CANTINA

Opal Protocol

Competition

March 27, 2024

Contents

1 Introduction

1.1 About Cantina

Cantina is a security services marketplace that connects top security researchers and solutions with clients. Learn more at cantina.xyz

1.2 Disclaimer

A competition provides a broad evaluation of the security posture of the code at a particular moment based on the information available at the time of the review. While competitions endeavor to identify and disclose all potential security issues, they cannot guarantee that every vulnerability will be detected or that the code will be entirely secure against all possible attacks. The assessment is conducted based on the specific commit and version of the code provided. Any subsequent modifications to the code may introduce new vulnerabilities, therefore, any changes made to the code would require an additional security review. Please be advised that competitions are not a replacement for continuous security measures such as penetration testing, vulnerability scanning, and regular code reviews.

1.3 Risk assessment

Severity	**Description**
Critical	*Must* fix as soon as possible (if already deployed).
High	Leads to a loss of a significant portion (>10%) of assets in the protocol, or significant harm to a majority of users.
Medium	Global losses <10% or losses to only a subset of users, but still unacceptable.
Low	Losses will be annoying but bearable. Applies to things like griefing attacks that can be easily repaired or even gas inefficiencies.
Gas Optimization	Suggestions around gas saving practices.
Informational	Suggestions around best practices or readability.

1.3.1 Severity Classification

The severity of security issues found during the security review is categorized based on the above table. Critical findings have a high likelihood of being exploited and must be addressed immediately. High findings are almost certain to occur, easy to perform, or not easy but highly incentivized thus must be fixed as soon as possible.

Medium findings are conditionally possible or incentivized but are still relatively likely to occur and should be addressed. Low findings a rare combination of circumstances to exploit, or offer little to no incentive to exploit but are recommended to be addressed.

Lastly, some findings might represent objective improvements that should be addressed but do not impact the project's overall security (Gas and Informational findings).

2 Security Review Summary

Opal is a protocol built on Ethereum, aiming to enhance Dexs' liquidity flywheel, starting with the Balancer ecosystem. It employs yield bearing "Omnipools", which are liquidity pools where users can deposit a single asset.

From Feb 12th to Feb 20th Cantina hosted a competition based on opal-contracts. The participants identified a total of **238** issues in the following risk categories:

- Critical Risk: 1
- High Risk: 22
- Medium Risk: 23
- Low Risk: 63
- Gas Optimizations: 61
- Informational: 68

The present report only outlines the **critical**, **high** and **medium** risk issues.

3 Findings

3.1 Critical Risk

3.1.1 You can `deposit` and `withdraw` simultaneously in the same block to manipulate exchangerate

Submitted by 8olidity, also found by giraffe0x, Chad0, qckhp and recursive

Severity: Critical Risk

Context: Omnipool.sol#L361

Description: In the depositFor and withdraw functions, the user's operations will be restricted so that the user cannot perform deposit and withdraw operations in the same block.

```
if (lastTransactionBlock[msg.sender] == block.number) {
    revert CantDepositAndWithdrawSameBlock();
}
```

However, since lptoken can be transferred, an attacker only needs two accounts to perform deposit and withdraw operations at the same time, thereby bypassing the above restrictions and further manipulating the exchangeRate.

Proof of concept:

- File: `test\Omnipool.fkt.sol`

```
interface IUSDC {
    function balanceOf(address account) external view returns (uint256);
    function mint(address to, uint256 amount) external;
    function configureMinter(address minter, uint256 minterAllowedAmount) external;
    function masterMinter() external view returns (address);
}
contract OmnipoolTest is SetupTest {
    IUSDC usdc = IUSDC(address(0xA0b86991c6218b36c1d19D4a2e9Eb0cE3606eB48));

    function testdepositwithdrawsameblock() public {
        address alice = address(0x1234567890123456789012345678901234561234);
        address bob = address(0x1234567890123456789012345678901234561235);

        address poolAddress = pools[0];
        IOmnipool pool = IOmnipool(poolAddress);
        IERC20Metadata token = IERC20Metadata(pool.getUnderlyingToken());
        console.log("-----");
        console.log("Depositing into Pool: %s", token.symbol());
        uint256 depositAmount = 10_000 * 10 ** token.decimals();
        vm.startPrank(usdc.masterMinter());
        usdc.configureMinter(alice, type(uint256).max);
        vm.stopPrank();

        vm.startPrank(alice);
        usdc.mint(alice, 100_000 * 10 ** token.decimals());
        vm.stopPrank();

        vm.startPrank(alice);
        token.approve(poolAddress, 10_000 * 10 ** token.decimals());
        pool.deposit(depositAmount, 1);
        uint256 stakedBalance = IERC20(pool.getLpToken()).balanceOf(alice);
        assertApproxEqRel(stakedBalance, depositAmount, 0.1e18);
        stakedBalances[poolAddress] = stakedBalance;
        console.log("Successfully deposited into pool: %s", token.symbol());
        IERC20(pool.getLpToken()).transfer(bob, stakedBalance);
        vm.stopPrank();

        vm.startPrank(bob);
        console.log("-----");
        console.log("Withdrawing from Pool: %s", token.symbol());
        uint256 underlyingBefore = token.balanceOf(bob);
        uint256 withdrawAmount = depositAmount / 2;
        pool.withdraw(withdrawAmount, withdrawAmount / 2);
        uint256 underlyingDiff = token.balanceOf(bob) - underlyingBefore;
        assertApproxEqRel(depositAmount / 2, underlyingDiff, 0.1e18);
        console.log("Successfully withdrew from pool: %s", token.symbol());
        vm.stopPrank();
    }
}
```

Output:

```
Running 1 test for test/Omnipool.fkt.sol:OmnipoolTest
[PASS] testdepositwithdrawsameblock() (gas: 4119596)
Logs:
  priceFeedOracle: 0xF67b8C6dB7601F45e27898ddA6D83c1EFd64aA4B
  setup oracle: 0xF67b8C6dB7601F45e27898ddA6D83c1EFd64aA4B
  Omnipool address: 0x6B950684E884e20ef4d61cb5A3ab2d87Eacb7372
  -----
  Depositing into Pool: USDC
  Successfully deposited into pool: USDC
  -----
  Withdrawing from Pool: USDC
  Successfully withdrew from pool: USDC

Test result: ok. 1 passed; 0 failed; 0 skipped; finished in 116.27s

Ran 1 test suites: 1 tests passed, 0 failed, 0 skipped (1 total tests)
```

The above proof of concept only demonstrates deposit and withdraw operations in the same block. This allows many attacks, such as

1. A user deposits `100` USDC (block 100).
2. User B deposits `1_000_000_000` USDC. At this time, the exchange has changed. User A withdraws `lptoken`. User B transfers `lptoken` to user C, and user C calls the withdraw operation (block 101).

Recommendation: It is recommended to whitelist restrictions on the transfer function of `lptoken`.

3.2 High Risk

3.2.1 Oracle manipulation via missing balancer vault read-only reentrancy check

Submitted by cuthalion0x, also found by Bauer, giraffe0x, pks271 and 0xadrii

Severity: High Risk

Context: BPTOracle.sol#L39

Description: The Balancer team announced a vulnerability related to read-only reentrancy in the Balancer Vault.

The issue involves a race condition between data reported by the Balancer Vault and data reported by Balancer pools. Any piece of external code querying both simultaneously:

1. Data from the Balancer Vault (usually token balances via `Vault.getPoolTokens()`).
2. Data from Balancer pools (usually BPT supply via `ERC20.totalSupply()`).

They can be manipulated by an attacker via a pool join or exit, as the mismatched data at the point of reentrancy will be used to compute highly inaccurate BPT prices. Some calls to just the pool (such as `StablePool.getRate()` or `WeightedPool.getInvariant()`) also query the Vault, thus fulfilling both data requests with only a single call from the integrating code base.

The Balancer team introduced a very simple fix: any piece of code querying the data mentioned above should protect itself by calling `VaultReentrancyLib.ensureNotInVaultContext()` to force a trigger of the Balancer Vault's internal reentrancy check within a read-only call. Balancer's provided function can be found here.

This code base includes the required library in `src/utils/VaultReentrancyLib.sol` but fails to properly utilize it. The code never actually invokes the `ensureNotInVaultContext()` function; the library is only referenced at src/pools/BPTOracle.sol#L39, which is insufficient.

Impact: Oracle manipulation can manifest in myriad ways, typically resulting in loss of funds for users or the protocol itself via incorrectly computed swap limits or shares issuance.

Recommendation: Add a call to `VaultReentrancyLib.ensureNotInVaultContext()` within each BPT pricing function:

- `BPTOracle.BptPriceStablePool()`
- `BPTOracle.BptPriceWeightPool()`
- `BPTOracle.BptPriceComposablePool()`

Or else within the top-level `BPTOracle.getPoolValuation()` function.

A diff adding one possible implementation is included below. Note that since this code base uses its own version of `IVault` called `IBalancerVault`, there is some extra type casting in this implementation. Ideally, only Balancer's official `IVault` would be used, but this fix makes no effort to address that issue.

```
diff --git a/src/pools/BPTOracle.sol b/src/pools/BPTOracle.sol
index 3fe2438..9ab82a8 100644
--- a/src/pools/BPTOracle.sol
+++ b/src/pools/BPTOracle.sol
@@ -12,6 +12,8 @@ import {IManagedPool} from
     "balancer-v2-monorepo/pkg/interfaces/contracts/pool-utils/IManagedPool.sol";
 import {IExternalWeightedMath} from
     "balancer-v2-monorepo/pkg/interfaces/contracts/pool-weighted/IExternalWeightedMath.sol";
+import {IVault} from
+    "balancer-v2-monorepo/pkg/interfaces/contracts/vault/IVault.sol";
 import {IBalancerPool} from "src/interfaces/Balancer/IBalancerPool.sol";
 import {IBalancerVault} from "src/interfaces/Balancer/IBalancerVault.sol";
 import {IPriceFeed} from "src/interfaces/IPriceFeed.sol";
@@ -36,8 +38,6 @@ import {
  * @dev A smart contract for providing price information for Balancer pools in various types.
  */
 contract BPTOracle {
-    using VaultReentrancyLib for IBalancerVault;
-
     using PRBMathUD60x18 for uint256;

     /**
@@ -91,6 +91,7 @@ contract BPTOracle {
      * @return The USD price for the stable pool.
      */
     function BptPriceStablePool(bytes32 poolId) public view returns (uint256) {
+        VaultReentrancyLib.ensureNotInVaultContext(IVault(address(balancerVault)));
         (address[] memory tokens,,) = balancerVault.getPoolTokens(poolId);

         (address poolAddress,) = balancerVault.getPool(poolId);
@@ -126,6 +127,7 @@ contract BPTOracle {
      * @return The USD price for the weighted pool.
      */
     function BptPriceWeightPool(bytes32 poolId) public view returns (uint256) {
+        VaultReentrancyLib.ensureNotInVaultContext(IVault(address(balancerVault)));
         (address[] memory tokens,,) = balancerVault.getPoolTokens(poolId);

         (address poolAddress,) = balancerVault.getPool(poolId);
@@ -173,6 +175,7 @@ contract BPTOracle {
      * @return The USD price for the composable pool.
      */
     function BptPriceComposablePool(bytes32 poolId) public view returns (uint256) {
+        VaultReentrancyLib.ensureNotInVaultContext(IVault(address(balancerVault)));
         (address[] memory tokens,,) = balancerVault.getPoolTokens(poolId);

         (address pool,) = balancerVault.getPool(poolId);
```

3.2.2 Attacker can censor liquidity providers deposits and withdrawals by front-running

Submitted by zigtur, also found by J4X98, kodyvim, bronzepickaxe, qckhp, Sujith Somraaj, 0xadrii, 0xJaeger, Victor Okafor and kogekar

Severity: High Risk

Context: Omnipool.sol#L239, Omnipool.sol#L361

Description: The `depositFor` and `withdraw` functions in `Omnipool` require that the address to which funds will be deposited/withdrawn has not executed deposit or withdraw transactions in the current `block.number`. In `depositFor`, this is done through the check attached to the finding.

In `depositFor` function, it is the `_depositFor` address that gets its mapping updated. An attacker can use `depositFor` to deposit a small underlyingToken amount to the `_depositFor` address. Then, this will deny the `_depositFor` address from executing any other deposits or withdrawals in the current block.

Moreover, as there are no minimum deposit amount, the attack is cheap to execute.

Likelihood and Impact: The attacker can front-run every deposit/withdrawal attempt from an address to lock its funds.

- Impact: High - Loss of user funds.
- Likelihood: Medium - Future users will be protocols with large amounts of funds that will be targeted by attackers.

Proof of Concept: The following code shows the structure of `depositFor` and `withdraw`:

```
function depositFor(uint256 _amountIn, address _depositFor, uint256 _minLpReceived) public {
    if (lastTransactionBlock[_depositFor] == block.number) {
        revert CantDepositAndWithdrawSameBlock();
    }

    // Do deposit actions


    lastTransactionBlock[_depositFor] = block.number; // here attacker control `_depositFor`
}

function withdraw(uint256 _amountOut, uint256 _minUnderlyingReceived) external override {
    if (lastTransactionBlock[msg.sender] == block.number) {
        revert CantDepositAndWithdrawSameBlock();
    }

    // Withdrawal actions

    lastTransactionBlock[msg.sender] = block.number;
}
```

At the early stage of the deposit/withdraw, the transaction is reverted when the last transaction was executed in the current block.

`_depositFor` is the recipient of the deposit, which is controlled by the attacker.

Recommendation: Multiple fixes could be implemented:

1. Restrict the protocol so users can deposit only for themselves.
2. Delete the lastTransactionBlock checks. **Note that this fix may bring unexpected issues and should be further studied.**

3.2.3 `Bptoracle.bptpriceweightpool()` tries to use `totalsupply()` to get the total supply of the pools, resulting in inaccurate `bptprice`

Submitted by AuditorPraise, also found by cuthalion0x, ZanyBonzy, Bauer, GeneralKay, crypticdefense, pks271, 0xadrii and 0xRizwan

Severity: High Risk

Context: BPTOracle.sol#L163, Omnipool.sol#L186

Description: Balancer pools have different methods to get their total supply of minted LP tokens, which is also specified in the docs here.

The docs specify the fact that `totalSupply` is only used for older stable and weighted pools, and should not be used without checking it first, since the newer pools have pre-minted BPT and `getActualSupply` should be used in that case. Most of the time, the assumption would be that only the new composable stable pools uses the `getActualSupply`, but that is not the case, since even the newer weighted pools have and uses the `getActualSupply`.

To give you few examples of newer weighted pools that uses `getActualSupply`, check addresses `0x9f9d...42f298`, `0x3ff3...c56a2e` and `0xcf7b...a1d52a`, the last one also being on Aura finance.

Because of that, attempting `BptPriceWeightPool()` on newer weighted pools and also the future weighted pools would result in inaccurate prices since the wrong total supply value is used when calculating price:

```
// 5. BPT Price (USD) = TVL / totalSupply
uint256 bptPrice = uint256((numerator.toInt().div(totalSupply)));
```

This will cause losses in `Omnipool._withdrawFromAuraPool()._bptPrice` won't be accurate. `Omnipool.getPoolTvl()` will also use wrong bptPrice.

Recommendation: Try to check first if the weighted pool you are interacting with is a newer one and uses the `getActualSupply` or if it is an older one and uses totalSupply, in that way the protocol could interact with multiple pools in the long run.

3.2.4 Theft of rewards via sandwich attack

Submitted by cuthalion0x, also found by zigtur, Bauer, giraffe0x, bronzepickaxe, qckhp, Naveen Kumar Naik J - 1nc0gn170, tsvetanovv, 0xhashiman and Victor Okafor

Severity: High Risk

Context: Omnipool.sol#L831-L834

Description: The `Omnipool.swapForGem()` function makes a Balancer `batchSwap()` without any slippage protection, thus leaving it exposed to sandwich attacks. These sandwich attacks can be used to steal nearly 100% of users' rewards as they attempt to claim.

The user-facing entry point of the `Omnipool.swapForGem()` function is `RewardManager.claimEarnings()`, and at no point throughout the claim pipeline is any slippage protection introduced. The function makes a blind Balancer `batchSwap()` with infinite `limits` as shown below, meaning it will accept any arbitrary exchange rate for the tokens including a near-zero price:

```
int256[] memory limits = new int256[](3);
limits[0] = type(int256).max;
limits[1] = type(int256).max;
limits[2] = type(int256).max;
```

Impact: Theft of user rewards up to 100%.

Recommendation: There are two options:

1. Allow users to set their own minimum reward amount via the `RewardManager.claimEarnings()` function. This minimum amount could be automatically calculated and populated by a user interface so that the user experience remains unchanged.
2. Use price oracles to calculate the Balancer `limits` that would constrain slippage to a set tolerance such as 0.5%.

