Staking contract can be inflation attacked

The staking contract is an ERC4626 contract. The ERC4626 contract has a common flaw, where the initial depositor can `donate` a large number of tokens to the vault to increase the share ratio, and then cause rounding errors on future depositors and steal their deposits.

The attack happens in the following steps:

1. Initial user deposits 1 wei of token, and mints 1 wei of shares.
2. They send 1e18 wei of tokens to the staking contract via an external transfer.
3. The second depositor comes and deposits 2e18 wei of tokens. The contract believes it has minted 1 wei shares for 1e18+1 wei of tokens. So it mints the depositor (2e18/(1e18+1) * 1) = 1 wei of shares.

4. Both the first depositor and second depositor have 1 wei of shares. So first depositor claims 1.5e18 tokens, netting them a gain of 0.5e18 tokens. the second depositor loses the same amount.

A common way to prevent this issue is by having the protocol devs deposit at least 1e6 wei of EACH token into the protocol, and then never removing them.

[L-04] Use `SafeTransferLib` to handle non-standard tokens

The contract transfers tokens with the raw `transfer` and `transferFrom` functions. The contract also sometimes checks the return value with an `if(!ERC20(token).transfer(...))` clause. A better way to handle this is by using the `SafeTransferLib` library which implements the `safeTransferFrom` function, which can handle a wide variety of token behaviors.

[L-05] Unnecessary iterations due to incorrect bounds check

The `_updateWeights` function updates the weights of the tokens. It loops over the tokens and updates the weights and virtual balances.

```
for (uint256 t = 0; t < MAX_NUM_TOKENS; ++t) {
            if (t <= numTokens) {
                //...
            }
        }
```

The issue is that the `if` check inside still runs the code if `t==numTokens`. Solidity has 0 indexed arrays, and when the iterator is equal to `numTokens`, it is already outside bounds. Thus the code does an extra unnecessary loop. this does not revert and only reads and sets 0 values, so has no actual impact other than extra gas.

[L-06] Missing Verification for `tokenOut_` in `swapExactOut`

The `swapExactOut` function correctly verifies several conditions for `tokenIn_`, but it lacks a similar verification for `tokenOut_`, which could lead to indexing issues if `tokenOut_` exceeds the number of tokens managed by the contract. This could potentially result in undefined behavior or errors at runtime. To address this, add a boundary check for `tokenOut_` similar to `tokenIn_` to ensure it is within valid range: `if (tokenOut_ >= _numTokens) revert Pool__IndexOutOfBounds();`. Implementing this check ensures robustness and prevents possible runtime errors due to invalid token indices.

[L-07] Incorrect slippage check in `removeLiquiditySingle`

In the `removeLiquiditySingle` function, the slippage check incorrectly uses a strict inequality, causing the function to revert if the `_tokenOutAmount` is exactly equal to `minTokenOutAmount_`. This behavior is inconsistent with typical slippage checks, which generally allow the transaction to proceed if the output amount meets or exceeds the minimum expected amount. Correct this by changing the conditional check to `if (_tokenOutAmount < minTokenOutAmount_) revert Pool__SlippageLimitExceeded();`. This adjustment aligns the function with common expectations and ensures consistency across similar checks in other functions.