3.2.5 `lastweightupdate` mapping in `omnipoolcontroller.sol` will always be 0

Submitted by dirtymic, also found by J4X98, kustrun, qckhp, crypticdefense and zanderbyte

Severity: High Risk

Context: *(No context files were provided by the reviewer)*

`OmnipoolController::lastWeightUpdate[address]` mapping is never updated. This has an effect on `GemMinterRebalancingReward::computeRebalancingRewards()` in the line `uint256 elapsedSinceUpdate = uint256(block.timestamp) - lastWeightUpdate;`. `elapsedSinceUpdate` will always be block.timestamp giving users more rewards than they should be receiving.

The following test shows getLastWeightUpdate amounting to 0 after `updateWeights` is called:

```
function testGetLastWeightUpdate() public {
    deal(
        address(gem),
        0x1234567890123456789012345678901234561234,
        type(uint256).max
    );

    vm.startPrank(0x1234567890123456789012345678901234561234);
    IERC20(gem).approve(
        registryContract.getContract(
            CONTRACT_GEM_MINTER_REBALANCING_REWARD
        ),
        type(uint256).max
    );
    uint256[] memory initWeight = omnipool.getAllUnderlyingPoolWeight();
    for (uint256 i = 0; i < initWeight.length; i++) {
        console.log("init weight: %s", initWeight[i]);
    }

    uint256 decimals = 6;
    vm.startPrank(user);
    address poolAddress = address(omnipool);
    console.log(poolAddress);
    IERC20(0xA0b86991c6218b36c1d19D4a2e9Eb0cE3606eB48).approve(
        address(omnipool),
        100_000 * 10 ** decimals
    );

    vm.stopPrank();

    vm.roll(1);
    vm.prank(user);
    omnipool.deposit(10_000 * 10 ** decimals, 1);

    skip(14 days);

    IOmnipoolController.WeightUpdate[]
        memory newWeights = new IOmnipoolController.WeightUpdate[](3);
    newWeights[0] = IOmnipoolController.WeightUpdate(TRI_POOL, 0.8e18);
    newWeights[1] = IOmnipoolController.WeightUpdate(USDC_STG, 0.2e18);
    newWeights[2] = IOmnipoolController.WeightUpdate(USDC_DOLA, 0);
    vm.prank(opal);
    controller.updateWeights(poolAddress, newWeights);
    initWeight = omnipool.getAllUnderlyingPoolWeight();
    for (uint256 i = 0; i < initWeight.length; i++) {
        console.log("init weight: %s", initWeight[i]);
    }

    console.log(
        "rate update: %s",
        controller.getLastWeightUpdate(address(omnipool))
    );

    skip(1 hours);

    assertTrue(omnipool.rebalancingRewardActive());

    uint256 deviationBefore = omnipool.computeTotalDeviation();
    uint256 gemBalanceBefore = IERC20(gem).balanceOf(user);
    vm.roll(2);
    vm.prank(user);
    omnipool.deposit(10_000 * 10 ** decimals, 1);
    uint256 deviationAfter = omnipool.computeTotalDeviation();
    assertLt(deviationAfter, deviationBefore);
    uint256 gemBalanceAfter = IERC20(gem).balanceOf(user);
    assertGt(gemBalanceAfter, gemBalanceBefore);
    console.log("reward user balance before: %s", gemBalanceBefore);
    console.log("reward user balance after: %s", gemBalanceAfter);
}
```

3.2.6 The `bptpricestablepool()` function of `bptoracle.sol` contract calculates the value of stable bpt incorrectly

Submitted by GeneralKay, also found by ZanyBonzy, Bauer, pks271 and 0xRizwan

Severity: High Risk

Context: BPTOracle.sol#L119

Description: This implementation of valuing Stable BPT is incorrect and can lead to incorrect valuation of the Stable BPT because rates are not considered.

The `BptPriceStablePool()` function of `BPTOracle.sol` contract calculates the value of Stable BPT incorrectly.

In the `BptPriceStablePool()` function of `BPTOracle.sol` contract, the price of the pool token for a Balancer stable pool is calculated using the formula: `min * IRateProvider(poolAddress).getRate() / 1e18`. where `min` is the minimum price from comparing the price of each token in the pool.

- File: `BPTOracle.sol`

```
function BptPriceStablePool(bytes32 poolId) public view returns (uint256) {
    (address[] memory tokens,,) = balancerVault.getPoolTokens(poolId);

    (address poolAddress,) = balancerVault.getPool(poolId);
    uint256 min = type(uint256).max;
    address token;
    uint256 length = tokens.length;
    for (uint256 i; i < length;) {
        token = address(tokens[i]);

        if (token == poolAddress) {
            unchecked {
                ++i;
            }
            continue;
        }

        uint256 value = getUSDPrice(token);
        if (value < min) {
            min = value;
        }

        unchecked {
            ++i;
        }
    }
    return (min * IRateProvider(poolAddress).getRate()) / 1e18;
}
```

The current implementation to calculated BPT price is flawed because the constituent tokens in a Stable pool may have different peg levels so using the minimum formula without considering this variation can lead to significantly incorrect results.

The correct approach to calculating the minimum price is to consider both the market price and the RateProvider price, normalizing them for a fair comparison. See the Balancer docs for details of this correct valuation.

However, the Balancer docs shows how to correctly calculate it by first dividing the individual prices by their respective rates before finding the minimum.

Impact: Wrong valuation of BPT which can be over/under valued.

Recommendation: For pools having rate providers, divide prices by rate before choosing the minimum and when no `RateProvider` is available use `1e18` as recommended in the link to the docs below.

The right valuation calculation is in the in the `#solution` heading of the Balancer docs.

```
   function BptPriceStablePool(bytes32 poolId) public view returns (uint256) {
       (address[] memory tokens,,) = balancerVault.getPoolTokens(poolId);

       (address poolAddress,) = balancerVault.getPool(poolId);
       uint256 min = type(uint256).max;
       address token;
       uint256 length = tokens.length;
       for (uint256 i; i < length;) {
           token = address(tokens[i]);

           if (token == poolAddress) {
               unchecked {
                   ++i;
               }
               continue;
           }

           uint256 value = getUSDPrice(token);
++        value = value * 1e18 / pool.getTokenRate(token)
           if (value < min) {
               min = value;
           }

           unchecked {
               ++i;
           }
       }
       return (min * IRateProvider(poolAddress).getRate()) / 1e18;
  }
```

3.2.7 Incorrect assumptions about bptindex may lead to incorrect input amounts in `_deposittoaurapool`

Submitted by zigtur, also found by Bauer, 0xadrii and Naveen Kumar Naik J - 1nc0gn170

Severity: High Risk

Context: Omnipool.sol#L445-L448, Omnipool.sol#L557

Description: The `Omnipool._depositToAuraPool` function checks if the assets include BPT by checking if `_pool.bptIndex > 0`. If so, it reduces the `_pool.assetIndex` by 1 before setting the value.

The problem lies in the fact that there is no check that `bptIndex < assetIndex`. When it is the case, the position of `assetIndex` should not be reduced by 1.

This issue wrongly formats the `userDataAmountsIn` array before executing the `JoinPoolRequest`.

Impact: Wrong amounts will be sent to the Pool for deposits, leading to potential loss of funds.

Proof of concept: Let's demonstrate the vulnerability with an example:

- `_underlyingAmountIn = 123`.
- `_pool.assetIndex = 1`.
- `_pool.bptIndex = 2`.
- Before bug: `userDataAmountsIn = [0, 123, 0]`.
- The vulnerable if statement is triggered, then `userDataAmountsIn = [ 123, 0]`.

But the expected format is `[0, 123]` as the bpt amount is located after the asset amount.

Recommendation: Ensure that the `assetIndex` is the expected one when `bptIndex > 0` and `bptIndex > assetIndex`. Note that it is done in the `_withdrawFromAuraPool` function:

```
// BPT not being in the assets array, we need to adjust the index
if (_pool.bptIndex > 0 && _pool.bptIndex < assetIndex) {
    assetIndex = assetIndex - 1;
}
```

3.2.8 Drain the rewards in the protocol

Submitted by Bauer

Severity: High Risk

Context: *(No context files were provided by the reviewer)*

In the `Omnipool.deposit()` function, the protocol transfers underlying tokens from the user and finally mints LP tokens to the user.

```
// Transfer underlying token to this contract
    underlyingToken.safeTransferFrom(msg.sender, address(this), _amountIn);

// ...
uint256 underlyingBalanceIncrease = afterTotalUnderlying - beforeTotalUnderlying;
    uint256 mintableUnderlyingAmount = _min(_amountIn, underlyingBalanceIncrease);
    uint256 lpReceived = mintableUnderlyingAmount.divDown(exchangeRate);
    require(lpReceived >= _minLpReceived, "too much slippage");

    lpToken.mint(_depositFor, lpReceived);
```

When a user calls `RewardManager.claimEarnings()`, the protocol first calls `omnipool.getUserTotalDeposit()` to retrieve the total deposit made by the user:

```
function _updateUserState(address _account) internal {
    // Get the user LP balance
    uint256 deposited = omnipool.getUserTotalDeposit(_account);

    // Update the pool state, claim the rewards and transfer them to the RewardManager
    _updateOmnipoolState();

    // Update the user's rewards
    _updateRewards(_account, deposited);
}
```

In the `getUserDeposit()` function, the protocol retrieves the quantity of LP tokens held by the user in the Balancer pool, then multiplies it by the price of LP tokens to determine the user's total deposit. Subsequently, the user's rewards are calculated based on this deposit.

```
function getUserDeposit(address user, uint256 poolId) public view returns (uint256) {
    UnderlyingPool memory pool = underlyingPools[poolId];
    uint256 bptBalance = IBalancerPool(pool.poolAddress).balanceOf(user);
    uint256 valuation = computeBptValution(poolId);
    return valuation * bptBalance;
}
```

The issue here is that any user who holds LP tokens from the aura pool, regardless of whether they deposited via `Omnipool.deposit()`, can claim rewards. They can deplete the rewards, causing users who deposited via `Omnipool.deposit()` to be unable to claim rewards.

Impact: Malicious users can deplete the rewards in the protocol.

Recommendation: Limit rewards to users who deposit through `Omnipool.deposit()` only.

3.2.9 Users will lose rewards

Submitted by Bauer

Severity: High Risk

Context: *(No context files were provided by the reviewer)*

As shown in the code below, in the `Omnipool.withdraw()` function, if `underlyingBalanceBefore_ < underlyingToReceive_`, the protocol calls `_withdrawFromAura()` to withdraw a portion of the funds from the Aura pool.

```
if (underlyingBalanceBefore_ < underlyingToReceive_) {
    uint256 underlyingToWithdraw_ = underlyingToReceive_ - underlyingBalanceBefore_;
    _withdrawFromAura(allocatedUnderlying_, allocatedPerPool, underlyingToWithdraw_);
}
```

When calling `auraPool.withdrawAndUnwrap()`, the parameter passed for whether to claim rewards is true, indicating that the Aura pool will transfer reward tokens to the protocol.

```
// Make sure we have enough BPT to withdraw
uint256 balance = auraPool.balanceOf(address(this));
require(balance >= _bptAmountOut, "not enough balance");
auraPool.withdrawAndUnwrap(_bptAmountOut, true);
```

```
function withdrawAndUnwrap(uint256 amount, bool claim) public returns(bool){
    _withdrawAndUnwrapTo(amount, msg.sender, msg.sender);
    //get rewards too
    if(claim){
        getReward(msg.sender,true);
    }
    return true;
}
```

However, in this case, `getReward()` is called, but the protocol does not update `RewardManager._updateRewards()`. Since the RewardManager calculates rewards based on the incremental quantity of reward tokens between before and after interactions, this results in the loss of those rewards.

Impact: All users will lose rewards.

Recommendation: A recommended fix is to use `auraPool.withdrawAndUnwrap(_bptAmountOut, false)`.

3.2.10 Existing balance of underlying tokens in omnipool.sol skews target allocation

Submitted by giraffe0x

Severity: High Risk

Context: Omnipool.sol#L262

Description: Existing balance of underlying tokens in `Omnipool.sol` skews target allocation and results in an imbalanced pool after a large deposit.

In `Omnipool.sol`, `depositFor` calculates `beforeAllocatedBalance = totalUnderlying_ + underlyingToken.balanceOf(address(this));` and passes it into `_depositForAura()`.

By taking into account `underlyingToken.balanceOf(address(this))`, it skews the calculation for `_getDepositPool` which tries to correctly allocate deposits to ensure a balanced pool.

```
// Omnipool.sol

//@audit totalUnderlying_ includes actual balance of underlying in contract
function _getDepositPool(uint256 totalUnderlying_, uint256[] memory allocatedPerPool)
    internal
    view
    returns (uint256 poolIndex, uint256 maxDepositAmount)
{
    int256 depositPoolIndex = -1;
    for (uint256 i; i < allocatedPerPool.length; i++) {
        UnderlyingPool memory pool = underlyingPools[i];
        uint256 currentAlloc = allocatedPerPool[i];
        uint256 targetWeight = (pool.targetWeight);

        //@audit Additional balance of underlying in contract will inflate targetAllocation_
        uint256 targetAllocation_ = totalUnderlying_.mulDown(targetWeight);
        if (currentAlloc >= targetAllocation_) continue;
        uint256 maxBalance_ = targetAllocation_ + targetAllocation_.mulDown(_getMaxDeviation());

        //@audit which results in an inflated maxDepositAmount
        uint256 maxDepositAmount_ = maxBalance_ - currentAlloc;
        if (maxDepositAmount_ <= maxDepositAmount) continue;
        maxDepositAmount = maxDepositAmount_;
        depositPoolIndex = int256(i);
    }
    require(depositPoolIndex > -1, "error retrieving deposit pool");
    poolIndex = uint256(depositPoolIndex);
}
```

Then when a large deposit is made (i.e. when `maxDeposit < depositsRemaining`) the inflated `maxDeposit` is added as liquidity which results in imbalanced pools.

Proof of concept:

1. Current Omnipool has total underlying of 100 USDC worth of BPT tokens (deposited in Balancer). There are two pools each with target 50% weight.
2. For whatever reason, there is also 10 USDC sitting in the Omnipool contract (could be donated).
3. Alice comes along and deposits another 100 USDC into the Omnipool.
4. `totalUnderlying_ = 100 + 10 + 100 = 210`, `targetAllocation = 210 * 0.5 = 105`.
5. `currentAllocation = 50`, so `maxDeposit = 105 - 50 = 55`.
6. Alice's 100 USDC is split 55 USDC into pool1 and 45 USDC into pool2.

Run this test:

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.20;

import "src/pools/BPTOracle.sol";
import "src/pools/Omnipool.sol";
import "src/pools/OmnipoolController.sol";
import {IOmnipool} from "src/interfaces/Omnipool/IOmnipool.sol";
import {IOmnipoolController} from "src/interfaces/Omnipool/IOmnipoolController.sol";
import "src/utils/constants.sol";
import "forge-std/console.sol";
import {stdStorage, StdStorage} from "forge-std/Test.sol";
import {SetupTest} from "../test/setup.t.sol";
import {GemMinterRebalancingReward} from "src/tokenomics/GemMinterRebalancingReward.sol";
import {IBalancerVault} from "./../src/interfaces/Balancer/IBalancerVault.sol";

interface IUSDC {
    function balanceOf(address account) external view returns (uint256);
    function mint(address to, uint256 amount) external;
    function configureMinter(address minter, uint256 minterAllowedAmount) external;
    function masterMinter() external view returns (address);
    }

contract OmnipoolTest is SetupTest {
    uint256 mainnetFork;
    BPTOracle bptPrice;
    address[] pools;
    uint256 balanceTracker;
    IERC20 usdc = IERC20(0xA0b86991c6218b36c1d19D4a2e9Eb0cE3606eB48);
    address user = 0xDa9CE944a37d218c3302F6B82a094844C6ECEb17;
    address USDC_STG = 0x8bd520Bf5d59F959b25EE7b78811142dDe543134;
    address STG = 0xAf5191B0De278C7286d6C7CC6ab6BB8A73bA2Cd6;
    address USDC_DOLA = 0xb139946D2F0E71b38e2c75d03D87C5E16339d2CD;
    address TRI_POOL = 0x2d9d3e3D0655766Aa801Ae0f6dC925db2DF291A1;
    Omnipool omnipool;
    OmnipoolController controller;
    GemMinterRebalancingReward handler;
    address public eve = vm.addr(0x60);

    mapping(address => uint256) depositAmounts;
    mapping(address => uint256) stakedBalances;

    error NullAddress();
    error NotAuthorized();
    error CannotSetRewardManagerTwice();

    using stdStorage for StdStorage;

    //registry Contract
    function setUp() public override {
        mainnetFork = vm.createFork("eth", 19105677);
        vm.selectFork(mainnetFork);
        super.setUp();
        deal(address(gem), 0x1234567890123456789012345678901234561234, 1000e18);
        omnipool = new Omnipool(
            address(0xA0b86991c6218b36c1d19D4a2e9Eb0cE3606eB48), //? underlying (circle USD)
            address(0xBA12222222228d8Ba445958a75a0704d566BF2C8), //? balancerVault
            address(registryContract),
            address(0xB188b1CB84Fb0bA13cb9ee1292769F903A9feC59), //? depositWrapper
            "Opal USDC Pool",
            "opalUSDC"
```

```
);

IUSDC usdc_ = IUSDC(address(usdc));
vm.startPrank(usdc_.masterMinter());
usdc_.configureMinter(bob, type(uint256).max);
vm.startPrank(bob);
usdc_.mint(bob, 1000e6);
usdc_.mint(alice, 1000e6);
// usdc_.mint(eve, 10e6);

controller = new OmnipoolController(address(omnipool), address(registryContract));

vm.startPrank(opal);
registryContract.setContract(CONTRACT_OMNIPOOL_CONTROLLER, address(controller));
registryAccess.addRole(ROLE_OMNIPOOL_CONTROLLER, address(controller));
handler = new GemMinterRebalancingReward(address(registryContract));
registryContract.setContract(CONTRACT_GEM_MINTER_REBALANCING_REWARD, address(handler));
controller.addOmnipool(address(omnipool));

// USDC / STG
omnipool.changeUnderlyingPool(
    0,
    USDC_STG,
    0x3ff3a210e57cfe679d9ad1e9ba6453a716c56a2e0002000000000000000005d5,
    0,
    0,
    0.5e18,
    PoolType.WEIGHTED
);
controller.addRebalancingRewardHandler(
    0x8bd520Bf5d59F959b25EE7b78811142dDe543134, address(handler)
);

// DAI / USDC / USDT
omnipool.changeUnderlyingPool(
    1,
    0x2d9d3e3D0655766Aa801Ae0f6dC925db2DF291A1,
    0x79c58f70905f734641735bc61e45c19dd9ad60bc0000000000000000000004e7,
    2,
    1,
    0.5e18,
    PoolType.STABLE
);

controller.addRebalancingRewardHandler(
    0x2d9d3e3D0655766Aa801Ae0f6dC925db2DF291A1, address(handler)
);

pools.push(address(omnipool));

vm.startPrank(opal);

registryAccess.addRole(ROLE_MINT_LP_TOKEN, address(omnipool));
registryAccess.addRole(ROLE_BURN_LP_TOKEN, address(omnipool));

// USDC
oracle.addPriceFeed(
    address(0xA0b86991c6218b36c1d19D4a2e9Eb0cE3606eB48),
    address(0x8fFfFfd4AfB6115b954Bd326cbe7B4BA576818f6)
);

// STG
oracle.addPriceFeed(
    address(0xAf5191B0De278C7286d6C7CC6ab6BB8A73bA2Cd6),
    address(0x7A9f34a0Aa917D438e9b6E630067062B7F8f6f3d)
);

// DAI
oracle.addPriceFeed(
    address(0x6B175474E89094C44Da98b954EedeAC495271d0F),
    address(0xAed0c38402a5d19df6E4c03F4E2DceD6e29c1ee9)
);

// USDT
oracle.addPriceFeed(
    address(0xdAC17F958D2ee523a2206206994597C13D831ec7),
            address(0x3E7d1eAB13ad0104d2750B8863b489D65364e32D)
        );

        bptOracle.setHeartbeat(0xA0b86991c6218b36c1d19D4a2e9Eb0cE3606eB48, 15 * 86_400);
        bptOracle.setHeartbeat(0xAf5191B0De278C7286d6C7CC6ab6BB8A73bA2Cd6, 15 * 86_400);
        bptOracle.setHeartbeat(0x6B175474E89094C44Da98b954EedeAC495271d0F, 15 * 86_400);
        bptOracle.setHeartbeat(0xdAC17F958D2ee523a2206206994597C13D831ec7, 15 * 86_400);
    }

    function testBugImbalancedPool() external {
        IOmnipool pool = IOmnipool(address(omnipool));
        IERC20Metadata token = IERC20Metadata(pool.getUnderlyingToken());

        console.log("-----");
        console.log("Initial deposit to setup pool...");
        vm.startPrank(bob);
        token.approve(address(pool), type(uint256).max);
        pool.deposit(100e6, 0);
        logUnderlying(pool);

        // Do donation to increase pool balance of underlying
        usdc.transfer(address(pool), 100e6);

        console.log("-----");
        console.log("Alice deposits...");
        vm.startPrank(alice);
        token.approve(address(pool), type(uint256).max);
        pool.deposit(10e6, 0);
        // Observe that pool has become imbalanced
        logUnderlying(pool);
    }

    function logUnderlying(IOmnipool pool) internal view {
      (uint256 totalUnderlying, uint256 totalAlloc, uint256[] memory perPoolUnderlying) =
↪   pool.getTotalAndPerPoolUnderlying();
        console.log("totalUnderlying:", totalUnderlying);
        console.log("Total allocated:", totalAlloc);
        console.log("Per pool underlying:");
        for (uint256 i = 0; i < perPoolUnderlying.length; i++) {
            console.log("Pool", i, ":", perPoolUnderlying[i]);
        }
    }
}
```

Impact: Any balance of underlying in the contract, whether donated intentionally or left in the contract after a withdraw or other actions will cause this bug. As a result, all future deposits will result in an imbalanced pool, costing Opal Gem tokens to incentivize rebalancing.

Recommendation: `depositToAura()` should receive `beforeTotalUnderlying` instead of `beforeAllocatedBalance` so as to disregard any balance of underlying tokens in the contract.

3.2.11 `_exchangerate` can be manipulated, leading to inflation attack

Submitted by 0xTheBlackPanther, also found by J4X98, Chad0, golu and Victor Okafor

Severity: High Risk

Context: Omnipool.sol#L679-L688

Description: The `_exchangeRate` in the `Omnipool` can be manipulated, allowing an attacker to set an arbitrary exchange rate during the deposit process. This manipulation can make it unprofitable for other users to deposit into the pool.

Proof of concept:

1. **Hacker Sets Exchange Rate:** The attacker initiates a deposit by setting the `exchangeRate` during a deposit, for example:

   ```
   vm.startPrank(hacker);
   omnipool.deposit(2, 0);
   token.transfer(address(omnipool), 2);
   vm.stopPrank();
   ```

2. **Victims Attempt to Deposit:** Other users (victims) try to deposit into the omnipool after the exchange rate manipulation. Due to the manipulated exchange rate, victims receive zero LP tokens for their deposits. Example:

```
vm.startPrank(user);
token.approve(address(omnipool), 10 ** 18);
omnipool.deposit(10 ** 18, 0);
vm.stopPrank();
```

3. **Attacker Withdraws All Tokens:** The attacker starts to withdraws all tokens from the pool, example:

```
vm.startPrank(hacker);
omnipool.withdraw(1, 0);
```

Impact:

- The attacker can make the omnipool unprofitable for users by manipulating the exchange rate during deposits, causing victims to receive zero LP tokens.
- The attacker can then withdraw all tokens from the pool, essentially draining it of funds.

Recommendation: One of the simplest solutions is to execute a deposit immediately after the deployment of the contract. By doing so, the exchangeRate can be adjusted to a desired and controlled value, mitigating the risk of potential manipulation.

Some of the recommendations along with their pros and cons, can be found in OpenZeppelin github issue 3706.

3.2.12 Composable pools can be calculated incorrectly

Submitted by bronzepickaxe, also found by cuthalion0x

Severity: High Risk

Context: BPTOracle.sol#L197

Description: In `BPTOracle.BptPriceComposablePool`, the price of a composable pool gets calculated:

```
function BptPriceComposablePool(bytes32 poolId) public view returns (uint256) {
    (address[] memory tokens,,) = balancerVault.getPoolTokens(poolId);

    (address pool,) = balancerVault.getPool(poolId);

    uint256 length = tokens.length;

    uint256 minPrice = type(uint256).max;
    uint256 poolRate;

    for (uint256 i; i < length;) {
        if (address(tokens[i]) == pool) {
            unchecked {
                ++i;
            }
            continue;
        }

        // Get token price
        uint256 assetPrice = getUSDPrice(address(tokens[i]));

        // Get pool rate
        poolRate = IRateProvider(pool).getRate();

        uint256 actualPrice = assetPrice * poolRate / poolRate;

        minPrice = minPrice < actualPrice ? minPrice : actualPrice;

        unchecked {
            ++i;
        }
    }

    uint256 priceResult = minPrice * poolRate;

    return priceResult / 1e18;
}
```

This functions makes use of the chainlink oracle by calling `BPTOracle.getUSDPrice()`. It also makes use of the `getRate()` function. However, the way this is calculated will not work an every composable price pool and can lead to price descrepancies, with all its consequences.

Proof of Concept: Let's look at a `wstETH` balancer pool, for example, `wstETH/aETHc`.

At the time of writing, the lowest price of these assets is: aETHc @ 3217.33

Now, we would need to call the `getRate()` from the pool. This shows the following value: `1015874971542880280`, which is approx 1.016

Now, let's do the calculation:

```
uint256 priceResult = minPrice * poolRate;
return priceResult / 1e18;
```

Assume all decimals are `1e18`. `(3217 * 1.015) = 3265.255`

This is the price that will be returned to the user. However, when we look at the actual price of the LP token of that pool, we will see that it is currently 2922.45

Which means that the current LP token is around 11.7% overvalued. This leads to users that get routed into depositing in this pool will lose 11.7% of their funds instantly due to the overvaluation, moreofer, the TVL will not be correctly displayed, which means this will affect other pools as well. All in all, this impacts the whole protocol.

Recommendation: Adjust the way of calculating composable pools.

3.2.13 Error in totalvotes formula reflects wrong number of votes

Submitted by innertia

Severity: High Risk

Context: GaugeController.sol#L331

Description: In `_getTotal()`, `votesTotal` is calculated by the formula: `votesTotal += typeVoteChange * typeWeight;` and is assigned to `totalVotes[timestamp] = votesTotal;`. However, `typeVoteChange` only reflects the amount of change in the timestamp, and when added together, it differs from the actual `totalVotes[timestamp]`. This is because it ignores the values accumulated up to that point.

The coded proof of concept consists in adding the following test to the `GaugeControllerVoteForGaugeWeightTest` contract:

```
function test_TotalVotesCalculationFormulaError() external {

    uint256 voteWeight = 1000;

    //Set the time ahead of the setUp() time
    uint256 toWarp = block.timestamp + WEEK * 1;
    uint256 nextPeriodAfterWarp = ((toWarp + WEEK) / WEEK) * WEEK;

    //Pre-voting state. Both are zero.
    assertEq(
        gaugeController.typeVotes(gaugeType, nextPeriodAfterWarp),
        gaugeController.totalVotes(nextPeriodAfterWarp)
    );

    vm.warp(toWarp);
    vm.prank(alice);
    gaugeController.voteForGaugeWeight(gauge, voteWeight);

    //Confirming the success of the vote
    assertEq(gaugeController.lastUserVote(alice, gauge), block.timestamp);

    //Since typeVotes rises but totalVotes does not, the bottom is an error. Essentially, they must be the
↪   same.
    //emit log(val: "Error: a == b not satisfied [uint]")
    //emit log_named_uint(key: " Left", val: 4250000000000000000000000000000000000 [4.25e37])
    //emit log_named_uint(key: " Right", val: 0)

    assertEq(
        gaugeController.typeVotes(gaugeType, nextPeriodAfterWarp)
        * gaugeController.typeWeights(gaugeType, nextPeriodAfterWarp),
        gaugeController.totalVotes(nextPeriodAfterWarp)
    );

}
```

Thus, `typeVotes` and `totalVotes` do not match. In this example, `typeVotes * typeWeight(1e18)` and `totalVotes` must match because there is only one type.

The `totalVotes` is the most important value in the voting system because it is the only total after the weights (`typeWeight`) are multiplied. The `typeVotes` and `gaugeVotes` are before the coefficient (`typeWeight`) is multiplied, and therefore cannot provide accurate information even if referenced. Therefore, it must be calculated accurately.

Recommendation: Should refer to the `typeVotes` themselves, not the amount of change at timestamp:

```
votesTotal += typeVotes[k][timestamp] * typeWeight;
```

3.2.14 Error due to skipped calculation of total number of votes

Submitted by innertia, also found by giraffe0x

Severity: High Risk

Context: GaugeController.sol#L320

Description: In `_getTotal()`, `uint256 timestamp = lastUpdate` is calculated, and if this is `timestamp > block.timestamp;`, the calculation of the `votesTotal` is skipped and the `votesTotal` returns an initial value of 0. This means that if `lastUpdate` already points to `timestamp > block.timestamp` by some process, the result of subsequent processing will not be reflected and `totalVotes` will return an incorrect value.

The coded proof of concept consists in adding the following test to the `GaugeControllerVoteForGaugeWeightTest` contract:

```
function test_InvalidTotalVotesByOverriding() external {

    //addType
    string memory name = "Type 1";
    uint256 weight = uint256(1);
    vm.warp(WEEK * 10);
    vm.prank(opalTeam);
    gaugeController.addType(name, weight);

    //add two Gauge
    int128 gaugeType = 0;
    uint256 gaugeWeight = uint256(1e18);
    address gauge = address(0x123);
    uint256 nextTimestamp = ((block.timestamp + WEEK) / WEEK) * WEEK;
    vm.prank(opalTeam);
    gaugeController.addGauge(gauge, gaugeType, gaugeWeight);

    assertEq(gaugeController.numberGauges(), 1);
    assertEq(gaugeController.gaugeVotes(gauge, nextTimestamp), gaugeWeight);
    assertEq(gaugeController.typeVotes(gaugeType, nextTimestamp), gaugeWeight);
    assertEq(
        gaugeController.totalVotes(nextTimestamp),
        gaugeController.typeVotes(gaugeType, nextTimestamp)
        * gaugeController.typeWeights(gaugeType, nextTimestamp)
    );

    address gauge2 = address(0x456);
    uint256 gaugeWeight2 = uint256(2e18);
    vm.prank(opalTeam);
    gaugeController.addGauge(gauge2, gaugeType, gaugeWeight2);

    assertEq(gaugeController.numberGauges(), 2);
    assertEq(gaugeController.gaugeVotes(gauge, nextTimestamp), gaugeWeight);
    assertEq(gaugeController.gaugeVotes(gauge2, nextTimestamp), gaugeWeight2);
    assertEq(gaugeController.typeVotes(gaugeType, nextTimestamp), gaugeWeight + gaugeWeight2);
    //totalVotes should be (gaugeWeight + gaugeWeight2) * uint256(1)(typeWeight), but here is gaugeWeight2 *
↪   uint256(1)(typeWeight).
    //emit log(val: "Error: a == b not satisfied [uint]")
        //emit log_named_uint(key: "      Left", val: 2000000000000000000 [2e18])
        //emit log_named_uint(key: "     Right", val: 3000000000000000000
    assertEq(
        gaugeController.totalVotes(nextTimestamp),
        gaugeController.typeVotes(gaugeType, nextTimestamp)
        * gaugeController.typeWeights(gaugeType, nextTimestamp)
    );
}
```

The `typeVotes` and `gaugeVotes` appear to be updated by the same mechanism, but these are adjusted to inherit the previous values. In contrast, `totalVotes` is seen as such a vulnerability because of its mechanism of adding from zero each time.

The `totalVotes` is the most important value in the voting system because it is the only total after the weights (`typeWeight`) are multiplied.

he `typeVotes` and `gaugeVotes` are before the coefficient (`typeWeight`) is multiplied, and therefore cannot provide accurate information even if referenced. Therefore, it must be calculated accurately.

Recommendation: The initial values can be as follows:

```
uint256 votesTotal = totalVotes[timestamp];
```

3.2.15 Improper totalvotes calculation by referring to `numbergauges` instead of `gaugetypes`

Submitted by innertia

Severity: High Risk

Context: GaugeController.sol#L327

Description: In `_getTotal()`, `numberGauges` (number of gauges) is referenced as a count of the number of loops. However, any variables used in the loop depend on the number of gauge types. There is no guarantee that the number of gauge types and the number of gauges will match, and this will cause errors in the calculation results.

For example, consider the following cases:

- Suppose there are four `gaugeTypes`, each of which is assigned the numbers 0 to 3.
- Then, there are three `gauges`, and the following types are assigned to each of them: `gauge1(type:0)`, `gauge2(type:2)`, `gauge3(type:3)`.
- Since `numberGauges=3`, the number of loops is three.
- In the calculation of the `totalVotes`, types 0, 1, and 2 can be calculated appropriately. However, the loop ends here, so we cannot get to the calculation of type 3.

Therefore, the number of votes for `gauge3` is ignored.

The `totalVotes` is the most important value in the voting system because it is the only total after the weights (`typeWeight`) are multiplied.

he `typeVotes` and `gaugeVotes` are before the coefficient (`typeWeight`) is multiplied, and therefore cannot provide accurate information even if referenced. Therefore, it must be calculated accurately.

Also, once added, a `gaugeType` or `gauge` cannot be deleted, so once this situation occurs, it becomes a serious problem.

Recommendation: Loop by reference to `numberGaugeTypes`, not `numberGauges`.

```
int128 _numberGaugeTypes = numberGaugeTypes;
```

3.2.16 Funds might get stuck in the pool due to `totaldeposited` underflow

Submitted by 0xa5df

Severity: High Risk

Context: *(No context files were provided by the reviewer)*

Description: The `totalDeposited` variable tracks the amount that gets deposited and withdrawn from the Omnipool. However, the amount withdrawn can be greater than the amount deposited. Consider the following scenario:

- The protocol deposits USDC into the USDC-WETH pool.
- The price of WETH increases.
- As a result, we get more USDC when exiting the pool.
- If users try to withdraw the function at some point it'll revert due to the underflow.
- Once users realize that, they might all try to withdraw from the pool (in order to not be the last ones), the last ones will lose their funds.

Recommendation: Either discard the `totalDeposited`, or add a check and set it to zero if it's about to underflow.

3.2.17 Improper accounting of `totaldeposited` resulting in corrupted pool state in `rewardsmanager`

Submitted by Sujith Somraaj, also found by Ch301

Severity: High Risk

Context: Omnipool.sol#L359

Description: The state variable `totalDeposited` in `OmniPool` tracks the total amount of underlying deposited to the pool. Its value is updated during every successful deposit and withdrawal.

However, during withdrawal, there is an error in accounting where the fees claimed by the protocol are not adjusted into the `totalDeposited` variable, leading to a corrupted state in the `Omnipool` contract. This corruption of state does not affect the `Omnipool` contract, but it does affect the `RewardsManager` contract.

In the `RewardsManager` contract, the `totalDeposited` variable is used in the `_updateOmnipoolState()` function to update a few critical variables required for reward calculation. It might lead to protocol insolvency (a high-impact issue).

Proof of concept:

```
function testDepositAndWithdraw() public {
    /// user deposit
    _testDeposit(pools[0]);
    /// user withdraws
    _testWithdraw(pools[0]);

    console.log(Omnipool(pools[0]).getTotalDeposited());
}
```

```
 9958236627 - withdrawn with fees
   49791183 - fees
10000000000 - totalDeposited
   91554556
```

By running the tests, it can be noticed that the actual deposited value after the withdrawal should be 41763373; instead, it is 91554556 because it fails to deduct the fees from the total deposited value.

Recommendation: Fix the totalDeposited accounting as per the recommendation below,

```
  function withdraw(uint256 _amountOut, uint256 _minUnderlyingReceived) external override {
    ...
+    totalDeposited -= underlyingWithdrawn_;
    uint256 underlyingFees = underlyingWithdrawn_ * WITHDRAW_FEES / SCALED_ONE;
    underlyingWithdrawn_ -= underlyingFees;
    ...
-    totalDeposited -= underlyingWithdrawn_;
    underlyingToken.safeTransfer(opalTreasury, underlyingFees);
    underlyingToken.safeTransfer(msg.sender, underlyingWithdrawn_);
  }
```

3.2.18 Pool weight calculation in `computepoolweights`/`computepoolweight` return wrong answer because wrong `gettotalunderlying` usage

Submitted by pks271, also found by kustrun

Severity: High Risk

Context: OmnipoolController.sol#L314

Description: `currentPool.getTotalUnderlying()` returns total number of underlying balancer pools, which should be the pools usd values instead of underlying balancer pools's length.

Recommendation: Change the code to returns total values of underlying balancer pools.

3.2.19 Pool weight calculation in `computepoolweights`/`computepoolweight` always return wrong answer because `totalusdvalue` returns 0

Submitted by pks271, also found by 8olidity and kustrun

Severity: High Risk

Context: OmnipoolController.sol#L313

Description: When calculating the pool's weight, it first gets the pool's `underlyingToken` by calling `PriceFeed.getUSDPrice`:

```
function getUSDPrice(address token) public view returns (uint256) {
    IOracle priceFeed = _priceFeedMapping[token];
    uint256 decimals = IERC20Metadata(token).decimals();
    if (address(priceFeed) == address(0)) revert PriceFeedNotFound();
    (, int256 price,,,) = priceFeed.latestRoundData();
    return uint256(price / int256(10 ** decimals));
}
```

The return data is always much smaller than the actual price. Then `currentPool.getTotalUnderlying()` returns the total number of underlying balancer pools, and `convertScale` will format the result with `decimals`. `ScaledMath.mulDown` is equal to `a * b / 1e18`.

For example, `underlying token` is USDC and the decimal is `6`, `currentPool.getTotalUnderlying()` returns 3, and the `price` = (ChainLink USDC price / `1e6`) = `1e8` / `1e6` = `100` (the ChainLink return price decimals is `1e8`), so `usdValue = 3 * 1e12 * 100 / 1e18 == 0`, the `totalUSDValue` is always return `0`, so `poolWeight` will always return `ScaledMath.ONE / pools.length` which is obviously a wrong answer.

Recommendation: Use `bptOracle.getUSDPrice` instead of `priceFeed.getUSDPrice`.

3.2.20 Wrong accounting enables attackers to prevent users from withdrawing their funds

Submitted by 0xadrii, also found by Chad0

Severity: High Risk

Context: Omnipool.sol#L377

Description: One of the key features of Opal is liquidity rebalancing, which enables Opal's funds to be distributed among several Aura pools based on a set of preconfigured weights for each pool.

When depositing into Opal, the protocol will seek to deploy funds so that weights are kept in the most possible balanced manner, depositing into the Aura pools where the difference between the currently allocated funds and the expected allocated funds is bigger.

Withdrawals are performed in the same manner, where Opal seeks to withdraw from the pools that will remain the least imbalanced after performing the withdrawal.

This particular vulnerability focuses in the way withdrawals are performed and calculated. When a withdrawal takes place, the `_withdrawFromAura()` function is called, which will select several pools to withdraw from until the requested `withdrawalsRemaining` amount is reached. An important parameter in this function is `totalUnderlying_`, which represents the total amount of underlying held in the pool (both the amount of underlying considering funds deployed in Aura pools, as well as the idle amount of underlying held in the Omnipool). `totalUnderlying_` is computed inside the `withdraw()` function, utilizing the `_getTotalAndPerPoolUnderlying()` helper:

```
// Omnipool.sol
function withdraw(uint256 _amountOut, uint256 _minUnderlyingReceived) external override {
    // ...

    uint256 underlyingBalanceBefore_ = underlyingToken.balanceOf(address(this));

    (uint256 totalUnderlying_, uint256 allocatedUnderlying_, uint256[] memory allocatedPerPool)
    = _getTotalAndPerPoolUnderlying(underlyingPrice);

    uint256 underlyingToReceive_ = _amountOut.mulDown(_exchangeRate(totalUnderlying_));

    if (underlyingBalanceBefore_ < underlyingToReceive_) {
        uint256 underlyingToWithdraw_ = underlyingToReceive_ - underlyingBalanceBefore_;
        _withdrawFromAura(allocatedUnderlying_, allocatedPerPool, underlyingToWithdraw_);
    }

    //...
}
```

The `totalUnderlying_` parameter in `_withdrawFromAura()` will be set to the `allocatedUnderlying_` (computed and returned from `_getTotalAndPerPoolUnderlying()`), which is simply the total deployed amount in underlying plus the idle amount of underlying (the idle amount of underlying being computed with a regular query to the underlying's `balanceOf` function). The important fact here is that the idle amount of underlying is considered when passing the `totalUnderlying_` parameter to `_withdrawFromAura()`:

```
// Omnipool.sol
function _withdrawFromAura(
    uint256 totalUnderlying_,
    uint256[] memory allocatedPerPool_,
    uint256 underlyingToWithdraw_
) internal {
    uint256 withdrawalsRemaining = underlyingToWithdraw_;

    uint256 totalAfterWithdrawal = totalUnderlying_ - underlyingToWithdraw_;
    uint256[] memory allocatedPerPoolCopy = allocatedPerPool_.copy();

    while (withdrawalsRemaining > 0) {
        (uint256 poolIndex, uint256 maxWithdrawal) =
            _getWithdrawPool(totalAfterWithdrawal, allocatedPerPoolCopy);
        // account for rounding errors
        if (withdrawalsRemaining < maxWithdrawal + 1e2) {
            maxWithdrawal = withdrawalsRemaining;
        }

        UnderlyingPool memory auraPool = underlyingPools[poolIndex];

        uint256 underlyingToWithdraw = _min(withdrawalsRemaining, maxWithdrawal);
        _withdrawFromAuraPool(auraPool, underlyingToWithdraw);

        withdrawalsRemaining -= underlyingToWithdraw;
        allocatedPerPoolCopy[poolIndex] -= underlyingToWithdraw;
    }
}
```

Inside `_withdrawFromAura()`, the pool selection is obtained from the `_getWithdrawPool()`, a vital function that will decide which pool to withdraw from. The way `_getWithdrawPool()` works is by predicting how the future pool's balances would be if the withdrawal was performed on each of the different available pools. Breaking down the process, the following steps take place to determine pool selection:

1. Obtain the current allocation of the pool.
2. Calculate the target allocation given the pool's `targetWeight` (the percentage from ALL funds in the protocol the pool should have) and the `totalUnderlying_`.
3. If the target allocation is greater than the current allocation it means no funds will be withdrawn from the pool because the pool already has a balance deficit. Otherwise, the pool will be selected and the amount to be withdrawn from the pool will be computed as the difference between the current allocation and the target allocation (`currentAlloc - minBalance_`). This computation simply shows the difference between the amount that is currently allocated in the pool, and the amount that should be allocated in it considering the total underlying funds and the withdrawn amount.

It is also important to highlight the fact that the transaction will revert if no pool is selected (i.e `withdrawPoolIndex` remains as -1):

```
// Omnipool.sol
function _getWithdrawPool(uint256 totalUnderlying_, uint256[] memory allocatedPerPool)
    internal
    view
    returns (uint256 poolIndex, uint256 maxWithdrawAmount)
{
    int256 withdrawPoolIndex = -1;
    for (uint256 i; i < allocatedPerPool.length; i++) {
        UnderlyingPool memory pool = underlyingPools[i];
        uint256 currentAlloc = allocatedPerPool[i];

        uint256 targetWeight = pool.targetWeight;

                    ...

        uint256 targetAllocation_ = totalUnderlying_.mulDown(targetWeight);

        if (currentAlloc <= targetAllocation_) continue; // @audit-issue [MEDIUM] - It is possible to DoS
↪   withdrawals in some situations by donating underlying tokens to the pool
        uint256 minBalance_ = targetAllocation_ - targetAllocation_.mulDown(_getMaxDeviation());
        uint256 maxWithdrawAmount_ = currentAlloc - minBalance_;
        if (maxWithdrawAmount_ <= maxWithdrawAmount) continue;
        maxWithdrawAmount = maxWithdrawAmount_;
        withdrawPoolIndex = int256(i);
    }
    require(withdrawPoolIndex > -1, "error retrieving withdraw pool");
    poolIndex = uint256(withdrawPoolIndex);
}
```

As mentioned before, the problem lies in the fact that the idle amount of underlying tokens is considered when performing withdrawals. An attacker can take advantage of this and send an amount of tokens to the pool in order to increase the `totalUnderlying_`.

Because the target allocation calculations are performed considering both the deployed + the idle amount of underlying, it will be impossible to fully withdraw the requested amount from the pools. This occurs because the idle liquidity considered in `totalUnderlying_` makes the `targetAllocation` computed for each pool always be greater than it should be in reality.

What will happen is that the amount requested to be withdrawn (the `maxWithdrawAmount_` computed in `_getWithdrawPool()`) will then be lower than it should. Eventually, all the pools will have been selected to withdraw liquidity from them. However, the `withdrawalsRemaining` will still be greater than 0 because the pools haven't allowed to withdraw enough funds.

In the final call to `_getWithdrawPool()`, all the pool's `currentAlloc` will be equal to `targetAllocation_`, so the `if (currentAlloc <= targetAllocation_) continue;` condition will hold true and all pools will be skipped, making the `withdrawPoolIndex` remain as -1 and eventually throwing the *"error retrieving withdraw pool"* error inside `_getWithdrawPool()`.

Impact: An improper accounting enables any attacker to prevent users from withdrawing by sending a minimal amount of underlying

Proof of concept: The following proof of concept shows how an attacker is able to break withdrawals by sending an amount as little as 1 USDC to the omnipool. `user` then tries to withdraw both their full balance of omnipool LP tokens, as well as a small amount of 2 USDC.

In order to reproduce the proof of concept, create a `Poc.t.sol` file in the test folder from Opal's project. Then, paste the following code inside it:

```
// SPDX-License-Identifier: MIT

pragma solidity ^0.8.20;

import "src/pools/BPTOracle.sol";
import "src/pools/Omnipool.sol";
import "src/pools/OmnipoolController.sol";
import {IOmnipool} from "src/interfaces/Omnipool/IOmnipool.sol";
import {IOmnipoolController} from "src/interfaces/Omnipool/IOmnipoolController.sol";
import "src/utils/constants.sol";
import "forge-std/console.sol";
import "forge-std/StdStorage.sol";
import {SetupTest} from "../test/setup.t.sol";
import {GemMinterRebalancingReward} from "src/tokenomics/GemMinterRebalancingReward.sol";

contract PocTest is SetupTest {
    using stdStorage for StdStorage;

    ////////////////////////////////////////////////////////////////
    //                          ERRORS                            //
    ////////////////////////////////////////////////////////////////
    error NullAddress();
    error NotAuthorized();
    error CannotSetRewardManagerTwice();

    ////////////////////////////////////////////////////////////////
    //                          STORAGE                           //
    ////////////////////////////////////////////////////////////////
    address[] pools;
    IERC20 usdc = IERC20(0xA0b86991c6218b36c1d19D4a2e9Eb0cE3606eB48);
    Omnipool omnipool;
    OmnipoolController controller;
    GemMinterRebalancingReward handler;
    IERC20Metadata token;
    address user;
    address user2;
    address attacker;


    ////////////////////////////////////////////////////////////////
    //                          SETUP                             //
    ////////////////////////////////////////////////////////////////
    function setUp() public override {
        vm.createSelectFork("eth");
        super.setUp();
        deal(address(gem), 0x1234567890123456789012345678901234561234, 1000e18);
        omnipool = new Omnipool(
            address(0xA0b86991c6218b36c1d19D4a2e9Eb0cE3606eB48),
            address(0xBA12222222228d8Ba445958a75a0704d566BF2C8),
            address(registryContract),
            address(0xB188b1CB84Fb0bA13cb9ee1292769F903A9feC59),
            "Opal USDC Pool",
            "opalUSDC"
        );

        console.log("Omnipool address: %s", address(omnipool));

        controller = new OmnipoolController(address(omnipool), address(registryContract));

        vm.startPrank(opal);
        registryContract.setContract(CONTRACT_OMNIPOOL_CONTROLLER, address(controller));
        registryAccess.addRole(ROLE_OMNIPOOL_CONTROLLER, address(controller));
        handler = new GemMinterRebalancingReward(address(registryContract));
        registryContract.setContract(CONTRACT_GEM_MINTER_REBALANCING_REWARD, address(handler));
        controller.addOmnipool(address(omnipool));

        // For rebalancing rewards
        gem.approve(address(handler), type(uint256).max);

        controller.addRebalancingRewardHandler(address(omnipool), address(handler));

        // USDC / STG
        omnipool.changeUnderlyingPool(
            0,
            0x8bd520Bf5d59F959b25EE7b78811142dDe543134,
            0x3ff3a210e57cfe679d9ad1e9ba6453a716c56a2e0002000000000000000005d5,
            0,
            0,
            0.4e18,
            PoolType.WEIGHTED
        );
        controller.addRebalancingRewardHandler(
            0x8bd520Bf5d59F959b25EE7b78811142dDe543134, address(handler)
        );

        // DAI / USDC / USDT
        omnipool.changeUnderlyingPool(
    1,
    0x2d9d3e3D0655766Aa801Ae0f6dC925db2DF291A1,
    0x79c58f70905f734641735bc61e45c19dd9ad60bc0000000000000000000004e7,
    2,
    1,
    0.3e18,
    PoolType.STABLE
);

controller.addRebalancingRewardHandler(
    0x2d9d3e3D0655766Aa801Ae0f6dC925db2DF291A1, address(handler)
);

// USDC / DOLA
omnipool.changeUnderlyingPool(
    2,
    0xb139946D2F0E71b38e2c75d03D87C5E16339d2CD,
    0xff4ce5aaab5a627bf82f4a571ab1ce94aa365ea6000200000000000000000426,
    1,
    0,
    0.3e18,
    PoolType.STABLE
);

controller.addRebalancingRewardHandler(
    0xb139946D2F0E71b38e2c75d03D87C5E16339d2CD, address(handler)
);

pools.push(address(omnipool));

vm.startPrank(opal);

registryAccess.addRole(ROLE_MINT_LP_TOKEN, address(omnipool));
registryAccess.addRole(ROLE_BURN_LP_TOKEN, address(omnipool));

// USDC
oracle.addPriceFeed(
    address(0xA0b86991c6218b36c1d19D4a2e9Eb0cE3606eB48),
    address(0x8fFfFfd4AfB6115b954Bd326cbe7B4BA576818f6)
);

// STG
oracle.addPriceFeed(
    address(0xAf5191B0De278C7286d6C7CC6ab6BB8A73bA2Cd6),
    address(0x7A9f34a0Aa917D438e9b6E630067062B7F8f6f3d)
);

// DAI
oracle.addPriceFeed(
    address(0x6B175474E89094C44Da98b954EedeAC495271d0F),
    address(0xAed0c38402a5d19df6E4c03F4E2DceD6e29c1ee9)
);

// USDT
oracle.addPriceFeed(
    address(0xdAC17F958D2ee523a2206206994597C13D831ec7),
    address(0x3E7d1eAB13ad0104d2750B8863b489D65364e32D)
);

// DOLA
oracle.addPriceFeed(
    address(0x865377367054516e17014CcdED1e7d814EDC9ce4),
    address(0x3E7d1eAB13ad0104d2750B8863b489D65364e32D)
);

// WETH
oracle.addPriceFeed(
    address(0xC02aaA39b223FE8D0A0e5C4F27eAD9083C756Cc2),
    address(0x5f4eC3Df9cbd43714FE2740f5E3616155c5b8419)
);

bptOracle.setHeartbeat(0xA0b86991c6218b36c1d19D4a2e9Eb0cE3606eB48, 15 * 86_400);
bptOracle.setHeartbeat(0xAf5191B0De278C7286d6C7CC6ab6BB8A73bA2Cd6, 15 * 86_400);
bptOracle.setHeartbeat(0x6B175474E89094C44Da98b954EedeAC495271d0F, 15 * 86_400);
bptOracle.setHeartbeat(0xdAC17F958D2ee523a2206206994597C13D831ec7, 15 * 86_400);
bptOracle.setHeartbeat(0x865377367054516e17014CcdED1e7d814EDC9ce4, 15 * 86_400);
```

```
        bptOracle.setHeartbeat(0xC02aaA39b223FE8D0A0e5C4F27eAD9083C756Cc2, 15 * 86_400);

        // Create users
        user     =  makeAddr("user");
        user2    = makeAddr("user2");
        attacker = makeAddr("attacker");

        // Deal tokens
        token = IERC20Metadata(omnipool.getUnderlyingToken());
        _dealToken(user, address(token), 100_000 * 10 ** token.decimals());
        _dealToken(user2, address(token), 1000 * 10 ** token.decimals());
        _dealToken(attacker, address(token), 1000 * 10 ** token.decimals());

        vm.startPrank(address(user));

    }

    ////////////////////////////////////////////////////////////////
    //                           POC                              //
    ////////////////////////////////////////////////////////////////
    // Pool with ID 0 -> USDC / STG with 40% weight
    // Pool with ID 1 -> DAI / USDC / USDT with 30% weight
    // Pool with ID 2 -> USDC / DOLA with 30% weight
    function testVuln_wrongAccountingLeadsToDoS() public {

        // Step 1. Perform deposit of 50 USDC with user 1 and 50 USDC with user 2 in the Omnipool
        uint256 depositAmount = 50 * 10 ** token.decimals();

        vm.startPrank(user);
        token.approve(address(omnipool), type(uint256).max);
        omnipool.deposit(depositAmount, 0);

        vm.startPrank(user2);
        token.approve(address(omnipool), type(uint256).max);
        omnipool.deposit(depositAmount, 0);

        // Step 2. Increase pool underlying token amount by directly tranferring tokens
        vm.startPrank(attacker);
        token.transfer(address(omnipool), 1 * 10**token.decimals());

        // Step 3. User tries to withdraw. This will always fail because the amount to withdraw (any amount as
↪   long as it is smaller than the idle
        // balance of underlying in the pool) can never be fully withdrawn becuase the pools need to keep some
↪   amount to account for the inflated tokens.
        // This prevents users from fully withdrawing all the requested amount. When the pools are finally
↪   balanced, there's still some amount that still
        // needs to be withdrawn but can't because of the pools needing to account for the inflation.
        vm.roll(block.number + 100); // roll 100 blocks forward
        vm.startPrank(user);
        uint256 withdrawAmount = IERC20(address(omnipool.lpToken())).balanceOf(user);

            // Trying to withdraw the full balance of LP tokens reverts.
        vm.expectRevert("error retrieving withdraw pool");
        omnipool.withdraw(withdrawAmount, 0); // withdrawAmount

        // An amount greater than the idle balance (1 USDC) also fails because it is as well needed to
↪   withdraw from the pools and the issue persists.
        withdrawAmount = 2 * 10 ** token.decimals();
        vm.expectRevert("error retrieving withdraw pool");
        omnipool.withdraw(withdrawAmount, 0); // withdrawAmount

    }

        /// @notice Internal helper function to deal tokens (forge threw a weird issue with `deal`, so this is
↪   a workaround)
    function _dealToken(address who, address _token, uint256 amt) internal {
        vm.startPrank(0xD6153F5af5679a75cC85D8974463545181f48772); // mainnet USDC Whale
        IERC20(_token).transfer(who, amt);
        vm.stopPrank();
    }

}
```

Finally, reproduce the poc by executing the following command: `forge test --mt testVuln_wrongAccountingLeadsToDoS`.

Recommendation: Consider not accounting for the idle balance held in the pool. If some idle balance is found in the omnipool, a recovery function could be added to obtain the difference between the deposited funds and the idle funds, preventing excess of underlying from being stuck forever in the contract.

3.2.21 Typevotes update error causes vote counts to be off forever

Submitted by innertia

Severity: High Risk

Context: GaugeController.sol#L452

Description: In `changeGaugeWeight`, `typeWeights[gaugeType][nextTimestamp] = newSum;` is assigned. However, `typeWeights` is a coefficient that is multiplied by vote, and it is `typevotes` to which `newSum` should be assigned instead.

Because of the unexpectedly large value of the coefficient, `totalVote` becomes highly inaccurate. Also, since `gaugeVotes` increases but `typevotes` does not, `typevotes` is permanently inaccurate (in fact, `typevotes` should be the sum of the `gaugeVotes` of the `gauges` belonging to it).

The following is a coded proof of concept (add the following test to the `GaugeControllerAddGaugeTest` contract:

```
function test_typevotesUpdateError() external {
    //In initialize(), one type is added (default)
    initialize();
    address gauge = address(0x123);
    int128 gaugeType = 0;
    uint256 gaugeWeight = uint256(1e18);
    uint256 nextTimestamp = ((block.timestamp + WEEK) / WEEK) * WEEK;
    //add gauge and check status
    vm.prank(opalTeam);
    gaugeController.addGauge(gauge, gaugeType, gaugeWeight);
    //Since both gauge and gaugeType are one, gaugeVotes and typeVotes match
    assertEq(gaugeController.gaugeVotes(gauge, nextTimestamp), gaugeWeight);
    assertEq(gaugeController.typeVotes(gaugeType, nextTimestamp), gaugeWeight);
    //typeWeights is set to uint256(1) in initialize();.
    assertEq(gaugeController.typeWeights(gaugeType, nextTimestamp), 1);

    uint256 newGaugeWeight = uint256(2e18);
    vm.prank(opalTeam);
    gaugeController.changeGaugeWeight(gauge, newGaugeWeight);
    assertEq(gaugeController.gaugeVotes(gauge, nextTimestamp), newGaugeWeight);

    //gaugeVotes and typeVotes should match, but typeVotes is still the same as before the update
    //emit log(val: "Error: a == b not satisfied [uint]")
    //emit log_named_uint(key: "      Left", val: 1000000000000000000 [1e18])
    //emit log_named_uint(key: "     Right", val: 2000000000000000000 [2e18])
    assertEq(gaugeController.typeVotes(gaugeType, nextTimestamp), newGaugeWeight);

    //Instead, the same value (2e18) as gaugeVotes has been assigned to typeWeights,
    //which has an initial value of 1 and should not be changed here.
    //emit log(val: "Error: a == b not satisfied [uint]")
    //emit log_named_uint(key: "      Left", val: 2000000000000000000 [2e18])
    //emit log_named_uint(key: "     Right", val: 1)
    assertEq(gaugeController.typeWeights(gaugeType, nextTimestamp), 1);
}
```

Recommendation: Perform the assignment to `typeVotes`, not `typeWeights`.

```
typeVotes[gaugeType][nextTimestamp] = newSum;
```

3.2.22 Error in totalvotes due to a mistake in the `totalweight` formula

Submitted by innertia, also found by giraffe0x

Severity: High Risk

Context: GaugeController.sol#L449

Description: In `_changeGaugeWeight`, the calculation of `totalVotes` is performed as follows:

- `oldSum`:

  This is the `typeVotes` (the total number of votes for the `gauges` that have been assigned the corresponding `gaugeType`).

  ```
  uint256 oldSum = _getSum(gaugeType);
  ```

- `typeWeight`:

  This is a coefficient (weight) that is to be multiplied by the `gaugeVote` or `typeVote`. One weight is set for each `gauge` (since one `gaugeType` is set for each `gauge`).

  ```
  uint256 typeWeight = _getTypeWeight(gaugeType);
  ```

- Calculated by the following formula and assigned to `totalVotes`. Here `weight` is the argument of this function:

  - GaugeController.sol#L44:

    ```
        totalWeight += (oldSum * weight) - (oldSum * typeWeight);
    ```

  - GaugeController.sol#L451

    ```
        totalVotes[nextTimestamp] = totalWeight;
    ```

    `weight` is the number of votes in the `gauge` itself, as can be seen in the following equation.

  - GaugeController.sol#L445

    ```
        gaugeVotes[gauge][nextTimestamp] = weight;
    ```

    If we look at `totalWeight` here, we find the formula `(oldSum * weight)`.

    ```
        totalWeight += (oldSum * weight) - (oldSum * typeWeight);
    ```

    `oldSum` is `typeVotes` and is the sum of the number of votes for all `gauges` belonging to that `gaugeType`. Multiplying it by the number of votes of the `gauge` is obviously wrong.

    As you can see from the subsequent subtraction, it is the `typeWeight` (the coefficient of the vote) that should be multiplied. And it is not `oldSum` that should be multiplied, but `newSum`.

  - GaugeController.sol#L448

    ```
        uint256 newSum = oldSum + (weight - oldGaugeWeight);
    ```

    This is modified to the following equation:

    ```
        totalWeight += (newSum * typeWeight) - (oldSum * typeWeight)
    ```

    This allows the function to correctly calculate the effect on `totalVotes` of a change in `gaugeWeight`, which is the purpose of this function.

Recommendation: Rewrite the expression for `totalWeight` as follows:

```
totalWeight += (newSum * typeWeight) - (oldSum * typeWeight)
```

3.3 Medium Risk

3.3.1 Chainlink's `latestrounddata` might return stale or incorrect results

Submitted by J4X98, also found by AuditorPraise, 8olidity, bronzepickaxe, Chad0, zanderbyte, Naveen Kumar Naik J - 1nc0gn170 and 0xhashiman

Severity: Medium Risk

Context: BPTOracle.sol#L242

Description: In `BPTOracle.sol`, we are using the `latestRoundData()` function, but there is no check if the return value indicates stale data. This could lead to potentially incorrect values being returned from the oracle, which would compromise the security of the protocol.

This could lead to stale prices according to the Chainlink documentation.

Recommendation: The issue can be mitigated by adding checks that will verify the correctness of the data:

```
(uint80 roundID, int256 answer, uint256 timestamp , uint256 updatedAt, uint80 answeredInRound) =
↪  priceFeed.latestRoundData();
if (updatedAt + tokenHeartbeat[token] < block.timestamp) revert StalePrice();
require(timestamp != 0,"Round not complete");
require(answeredInRound >= roundID, "Stale Price");
require(answer > 0, "Invalid Price");
```

3.3.2 Attacker can reset the gauge associated to a `lptoken`

Submitted by zigtur, also found by 8olidity, 0xTheBlackPanther, kustrun, 0x4non, crypticdefense, Naveen Kumar Naik J - 1nc0gn170 and Lalanda

Severity: Medium Risk

Context: GaugeFactory.sol#L66

Description: The `GaugeFactory` contract deploys gauge contracts by cloning the existing implementation through the `deployGauge` function. Once deployed, the `lpTokenToGauge` mapping associates the provided `lpToken` address to the new gauge.

Through calling `deployGauge`, the attacker can overwrite the existing `lpTokenToGauge[lpToken]` entry.

Impact: The `lpTokenToGauge[lpToken]` mapping is likely to be used to retrieve the Gauge address associated to an `lpToken` address.

Exploiting the vulnerability will lead the protocol to use a newly initialized gauge contract, losing every state registered in the previous gauge. This leads to loss of funds.

Proof of concept: The following test can be added to `test/GaugeFactory.t.sol`:

```
/**
    * @notice  Should overwrite the existing liquidity gauge with a new one
    */
function test_zigturExistingOverwriteGauge() external {
    vm.prank(alice);
    address gauge = gaugeFactory.deployGauge(address(0x124));

    assertEq(gaugeFactory.gaugeToLpToken(gauge), address(0x124));
    assertEq(gaugeFactory.lpTokenToGauge(address(0x124)), gauge);
    assertEq(gaugeFactory.isFactoryGauge(gauge), true);
    assertEq(gauge == address(0), false);


    address bob = vm.addr(0x07);
    vm.prank(bob);
    address gaugeBob = gaugeFactory.deployGauge(address(0x124));


    assertEq(gaugeFactory.gaugeToLpToken(gaugeBob), address(0x124));
    // Check that the mapping(lpToken => gaugeAddress) has been overwritten
    assertEq(gaugeFactory.lpTokenToGauge(address(0x124)), gaugeBob);
    assertFalse(gaugeFactory.lpTokenToGauge(address(0x124)) == gauge);
    assertEq(gaugeFactory.isFactoryGauge(gaugeBob), true);
}
```

Recommendation: Depending on the expected behaviors, there are several options to address the issue:

1. Implement access control on the `deployGauge` function if the gauge contract is used temporarly and should be renewed.
2. Ensure that `lpTokenToGauge[lpToken] == address(0)` through a require statement if only one gauge can be associated to one `lpToken`.

3.3.3 Ineffective `deadline` parameter allows swap transactions to be included at any future time

Submitted by cuthalion0x, also found by giraffe0x, qckhp, pks271 and Victor Okafor

Severity: Medium Risk

Context: Omnipool.sol#L836

Description: The Balancer `batchSwap()`, like most AMM swap methods, includes a `deadline` argument that is passed by the caller. Its intention is to prevent swap transactions from being forgotten and then included in far future blocks when economic conditions are less favorable to the caller.

It is a common anti-pattern to produce an on-chain `deadline` that depends on `block.timestamp`. This is an anti-pattern because `deadline` is meant to be an absolute timestamp passed in from off-chain. Because `block.timestamp` will always be the time that the transaction is included in the block, whether now or in the far future, it effectively represents an infinite `deadline`.

Recommendation: Allow the caller to provide an absolute `deadline` when calling `Omnipool.swapForGem()`.

3.3.4 Check for no deposit and withdrawal in the same block, also blocks double deposits

Submitted by J4X98, also found by kustrun

Severity: Medium Risk

Context: Omnipool.sol#L239, Omnipool.sol#L243, Omnipool.sol#L360

Description: The Opal protocol implements depositing and withdrawing functionalities for the pool. To ensure users do not deposit and withdraw too frequently, there needs to be at least a single block difference between the deposit and the withdrawal.

Unfortunately, the current implementation only uses the `lastTransactionBlock` to track this, and sets this variable to the block.number on each deposit/withdraw. When someone wants to deposit/withdraw again, this number is checked and the contract reverts if no blocks have passed since then.

This leads to issues in the case of a user wanting to deposit or withdraw twice in the same block as the variable will also be set, resulting in the second call reverting.

Recommendation: The issue can be mitigated by splitting the mapping into two mappings `lastDeposit` and `lastWithdrawal`. When a user tries depositing, it should be checked if the `block.number` is bigger than the one in `lastWithdrawal` and vice versa. This approach keeps the intended functionality intact while removing the blockage of double deposits/withdrawals.

3.3.5 Users will still be able to deposit into deactivated pools

Submitted by J4X98, also found by kustrun, giraffe0x, bronzepickaxe, Sujith Somraaj, 0xTheBlackPanther and Naveen Kumar Naik J - 1nc0gn170

Severity: Medium Risk

Context: Omnipool.sol#L879

Description: The `OmnipoolController` has the ability to deactivate pools. this is done by it calling the `desactivate()` function of the Omnipool:

```
function desactivate() external onlyController {
    if (isShutdown) {
        revert PoolAlreadyShutdown();
    }
    isShutdown = true;
    emit Shutdown();
}
```

Unfortunately, this function just sets the `isShutdown` bool to `true`, but the bool does not affect the functionality of the contract in any way. Users are still able to deposit and withdraw funds. One can take a look at the `Votelocker` (`src/tokenomics/Votelocker.sol`) contract to see how this functionality should work. In the `Votelocker`, all new locking is stopped once the contract is shut down.

Recommendation: When the `Omnipool` is shut down, no new deposits should be possible anymore. This can be achieved by enforcing that `isShutdown` is false at the start of `depositFor()`

```
if (isShutdown) revert ContractShutdown();
```

3.3.6 `getusdprice` will revert when access to chainlink oracle data feed is blocked

Submitted by 0xTheBlackPanther, also found by 0xRizwan

Severity: Medium Risk

Context: BPTOracle.sol#L235-L247

Description: The `getUSDPrice` function in the `BPTOracle` contract directly calls `latestRoundData()` on Chainlink price feeds. This direct call could potentially revert, leading to a Denial-of-Service (DoS) scenario, as Chainlink multisigs can block access to price feeds. To mitigate this risk, it is recommended to wrap the calls to Oracles in try/catch blocks, allowing the contract to handle errors safely and explicitly.

Impact: The impact of the current implementation is significant, as it could result in a DoS scenario for smart contracts relying on the `getUSDPrice` function. By implementing the recommended defensive approach, the contract will be better prepared to handle errors and ensure continued functionality even in the event of Oracle-related issues.

- Affected Functions

1. `BptPriceStablePool`: The `BptPriceStablePool` function utilizes the `getUSDPrice` function, and any revert in `getUSDPrice` could impact the stability calculation for stable pools.
2. `BptPriceWeightPool` and `BptPriceComposablePool`: Similarly, the `BptPriceWeightPool` and `BptPriceComposablePool` functions depend on `getUSDPrice`, making them susceptible to potential reverts.

Recommendation: Wrap Oracle Calls in try/catch Blocks: Surround the call to `latestRoundData()` with a try/catch structure in the `getUSDPrice` function. This defensive approach will prevent reverts from causing a complete DoS to smart contracts relying on price feeds.

```
function getUSDPrice(address token) public view returns (uint256 priceInUSD) {
    try IOracle(priceFeedAddress).latestRoundData() returns (, int256 priceInUSDInt,, uint256 updatedAt,) {
        // Existing code for successful oracle call
        // ...
    } catch {
        // Handle the error, e.g., call a fallback oracle or take appropriate action
        revert("Error querying Oracle");
    }
}
```

3.3.7 `Bptoracle` makes assumptions on the usd price decimals

Submitted by zigtur, also found by n1punp, Beelzebufo, gd, bronzepickaxe, 0x4non, kustrun, shaka, Lalanda and recursive

Severity: Medium Risk

Context: BPTOracle.sol#L245

Description: The `getUSDPrice` function in `BPTOracle` retrieves the USD price of an asset from the priceFeed. This function makes the assumption that the returned price is based on 8 decimals.

The function lacks a check to ensure that the returned price is based on 8 decimals.

Impact: The USD Price returned by a price feed will be handled in an incorrect way, leading to underevaluate or overevaluate an asset value. This may lead to loss of funds.

Recommendation: Consider adding a price decimals check in the `getUSDPrice` function. For example, the following `getUSDPrice` can be implemented:

```
function getUSDPrice(address token) public view returns (uint256 priceInUSD) {
    if (tokenHeartbeat[token] == 0) {
        revert HeartbeatNotSet();
    }

    IOracle priceFeed = IPriceFeed(priceFeedAddress).getPriceFeedFromAsset(token);
    if (address(priceFeed) == address(0)) revert PriceFeedNotFound();
    require(priceFeed.decimals() == 8);
    (, int256 priceInUSDInt,, uint256 updatedAt,) = priceFeed.latestRoundData();
    if (updatedAt + tokenHeartbeat[token] < block.timestamp) revert StalePrice();
    // Oracle answer are normalized to 8 decimals
    uint256 newPrice = _normalizeAmount(uint256(priceInUSDInt), 8);
    return newPrice;
}
```

3.3.8 Loss of gem token incentive for a `depositedfor` user, whenever a user deposits for another user via `depositfor()`

Submitted by AuditorPraise

Severity: Medium Risk

Context: GemMinterRebalancingReward.sol#L116, Omnipool.sol#L277

Description: whenever a user deposits for someone else via `depositFor()` in the `omniPool`, the "depositedFor" user is supposed to get the GEM Tokens as incentive whenever `rebalancingRewardActive` == true. But there's an issue in `depositedFor()` which causes the GEM Tokens to be minted to the `msg.sender` instead of the `_depositFor` (*the address of the user for whom to deposit*).

`_handleRebalancingRewards()` is called inside `depositFor()` at L276 with `msg.sender` as account instead of `_depositFor`. In `_distributeRebalancingRewards()` GEM will be transferred to the `msg.sender` instead of `_depositFor`:

```
IERC20(gem).safeTransferFrom(incentivesMs, account, amount);
```

Recommendation: When calling `_handleRebalancingRewards()` in `depositFor()` at L276 use `_depositFor` as account instead of `msg.sender`.

3.3.9 There is no enforcement of the delay when calling `updateweights` in `omnipoolcontroller.sol`

Submitted by dirtymic, also found by J4X98, zigtur, john-femi, Naveen Kumar Naik J - 1nc0gn170, bronzepickaxe and Lalanda

Severity: Medium Risk

Context: *(No context files were provided by the reviewer)*

This allows the weights to be changed at any time. If `rebalancingRewardActive` is `true`, the allocations can be changed to game the rebalancing rewards system. Allowing deposits to gather more rewards.

```
function testRebalance() public {
    deal(
        address(gem),
        0x1234567890123456789012345678901234561234,
        type(uint256).max
    );

    vm.startPrank(0x1234567890123456789012345678901234561234);
    IERC20(gem).approve(
        registryContract.getContract(
            CONTRACT_GEM_MINTER_REBALANCING_REWARD
        ),
        type(uint256).max
    );
    uint256[] memory initWeight = omnipool.getAllUnderlyingPoolWeight();
    for (uint256 i = 0; i < initWeight.length; i++) {
        console.log("init weight: %s", initWeight[i]);
    }

    uint256 decimals = 6;
    vm.startPrank(user);
    address poolAddress = address(omnipool);
    console.log(poolAddress);
    IERC20(0xA0b86991c6218b36c1d19D4a2e9Eb0cE3606eB48).approve(
        address(omnipool),
        100_000 * 10 ** decimals
    );

    vm.stopPrank();

    vm.roll(1);
    vm.prank(user);
    omnipool.deposit(10_000 * 10 ** decimals, 1);

    skip(14 days);

    IOmnipoolController.WeightUpdate[]
        memory newWeights = new IOmnipoolController.WeightUpdate[](3);
    newWeights[0] = IOmnipoolController.WeightUpdate(TRI_POOL, 0.8e18);
    newWeights[1] = IOmnipoolController.WeightUpdate(USDC_STG, 0.2e18);
    newWeights[2] = IOmnipoolController.WeightUpdate(USDC_DOLA, 0);
    vm.prank(opal);
    controller.updateWeights(poolAddress, newWeights);

    initWeight = omnipool.getAllUnderlyingPoolWeight();
    for (uint256 i = 0; i < initWeight.length; i++) {
        console.log("init weight: %s", initWeight[i]);
    }

    skip(1 hours);

    assertTrue(omnipool.rebalancingRewardActive());

    uint256 snapshotId = vm.snapshot();

    uint256 deviationBefore = omnipool.computeTotalDeviation();
    uint256 gemBalanceBefore = IERC20(gem).balanceOf(user);
    vm.roll(2);
    vm.prank(user);
    omnipool.deposit(10_000 * 10 ** decimals, 1);
    uint256 deviationAfter = omnipool.computeTotalDeviation();
    assertLt(deviationAfter, deviationBefore);
    uint256 gemBalanceAfter = IERC20(gem).balanceOf(user);
    assertGt(gemBalanceAfter, gemBalanceBefore);
    console.log("reward user balance before: %s", gemBalanceBefore);
    console.log("reward user balance after: %s", gemBalanceAfter);

    vm.revertTo(snapshotId);

    uint256 secondDeviationBefore = omnipool.computeTotalDeviation();
    uint256 secondGemBalanceBefore = IERC20(gem).balanceOf(user);
    vm.roll(2);

    IOmnipoolController.WeightUpdate[]
        memory skipWeights = new IOmnipoolController.WeightUpdate[](3);
    skipWeights[0] = IOmnipoolController.WeightUpdate(TRI_POOL, 0.3e18);
    skipWeights[1] = IOmnipoolController.WeightUpdate(USDC_STG, 0.3e18);
    skipWeights[2] = IOmnipoolController.WeightUpdate(USDC_DOLA, 0.4e18);
    vm.prank(opal);
    controller.updateWeights(poolAddress, skipWeights);

    initWeight = omnipool.getAllUnderlyingPoolWeight();
    for (uint256 i = 0; i < initWeight.length; i++) {
        console.log("gaming weight: %s", initWeight[i]);
    }

    vm.prank(user);
    omnipool.deposit(10_000 * 10 ** decimals, 1);
    uint256 secondDeviationAfter = omnipool.computeTotalDeviation();
    assertLt(secondDeviationAfter, secondDeviationBefore);
    uint256 secondGemBalanceAfter = IERC20(gem).balanceOf(user);
    assertGt(secondGemBalanceAfter, secondGemBalanceBefore);
    assertGt(secondGemBalanceAfter, gemBalanceAfter);
    console.log("reward user balance before: %s", secondGemBalanceBefore);
    console.log("reward user balance after: %s", secondGemBalanceAfter);
    console.log("-----------------------------------------------------");
    console.log("Gamed user balance after: %s", secondGemBalanceAfter);
    console.log("Regular user balance after: %s", gemBalanceAfter);
}
```

3.3.10 `_withdrawfromaurapool()` will revert if bpt isn't much, this shouldn't be so as it still try to withdraw some bpt to make up

Submitted by AuditorPraise

Severity: Medium Risk

Context: Omnipool.sol#L551

Description: In `Omnipool._withdrawFromAuraPool()` there will always be a revert whenever the balance of the BPT to withdraw is < `_bptAmountOut`. This is wrong since `Omnipool._withdrawFromAuraPool()` still tries to withdraw some BPT to make up the expected `_bptAmountOut`.

I believe this check (`require(balance >= _bptAmountOut, "not enough balance");`) should have been after `_withdrawFromAuraPool()` tries to withdraw some BPT from the `auraPool` to make up the expected `_bptAmountOut`, that way if the BPT balance is still lower than the expected `_bptAmountOut`for whatever reason it can now revert.

Recommendation: Change the order of the code:

```
+ auraPool.withdrawAndUnwrap(_bptAmountOut, true); // @audit-ok withdraw first from auraPool
  // Make sure we have enough BPT to withdraw
  uint256 balance = auraPool.balanceOf(address(this));
  require(balance >= _bptAmountOut, "not enough balance");//@audit-issue this renders the below line of code
↪  useless
- auraPool.withdrawAndUnwrap(_bptAmountOut, true);
```

3.3.11 The return `value(bool)` of `aurapool.withdrawandunwrap()` isn't checked

Submitted by AuditorPraise, also found by 0xTheBlackPanther, ZanyBonzy, Bauer and mxuse

Severity: Medium Risk

Context: Omnipool.sol#L552

Description: `auraPool.withdrawAndUnwrap()` returns a boolean but its never checked.

```
function withdrawAndUnwrap(uint256 _amount, bool _claim) external returns (bool);
```

The returned boolean of `auraPool.withdrawAndUnwrap()` signifies if the operation was indeed successful or not hence its crucial to check it.

Recommendation: Please check the returned bool of `auraPool.withdrawAndUnwrap()` and ensure it's successful

3.3.12 Exchange rate rounding allows users to get more shares more than intended

Submitted by J4X98, also found by Bauer

Severity: Medium Risk

Context: Omnipool.sol#L687

Description: The Opal pool employs a similar system to an ERC4626 vault. It allows for depositing and withdrawing of assets to gain shares in the vault. To calculate the exchange rate between shares and the underlying assets, which are distributed in balancer vaults, the `_exchangeRate()` function is used:

```
function _exchangeRate(uint256 totalUnderlying_) internal view returns (uint256) {
    uint256 lpSupply = lpToken.totalSupply();
    if (lpSupply == 0 || totalUnderlying_ == 0) return 10 ** 18;

    return totalUnderlying_.divDown(lpSupply);
}
```

This function calculates the exchange rate (like an ERC4626 vault) based on

$$totalUnderlying/totalShares$$

When doing this calculation, the division is rounded down. This is correct in the case of a user withdrawing from the Omnipool where the calculation for the underlying that he will receive is:

$$underlyingReceived = sharesWithdrawn * exchangeRate$$

So in that case a rounded down exchange rate will guarantee that the user does not receive more underlying than he would be allowed to.

Unfortunately, the same rounding is applied when calculating the exchange rate for user deposits. The calculation in that case is:

$$sharesReceived = amountIn/exchangeRate$$

So in that case, a lower exchange rate will result in the user getting more shares than intended.

Proof of concept: The issue can be showcased in a calculation example: In this case, a user has already deposited 100 tokens a year ago and now earned a 10% yield on them. A second user deposits another 100 tokens now.

```
underlying = 110
shares = 100
amountIn = 100

exchangeRate = roundDown(110/100) = 1

sharesReceived = 100 / 1 = 100
```

So in this example, one can see that the second user will be able to steal 5 of the first users' tokens when withdrawing his shares again.

Recommendation: The `_exchangeRate()` function should be adapted so that it can be given a rounding parameter, which is used to decide how to round on the division:

```
function _exchangeRate(uint256 totalUnderlying_, bool roundUp) internal view returns (uint256) {
    uint256 lpSupply = lpToken.totalSupply();
    if (lpSupply == 0 || totalUnderlying_ == 0) return 10 ** 18;

    if (!roundUp)
        return totalUnderlying_.divDown(lpSupply);
    else
        return totalUnderlying_.divUp(lpSupply);
}
```

On deposits the calculation should round up and on withdraws it should round down.

3.3.13 Reducing the weight of a voted gaugetype will always revert

Submitted by J4X98, also found by giraffe0x

Severity: Medium Risk

Context: GaugeController.sol#L449

Description: The `GaugeController` contract gives the opal team the ability to change the weight of gauge types. This is implemented in the `_changeTypeWeight()` function:

```
function _changeTypeWeight(int128 gaugeType, uint256 weight) internal
{
    uint256 oldWeight = _getTypeWeight(gaugeType);
    uint256 oldSum = _getSum(gaugeType);
    uint256 totalWeight = _getTotal();
    uint256 nextTimestamp = ((block.timestamp + WEEK) / WEEK) * WEEK;

    totalWeight += (oldSum * weight) - (oldSum * oldWeight);
    totalVotes[nextTimestamp] = totalWeight;
    typeWeights[gaugeType][nextTimestamp] = weight;
    lastUpdate = nextTimestamp;
    lastTypeWeightUpdate[gaugeType] = nextTimestamp;

    emit NewTypeWeight(gaugeType, nextTimestamp, weight, totalWeight);
}
```

As this contract is compiled with a new version of Solidity which includes under and overflow protection, the calculation of `totalWeight` will potentially lead to a revert. This will happen in the case where `oldWeight` is bigger than `weight`.

```
uint256 newSum = oldSum + (weight - oldGaugeWeight); <-- Overflow here
totalWeight += (oldSum * weight) - (oldSum * oldWeight); <-- Overflow here
```

Proof of concept: A simple testcase showcases the issue:

```
function test_revertNegativeChangeTypeWeight() external {
    //------------- SETUP
    // Initialize the gauge type
    uint256 weight = uint256(2e18);
    int128 gaugeType = 0;
    initialize(weight);

    // Deploy a gauge
    address gauge = address(0x123);
    vm.startPrank(opalTeam);
    gaugeController.addGauge(gauge, gaugeType, weight);
    vm.stopPrank();

    //Alice has some locks
    uint256 currentPeriod = (block.timestamp / WEEK) * WEEK;
    MockedVoteLocker.LockedBalance[] memory userLocks = new MockedVoteLocker.LockedBalance[](3);
    userLocks[0] = MockedVoteLocker.LockedBalance(uint112(200e18), uint32(currentPeriod + WEEK * 4));
    userLocks[1] = MockedVoteLocker.LockedBalance(uint112(150e18), uint32(currentPeriod + WEEK * 6));
    userLocks[2] = MockedVoteLocker.LockedBalance(uint112(75e18), uint32(currentPeriod + WEEK * 9));
    voteLocker.setUserLocks(alice, userLocks);

    // Alice votes
    vm.prank(alice);
    gaugeController.voteForGaugeWeight(gauge, 10000);

    //------------- ISSUE --------------------------------

    //The team tries to reduce the weigth of the gauge type
    uint256 newWeight = uint256(1e18);
    vm.prank(opalTeam);
    vm.expectRevert();
    gaugeController.changeTypeWeight(gaugeType, newWeight);
}
```

The test can be added to `test/GaugeController.t.sol` and run using `forge test -vvvv --match-test "test_revertNegativeChangeTypeWeight"`.

Recommendation: The issue can be mitigated by checking if `oldWeight` is bigger than `weight`. In that case, the total Weight should be reduced instead of increased.

```
if (oldWeight > weight)
{
    totalWeight -= (oldSum * oldWeight) - (oldSum * weight);
        uint256 newSum = oldSum - (oldGaugeWeight - weight);
}
else
{
    totalWeight += (oldSum * weight) - (oldSum * oldWeight);
        uint256 newSum = oldSum + (weight - oldGaugeWeight);
}
```

3.3.14 Locks unlocking at the `nexttimestamp` will not trigger `noactivelocks`

Submitted by J4X98

Severity: Medium Risk

Context: GaugeController.sol#L494

Description: The Opal protocol implements a governance mechanism that allows the users to vote on the weight of gauges. This is facilitated in the `_voteForGaugeweight()` function. This function imposes three checks on the `unlockTime` of locks:

```
if (locks[vars.len - 1].unlockTime < vars.nextTimestamp) revert NoActiveLocks();
```

This check enforces that the last lock (the one that will be locked for the longest) needs to unlock at `unlockTime >= vars.nextTimestamp`, otherwise all the locks are invalid for this voting period.

The second check is enforced later when the voting power from all locks is summed:

```
Unlocks[] memory unlocks = new Unlocks[](vars.len);
uint256 i = vars.len - 1;
IVoteLocker.LockedBalance memory currentLock = locks[i];
while (currentLock.unlockTime > vars.nextTimestamp) {
    uint256 weightedAmount = currentLock.amount * voteWeight / 10_000;
    newUserVote.amount += weightedAmount;

    unlocks[i] = Unlocks({
        amount: uint208(weightedAmount),
        unlockTime: currentLock.unlockTime
    });

    if (i > 0) {
        i--;
        currentLock = locks[i];
    } else {
        break;
    }
}
```

Now all locks where `unlockTime > vars.nextTimestamp` get summed up (are eligible to vote). The third check implements another invariant:

```
for (uint256 l; l < vars.len; l++) {
    // Also covers case were unlockTime is 0 (empty array item)
    if (unlocks[l].unlockTime <= block.timestamp) continue;

    userVoteUnlocks[user][gauge].push(unlocks[l]);
}
```

This time the invariant of who is allowed to vote is `unlocks[l].unlockTime > block.timestamp`. Here the cut off date is the `block.timestamp` instead of the `vars.nextTimestamp`.

Unfortunately, these 3 invariants contradict each other, resulting in a broken voting system.

Recommendation: The issue can be mitigated by changing the three conditionals to follow the same invariant. This recommended invariant would be `unlockTime > vars.nextTimestamp`.

3.3.15 Using `updateweight` may break the total weight assumption

Submitted by zigtur, also found by Ch301

Severity: Medium Risk

Context: Omnipool.sol#L1032

Description: The `Omnipool.updateWeight` allows to update the target weight of a single pool.

The `updateWeights` function shows that the total of all the target weights should be equal to `ScaledMath.ONE`. By using `updateWeight` this assumption may be broken, leading to inaccurate calculation in several functions of the Omnipool.

Impact: Breaking the assumption that `sum(target weights) == ScaledMath.ONE` will break Omnipool's calculations.

For example, the `_getDepositPool` calculates `targetAllocation_` by using the assumption through the `mulDown` function:

```
uint256 targetWeight = (pool.targetWeight);
uint256 targetAllocation_ = totalUnderlying_.mulDown(targetWeight);
```

Recommendation: Consider restricting weight update to the use of `updateWeights`, to ensure that the total target weights assumption holds. This can be done by deleting the `updateWeight` function.

3.3.16 Omnipool does not take into account pending balancer protocol fees

Submitted by giraffe0x

Severity: Medium Risk

Context: Omnipool.sol#L186, BPTOracle.sol#L166

Description: Omnipool does not take into account pending Balancer protocol fees, therefore over-estimating Balance Pool Token (BPT) valuation and under-minting LP tokens to users interacting with Omnipool.

When depositing or withdrawing from a Balancer pool, accrued protocol fees are first minted to Balancer (increasing total supply) before they are minted to the depositor/withdrawer. See docs and code.

During a deposit in `Omnipool.sol`, `beforeTotalUnderlying` is calculated to established the before-state of the omnipool.

The calculation for `beforeTotalUnderlying` takes how much BPT the omnipool holds, multiplied by `BPT-Valuation` which in turn is calculated by BPT TVL divided by `totalSupply`. As pending protocol fees have not been factored into this `totalSupply` (i.e. `totalSupply` should be larger), BPT valuation and `beforeTo-talUnderlying` is over-estimated.

After the deposit, `afterTotalUnderlying` is compared against `beforeTotalUnderlying` to determine how much lpTokens to mint to the depositor, who ends up getting less than desired due to the over-estimation of before state.

```
// Omnipool.sol
function computeBptValution(uint256 poolId) public view returns (uint256) {
    UnderlyingPool memory pool = underlyingPools[poolId];
    return bptOracle.getPoolValuation(pool.poolId, pool.poolType);
}

// BPTOracle.sol
function getPoolValuation(bytes32 poolId, PoolType poolType) public view returns (uint256) {
    if (poolType == PoolType.WEIGHTED) {
        return BptPriceWeightPool(poolId);
    }
}

function BptPriceWeightPool(bytes32 poolId) public view returns (uint256) {
    // ...
    // 4. Total Supply of BPT tokens for this pool
    int256 totalSupply = int256(IBalancerPool(poolAddress).totalSupply());

    // 5. BPT Price (USD) = TVL / totalSupply
    //@audit totalSupply does not factor in accrued protocol fees
    uint256 bptPrice = uint256((numerator.toInt().div(totalSupply)));
    return bptPrice;
}
```

Proof of concept: In the proof of concept below, we show how after many swaps have occured (to accrue high protocol fees), Alice a depositor into Omnipool deposits 10 USDC, withdraws after 1 block, and receives back only 9.6 USDC (4% loss).

```
// SPDX-License-Identifier: MIT

pragma solidity ^0.8.20;

import "src/pools/BPTOracle.sol";
import "src/pools/Omnipool.sol";
import "src/pools/OmnipoolController.sol";
import {IOmnipool} from "src/interfaces/Omnipool/IOmnipool.sol";
import {IOmnipoolController} from "src/interfaces/Omnipool/IOmnipoolController.sol";
import "src/utils/constants.sol";
import "forge-std/console.sol";
import {stdStorage, StdStorage} from "forge-std/Test.sol";
import {SetupTest} from "../test/setup.t.sol";
import {GemMinterRebalancingReward} from "src/tokenomics/GemMinterRebalancingReward.sol";
import {IBalancerVault} from "./../src/interfaces/Balancer/IBalancerVault.sol";

interface IUSDC {
    function balanceOf(address account) external view returns (uint256);
    function mint(address to, uint256 amount) external;
    function configureMinter(address minter, uint256 minterAllowedAmount) external;
    function masterMinter() external view returns (address);
    }

contract OmnipoolTest is SetupTest {
    uint256 mainnetFork;
    BPTOracle bptPrice;
    address[] pools;
    uint256 balanceTracker;
    IERC20 usdc = IERC20(0xA0b86991c6218b36c1d19D4a2e9Eb0cE3606eB48);
    address user = 0xDa9CE944a37d218c3302F6B82a094844C6ECEb17;
    address USDC_STG = 0x8bd520Bf5d59F959b25EE7b78811142dDe543134;
    address STG = 0xAf5191B0De278C7286d6C7CC6ab6BB8A73bA2Cd6;
    // address USDC_DOLA = 0xb139946D2F0E71b38e2c75d03D87C5E16339d2CD;
    // address TRI_POOL = 0x2d9d3e3D0655766Aa801Ae0f6dC925db2DF291A1;
    Omnipool omnipool;
    OmnipoolController controller;
    GemMinterRebalancingReward handler;
    address public eve = vm.addr(0x60);

    mapping(address => uint256) depositAmounts;
    mapping(address => uint256) stakedBalances;

    error NullAddress();
    error NotAuthorized();
    error CannotSetRewardManagerTwice();

    using stdStorage for StdStorage;

    //registry Contract
    function setUp() public override {
        mainnetFork = vm.createFork("eth", 19105677);
        vm.selectFork(mainnetFork);
        super.setUp();
        deal(address(gem), 0x1234567890123456789012345678901234561234, 1000e18);
        omnipool = new Omnipool(
            address(0xA0b86991c6218b36c1d19D4a2e9Eb0cE3606eB48), //? underlying (circle USD)
            address(0xBA12222222228d8Ba445958a75a0704d566BF2C8), //? balancerVault
            address(registryContract),
            address(0xB188b1CB84Fb0bA13cb9ee1292769F903A9feC59), //? depositWrapper
            "Opal USDC Pool",
            "opalUSDC"
        );

        IUSDC usdc_ = IUSDC(address(usdc));
        vm.startPrank(usdc_.masterMinter());
        usdc_.configureMinter(bob, type(uint256).max);
        vm.startPrank(bob);
        usdc_.mint(bob, 1_000_000e6);
        usdc_.mint(alice, 10e6);
        usdc_.mint(eve, 10e6);

        controller = new OmnipoolController(address(omnipool), address(registryContract));

        vm.startPrank(opal);
        registryContract.setContract(CONTRACT_OMNIPOOL_CONTROLLER, address(controller));
        registryAccess.addRole(ROLE_OMNIPOOL_CONTROLLER, address(controller));
        handler = new GemMinterRebalancingReward(address(registryContract));
        registryContract.setContract(CONTRACT_GEM_MINTER_REBALANCING_REWARD, address(handler));
        controller.addOmnipool(address(omnipool));

        // // For rebalancing rewards
        // gem.approve(address(handler), type(uint256).max);

        // controller.addRebalancingRewardHandler(address(omnipool), address(handler));

        // USDC / STG
        omnipool.changeUnderlyingPool(
            0,
            USDC_STG,
            0x3ff3a210e57cfe679d9ad1e9ba6453a716c56a2e0002000000000000000005d5,
            0,
            0,
            1e18,
            PoolType.WEIGHTED
        );
        controller.addRebalancingRewardHandler(
            0x8bd520Bf5d59F959b25EE7b78811142dDe543134, address(handler)
        );
        pools.push(address(omnipool));

        vm.startPrank(opal);

        registryAccess.addRole(ROLE_MINT_LP_TOKEN, address(omnipool));
        registryAccess.addRole(ROLE_BURN_LP_TOKEN, address(omnipool));

        // USDC
        oracle.addPriceFeed(
            address(0xA0b86991c6218b36c1d19D4a2e9Eb0cE3606eB48),
            address(0x8fFfFfd4AfB6115b954Bd326cbe7B4BA576818f6)
        );

        // STG
        oracle.addPriceFeed(
            address(0xAf5191B0De278C7286d6C7CC6ab6BB8A73bA2Cd6),
            address(0x7A9f34a0Aa917D438e9b6E630067062B7F8f6f3d)
        );

        bptOracle.setHeartbeat(0xA0b86991c6218b36c1d19D4a2e9Eb0cE3606eB48, 15 * 86_400);
        bptOracle.setHeartbeat(0xAf5191B0De278C7286d6C7CC6ab6BB8A73bA2Cd6, 15 * 86_400);
    }

    function testPOC1() external {
        IOmnipool pool = IOmnipool(address(omnipool));
        IERC20Metadata token = IERC20Metadata(pool.getUnderlyingToken());
        IERC20 lpToken = IERC20(pool.getLpToken());

        uint256 depositAmount = 10e6;

        console.log("-----");
        console.log("Initial deposit to setup pool...");
        vm.startPrank(bob);
        token.approve(address(pool), type(uint256).max);
        pool.deposit(depositAmount, 0);
        console.log("Bob's LP balance", lpToken.balanceOf(bob));

        // SWAP
        console.log("-----");
        console.log("simulate swaps...");
        // THe more swaps the more Alice loses USDC balance after withdraw
        for (uint256 i; i< 1000; i++) {
          doSwap(address(usdc), STG, 10_000e6);
          doSwap(STG, address(usdc), IERC20(STG).balanceOf(bob));
        }

        console.log("-----");
        console.log("depositing for alice...");
        console.log("alice initial USDC bal:", usdc.balanceOf(alice));
        vm.startPrank(alice);
        token.approve(address(pool), type(uint256).max);
        pool.deposit(depositAmount, 0);
        console.log("alice's LP balance", lpToken.balanceOf(alice));

        vm.roll(1);

        console.log("-----");
        console.log("withdraw for alice...");
        vm.startPrank(alice);
        pool.withdraw(lpToken.balanceOf(alice), 0);
        // Loss of 20% from deposit and withdraw
        assertTrue(usdc.balanceOf(alice) < 9.8e6);
        console.log("alice after USDC bal:", usdc.balanceOf(alice));
    }
```

Impact: The impact depends on how much protocol fees have been accrued in the Balancer pool. The greater the fees, the more a Omnipool depositor suffers.

It will also benefit a user to deposit right after another user had deposited, to avoid impact from this bug.

See live transaction where during `joinPool` it shows 7769 of BPT minted as protocol fees to `0xce88...109f9F` (the `protocolFeesCollector`) first before the rest of the transaction continues.

Recommendation: When computing BPT valuation, `totalSupply` should take into account accrued protocol fees.

3.3.17 Dangerous 1 to 1 price assumption for ETH derivatives

Submitted by giraffe0x

Severity: Medium Risk

Context: Omnipool.sol#L247

Description: `Omnipool.sol` can only handle underlying tokens with a Chainlink USD price feed (e.g. STG/USD). This is evident from

```
uint256 underlyingPrice_ = bptOracle.getUSDPrice(address(underlyingToken));
```

`BPTOracle` **cannot** tokens without a USD price feed such as rETH or vETH, which typically has a ETH price feed e.g. RETH/ETH.

In contrast, in test file `BPTOracle.t.sol` ETH derivatives such as rETH, vETH and uniETH were tested, indicating the team's intention to launch such Omnipools. In the test file, these bad assumptions in the comments were observed:

```
// rETH (considering 1rETH = 1ETH) - We should use the real rETH price feed (which is in rETH/ETH)
oracle.addPriceFeed(
    address(0xae78736Cd615f374D3085123A210448E74Fc6393),
    address(0x5f4eC3Df9cbd43714FE2740f5E3616155c5b8419)
);

// vETH (considering 1vETH = 1ETH)
oracle.addPriceFeed(
    address(0x4Bc3263Eb5bb2Ef7Ad9aB6FB68be80E43b43801F),
    address(0x5f4eC3Df9cbd43714FE2740f5E3616155c5b8419)
);

// uniETH (considering 1vETH = 1ETH)
oracle.addPriceFeed(
    address(0xF1376bceF0f78459C0Ed0ba5ddce976F1ddF51F4),
    address(0x5f4eC3Df9cbd43714FE2740f5E3616155c5b8419)
);
```

Assuming that `1 rETH == 1 ETH` is dangerous for several reasons. First, rETH is non-rebasing and is priced higher than ETH (at time of writing: rETH = $3056 vs. ETH = $2779). Next, rETH value fluctuates depending or demand and supply, so prices could easily "*depeg*" from each other.

Proof of concept:

- Deposit:
  - Omnipool `totalSupply` = 100, valuation = $100000 (from `BPTOracle`).
  - BPT = rETH/WETH, price of rETH = $1500, price of WETH = $1200.
  - `beforeUnderlyingAmount` = $100000 / $1200 = 83.33 → using WETH price due to assumption rETH = ETH
  - `exchangeRate` = 83.33 / 100 = 0.833.
  - Alice deposits 1 rETH ($1,500), receives 1 / 0.833 = 1.2 omnipool LP.
- Withdrawal:
  - Assume temporary depeg: price of rETH = $1500, WETH = $900.
  - Omnipool `totalSupply` = 101.2, valuation = $101500 (no change in rETH price).
  - Alice withdraws all her funds.
  - `totalUnderlying` = $101500 / $900 = 112.78.
  - `exchangeRate` = 112.78 / 101.2 = 1.11.
  - `underlyingToReceive` = 1.2 (amountOut) * 1.11 = 1.332 rETH.

– Alice deposited 1 rETH, withdraws 1.332 rETH, profit of 0.332 rETH.

Impact: A user could take advantage of temporary price difference between rETH and ETH to withdraw more underlying, at the cost to other users who assume that underlying price is based on rETH.

Recommendation: Do not make the assumption that 1 rETH (or any other ETH derivative) = 1 ETH. Also, consider adding oracle price feeds for non-USD pairs by converting to ETH first then ETH to USD.

3.3.18 No way to reduce or set gauge weight to zero due to underflow

Submitted by giraffe0x, also found by 0xTheBlackPanther

Severity: Medium Risk

Context: GaugeController.sol#L448

Description: Due to multiple underflow errors in `_changeGaugeWeight`, admin does not have the ability to reduce or set a gauge's weight to zero:

```
// GaugeController.sol
function _changeGaugeWeight(address gauge, uint256 weight) internal {
    int128 gaugeType = gaugeTypes[gauge] - 1;
    uint256 oldGaugeWeight = _getWeight(gauge);
    uint256 typeWeight = _getTypeWeight(gaugeType);
    uint256 oldSum = _getSum(gaugeType);
    uint256 totalWeight = _getTotal();
    uint256 nextTimestamp = ((block.timestamp + WEEK) / WEEK) * WEEK;

    gaugeVotes[gauge][nextTimestamp] = weight;
    lastGaugeUpdate[gauge] = nextTimestamp;

    //@audit if weight is less than oldGaugeWeight, this will underflow
    uint256 newSum = oldSum + (weight - oldGaugeWeight);

    //@audit this will also underflow when new weight = 0
    totalWeight += (oldSum * weight) - (oldSum * typeWeight);
}
```

If `weight` is less than `oldGaugeWeight`, the calculation for `newSum` will underflow and revert. This is also true for calculation of `totalWeight`.

Impact: There are no ways for an admin to reduce the weight of a gauge or deprecate it by setting weight to zero.

Recommendation: Change the line to `uint256 newSum = oldSum + weight - oldGaugeWeight;` by removing brackets. Need re-look into `totalWeight` calculations too.

3.3.19 Refunds meant for the depositors are lost and stuck in the `omnipool.sol` contract

Submitted by bronzepickaxe

Severity: Medium Risk

Context: Omnipool.sol#L463-L469

Description: In the `Omnipool.deposit` flow, the `depositSingle` fucntion gets called of Aura Finance:

```
function _depositToAuraPool(UnderlyingPool memory _pool, uint256 _underlyingAmountIn)
    internal
{
    uint256[] memory amountsIn = new uint256[](_pool.assets.length);

    // create join request
    uint256[] memory userDataAmountsIn = amountsIn;
    amountsIn[_pool.assetIndex] = _underlyingAmountIn;

    // if the assets include bpt, we must remove it from the amountsIn in the userData
    if (_pool.bptIndex > 0) {
        userDataAmountsIn = new uint256[](_pool.assets.length - 1);
        userDataAmountsIn[_pool.assetIndex - 1] = _underlyingAmountIn;
    }

    bytes memory userData =
        abi.encode(IBalancerVault.JoinKind.EXACT_TOKENS_IN_FOR_BPT_OUT, userDataAmountsIn, 2);

    // join balancer pool
    IRewardPoolDepositWrapper.JoinPoolRequest memory joinRequest = IRewardPoolDepositWrapper
        .JoinPoolRequest({
        assets: _pool.assets,
        maxAmountsIn: amountsIn,
        userData: userData,
        fromInternalBalance: false
    });

    // deposit into aura
        auraRewardPoolDepositWrapper.depositSingle(
        address(_pool.poolAddress),
        underlyingToken,
        _underlyingAmountIn,
        _pool.poolId,
        joinRequest
    );
}
```

If we take a look at this `depositSingle` function, we find the following:

```
function depositSingle(
    address _rewardPoolAddress,
    IERC20 _inputToken,
    uint256 _inputAmount,
    bytes32 _balancerPoolId,
    IBalancerVault.JoinPoolRequest memory _request
) external {
    // 1. Transfer input token
    _inputToken.safeTransferFrom(msg.sender, address(this), _inputAmount);

    // 2. Deposit to balancer pool
    (address pool, ) = bVault.getPool(_balancerPoolId);

    _inputToken.approve(address(bVault), _inputAmount);
    bVault.joinPool(_balancerPoolId, address(this), address(this), _request);

    uint256 minted = IERC20(pool).balanceOf(address(this));
    require(minted > 0, "!mint");

    uint256 inputBalAfter = _inputToken.balanceOf(address(this));
    if (inputBalAfter != 0) {
        // Refund any amount left in the contract
        _inputToken.transfer(msg.sender, inputBalAfter);
    }

    // 3. Deposit to reward pool
    IERC20(pool).approve(_rewardPoolAddress, minted);
    IRewardPool4626(_rewardPoolAddress).deposit(minted, msg.sender);
}
```

As you can see, in this line:

```
_inputToken.transfer(msg.sender, inputBalAfter);
```

If there's any amount left, it will be refunded to the `msg.sender`. However, the `msg.sender` in this case will be the `Omnipool.sol` contract.

This means that the refund, which is meant for the person making a deposit, will not be creditted to the person making the deposit but instead will be stuck in the `Omnipool.sol` contract, effectively losing a portion of the funds of the person who made the deposit.

Recommendation: Handle the cases where a refund is returned to the `Omnipool.sol` contract.

3.3.20 Missing check for the `minprice/maxprice` price in the `bptoracle.sol` contract

Submitted by Cheems, also found by 0xRizwan

Severity: Medium Risk

Context: *(No context files were provided by the reviewer)*

There are 2 issues in `BPTOracle.getUSDPrice`:

1. There is no check for negative or zero price returned by `priceFeed.latestRoundData()`, which could lead to underflow:

```
function getUSDPrice(address token) public view returns (uint256 priceInUSD) {
    if (tokenHeartbeat[token] == 0) { revert HeartbeatNotSet(); }

    IOracle priceFeed = IPriceFeed(priceFeedAddress).getPriceFeedFromAsset(token);
    if (address(priceFeed) == address(0)) revert PriceFeedNotFound();
    // @audit there is no check for negative or zero price.
    (, int256 priceInUSDInt,, uint256 updatedAt,) = priceFeed.latestRoundData();
    if (updatedAt + tokenHeartbeat[token] < block.timestamp) revert StalePrice();

    // simulate negative price
    priceInUSDInt = -1;
    // Oracle answer are normalized to 8 decimals
    uint256 newPrice = _normalizeAmount(uint256(priceInUSDInt), 8);
    return newPrice;
}

// Test
function test_NegativePrice() public {
    uint256 amount = bptPrice.BptPriceStablePool(
        bytes32(0x06df3b2bbb68adc8b0e302443692037ed9f91b42000000000000000000000063)
    );
}
```

2. There is no check to ensure returned `priceInUSDInt` is between `minPrice/maxPrice`, which is recommended. Chainlink has circuit breaker if the price of an asset goes outside of a predetermined price band (`minPrice/maxPrice`).

For example if asset had a huge drop in value it will continue to return the `minPrice` instead of the actual price. in this case `getUSDPrice` will return the wrong price.

Impact: If an asset drop in value wrong price will be returned and wrong calculattion will happen when using `getUSDPrice` wich will affect the protocol.

Recommendation: Ensure that the returned `priceInUSDInt` is between `minPrice/maxPrice` to avoid results based on incorrect prices.

```
  function getUSDPrice(address token) public view returns (uint256 priceInUSD) {
      if (tokenHeartbeat[token] == 0) { revert HeartbeatNotSet(); }

      IOracle priceFeed = IPriceFeed(priceFeedAddress).getPriceFeedFromAsset(token);
      if (address(priceFeed) == address(0)) revert PriceFeedNotFound();
      (, int256 priceInUSDInt,, uint256 updatedAt,) = priceFeed.latestRoundData();
      if (updatedAt + tokenHeartbeat[token] < block.timestamp) revert StalePrice();

+     if (signedPrice < 0) revert negative_priceInUSDInt();
+     if (priceInUSDInt < minPrice) revert hit_minPrice();
+     if (priceInUSDInt > maxPrice) revert hit_maxPrice();

      // Oracle answer are normalized to 8 decimals
      uint256 newPrice = _normalizeAmount(uint256(priceInUSDInt), 8);
      return newPrice;
  }
```

3.3.21 Users will lose their rebalancing rewards due to rebalancing flag not being reset on pool imbalances

Submitted by 0xadrii, also found by 0xTheBlackPanther

Severity: Medium Risk

Context: Omnipool.sol#L1050

Description: Opal's liquidity rebalancing feature allows deposited assets to be distributed among several Aura pools considering certain preconfigured weights. In order to incentivize users to deposit liquidity so that pools are balanced, Opal offers rewards in the form of `GEM`. Such rewards are distributed at the end of each `depositFor()` function call, via the `_handleRebalancingRewards()` function:

```
// Omnipool.sol

function _handleRebalancingRewards(
    address account,
    uint256 allocatedBalanceBefore_,
    uint256 allocatedBalanceAfter_,
    uint256[] memory allocatedPerPoolBefore,
    uint256[] memory allocatedPerPoolAfter
) internal {
    if (!rebalancingRewardActive) return;
    uint256 deviationBefore =
        _computeTotalDeviation(allocatedBalanceBefore_, allocatedPerPoolBefore);
    uint256 deviationAfter =
        _computeTotalDeviation(allocatedBalanceAfter_, allocatedPerPoolAfter);

    IOmnipoolController controller =
        IOmnipoolController(registryContract.getContract(CONTRACT_OMNIPOOL_CONTROLLER));
    controller.handleRebalancingRewards(account, deviationBefore, deviationAfter);

    if (_isBalanced(allocatedPerPoolAfter, allocatedBalanceAfter_)) {
        rebalancingRewardActive = false;
    }
}
```

The code snippet shows how the `rebalancingRewardActive` is crucial in order to determine if rewards should be handled or not. If `rebalancingRewardActive` is set to `false`, the function will simply return and no `GEM` rewards will be distributed. This flag will be set to false if it is detected that pools are balanced (i.e. `_isBalanced()` returns `true`).

This vulnerability aims at describing how not setting the `rebalancingRewardActive` to `true` in other situations might lead to users not obtaining their entitled GEM rewards.

The `rebalancingRewardActive` is set as `false` by default. It will only be set to `true` when the `updateWeights()` or `updateWeight()` functions are called. These functions allow Opal to change the currently configured pool weights and, if after changing them the pools are not balanced (i.e. `_isBalanced()` returns `false`), then the `rebalancingRewardActive` will be set to `true`, allowing users who deposit and help the pools return to a balanced state be rewarded with `GEM` tokens:

```
// Omnipool.sol
function updateWeights(IOmnipoolController.WeightUpdate[] calldata poolWeights)
    external
    override
    onlyController
{
    // ...
    rebalancingRewardActive = !_isBalanced(allocatedPerPool, totalAllocated);
}

// ...

function updateWeight(address poolAddress, uint256 newWeight)
    external
    override
    onlyController
{
    // ...
    rebalancingRewardActive = !_isBalanced(allocatedPerPool, totalAllocated);
}
```

The problem with the current approach is that Opal considers that the only situation where pools might be imbalanced (and thus `rebalancingRewardActive` should be set to `true`) is when updating the pools' weights via the previously mentioned functions.

However, pools might be imbalanced given other circumstances. A good example of a situation where pools might become imbalanced is by withdrawing small amounts from Opal. Such small withdrawal amounts might make withdrawing from only one pool more than enough to cover the withdrawal request, reducing the pool's allocated assets and effectively causing an imbalance considering the configured pool weights.

Consider the following scenario, with an initial state where pools are imbalanced and hence `rebalancingRewardActive` is set to `true`:

1. A user deposits into Opal and `_handleRebalancingRewards()` is executed. The deposit makes the pools be balanced, and the `rebalancingRewardActive` is set to `false`.
2. A second user withdraws from Opal and causes pools to be imbalanced.
3. A third user deposits into Opal and pools get balanced again. However, because the `rebalancingRewardActive` was set to `false` in step 1, `_handleRebalancingRewards()` will simply skip the reward process, making the third user not receiving the rewards that he's actually entitled to

Impact: Medium. Rewards might be lost in certain situations, and Opal's liquidity rebalancing feature (one of the strong points of the protocol) might be affected due to improperly incentivizing users.

Recommendation: Check if pools are actually balanced at the beginning of the `_handleRebalancingRewards()` execution, and reset the `rebalancingRewardActive` accordingly. This will make Opal able to decide if rewards should be handled and distributed to the depositor.

3.3.22 `_getdepositpool()` and `_getwithdrawpool()` can halt deposits and withdrawals for an omnipool

Submitted by 0xJaeger, also found by zigtur

Severity: Medium Risk

Context: Omnipool.sol#L522, Omnipool.sol#L620

Description: The `_getDepositPool()` and `_getWithdrawPool()` functions are responsible for managing the allocation between pools. However, if all pools reach their target allocation, both functions will fail to select a pool for depositing/withdrawing funds, effectively halting all withdrawals and deposits until the target weight of an underlying pool is adjusted.

Proof of concept:

1. An omnipool is set up with two underlying balancer pools, each with a target weight of 50/50. (for sake of simplicity).
2. User1 deposits 500 tokens, allocated to pool 1.
3. User2 deposits 500 tokens, allocated to pool 2 as pool 1 has reached its target weight.
4. Both pools are now at their target allocations.
5. User3 attempts to deposit funds but encounters an error message from `_getDepositPool()` due to the inability to select a deposit pool index.
6. User4 attempts to withdraw funds but encounters an error message from `_getWithdrawPool()` due to the inability to select a withdrawal pool index.

Impact Deposits and withdrawals in an omnipool can be halted for a minimum of two weeks, as this is the official designated period for Liquidity Allocation Votes (LAV) for an omnipool (per documentation).

Omnipools are most vulnerable when freshly deployed and beginning to accept deposits, as attackers can easily manipulate pool balances with minimal funds. However, as more time passes and more funds are locked in the omnipool, executing such an attack becomes increasingly difficult.

Recommendation: Remove the equality checks in both cases to allow users to deposit/withdraw funds within the permitted deviation. This adjustment will enable deposits/withdrawals to be distributed equally among all pools_

```
  //_getDepositPool()
- if (currentAlloc >= targetAllocation_) continue;
+ if (currentAlloc > targetAllocation_) continue;

  //_getWithdrawPool()
- if (currentAlloc <= targetAllocation_) continue;
+ if (currentAlloc < targetAllocation_) continue;
```

3.3.23 Removedamount will remain stuck forever in the contract if `totalvesting` becomes 0

Submitted by 0xadrii, also found by Sujith Somraaj and innertia

Severity: Medium Risk

Context: EscrowedToken.sol#L300

Description: The `EscrowedToken` contract allows users to claim some escrowed tokens during a certain period of time. If users wait until the end of the claiming period, they will be able to claim the full amount of escrowed tokens. However, if they try to claim the tokens prior to the ending period of time, the proportional amount of tokens that should be claimed in the remaining time until the claiming period ends will be distributed among the users that have funds currently escrowed.

In order to distribute these "*missed*" funds, a `ratePerToken` variable is used, which acts as a ratio that will increment so that the remaining users can claim the missed funds by the early claimer. This ratio is increased inside the `_claim()` function:

```
function _claim(address account, uint256 vestingIndex) internal {

    // ...

    uint256 claimAmount = (
        userVesting.amount * (SCALED_ONE + (ratePerToken - userVesting.ratePerToken))
    ) / SCALED_ONE;
    uint256 removedAmount = (claimAmount * remainingTime) / vestingDuration;
    claimAmount -= removedAmount;

    userVesting.claimed = true;
    totalVesting -= userVesting.amount;

    // ...

    if (totalVesting > 0) {
        ratePerToken += (SCALED_ONE * removedAmount) / totalVesting;
    }

    // ...
}
```

As the snippet shows, the `ratePerToken` will increment by `(SCALED_ONE * removedAmount) / totalVesting` . However, a problem arises with this approach. The Opal team decided to add a check that prevents the `ratePerToken` be incremented if `totalVesting` becomes 0 (i.e. no vested funds remain in the contract). This is done to avoid the previous calculation to perform a division by 0.

The problem with this implementation is that because funds are not distributed among vesting users (because there are no remaining vested users, and `ratePerToken` does not increment), and no other action is taken to handle the missed escrowed tokens, the `removedAmount` funds will remain stuck forever in the contract, because such funds are simply not accounted or tracked in any other way.

Impact: This situation might arise at some point, effectively leaving some of the escrowed tokens stuck forever in the contract, without any way to retrieve them.

Proof of concept: The following proof of concept shows how this situation might arise. In this case, Alice claims the escrowed tokens only 10 seconds after the vesting has been created. Hence, most of her rewards remain stuck forever in the contract.

In order to run the proof of concept, copy the following code in the `EscrowedToken.t.sol` contract inside the `test` folder (make sure you import `"forge-std/console.sol";` at the top of the file):

```
function testVuln_fundsRemainStuckForeverIfTotalVestingBecomesZero() public {

    // Step 1: Vest 1e18 tokens with a `startTimestamp` of 50
    createAVesting(1 ether, alice, 50);

    // Step 2: Go forward in time to block.timestamp 60. Fetch alice's balance prior to claiming
    vm.warp(60);

    // Step 3: Claim
    vm.prank(alice);
    escrowedToken.claim(0);

    // Step 4: Perform validations
    // Validate that the `ratePerToken` has not incremented
    assertEq(escrowedToken.ratePerToken(), SCALED_ONE);
    // Because `token` has not been fully distributed to Alice, a remaining amount will be kept forever in the
↪   contract
    assertGt(token.balanceOf(address(escrowedToken)), 0);
}
```

Then, run the proof of concept with the following command: `forge test --mt testVuln_fundsRemainStuckForeverIfTotalVestingBecomesZero -vv`

Recommendation: It is recommended to add a `recoveUnclaimableFunds()` function that allows the Opal team to recover funds that might remain stuck in the contract forever due to the mentioned situation. It is important to ensure that this new function can only recover funds that are actually stuck forever in the contract, and not funds that might be distributed. Such stuck funds could be tracked with a `unclaimableFunds` variable that gets incremented by `removedAmount` if `totalVesting` becomes zero when claiming the assets. Then, the new function should set the `unclaimableFunds` to zero again, so that no more funds than the intended are claimed:

```
function recoveUnclaimableFunds() external onlyOpalTeam {

    token.safeTransfer(msg.sender, unclaimableFunds);

    unclaimableFunds = 0;

    emit StuckFundsRecovered();

}
```