OlympusDAO Single Sided Liquidity Vault System (SSLV) Audit

Introduction & Scope

This audit looks into Pull Request 103 as seen here (https://github.com/OlympusDAO/bophades/pull/103/files). This includes the following contracts:

- `LQREG.v1.sol` (https://github.com/OlympusDAO/bophades/pull/103/files#diff-5b92f893d14c85ef44073dd716b1dcfd32cb794762219a99711ed507fd087bdb)
- `OlympusLiquidityRegistry.sol` (https://github.com/OlympusDAO/bophades/pull/103/files#diff-5e7445bb67500e7f1fd097919a7e0603c675db0229ddad3fc581b8749e658fbf)
- `SingleSidedLiquidityVault.sol` (https://github.com/OlympusDAO/bophades/pull/103/files#diff-e0e208e2b4d221c3bd3cb086cb9766f5eaba0574bae0548eb776b763e96c2b03)
- `StethLiquidityVault.sol` (https://github.com/OlympusDAO/bophades/pull/103/files#diff-af78df6669860e2f7153d75337038924a5a1ff115ed548adb39a75dd74876aaf)

This audit was conducted by kebabsec (https://twitter.com/kebabsec) members sai (https://twitter.com/sigh242), FlameHorizon (https://twitter.com/FlameHorizon1) and okkothejawa (https://twitter.com/okkothejawa).

As SSLV can affect the overall OHM supply, its design itself may have unexpected consequences for the economics of OHM ecosystem. As the authors of this report lack the expertise and proper information to audit the economic design of the system, this audit should not be treated as a design audit.

Executive Summary

Table of Contents

Findings:

1. [MED] Wrong decimal value for stETH/USD price feed in `StethLiquidityVault.sol`

- In `StethLiquidityVault.sol` , line 215 (https://github.com/OlympusDAO/bophades/blob/0b57e988377afa84b52727de0f718b9e265e7bb1/src/policies/lending/StethLiquidityVault.sol#L215), a comment affirms that the oracle price feed for stETH/USD reports a price in 18 decimals, after double checking it, we noticed this was not accurate, and that the price was in fact being reported in 8 decimals. Incorrectly assuming the decimal count of a feed would complicate the calculations to come after contained in the same function, more specifically the math for the return value in line 223 (https://github.com/OlympusDAO/bophades/blob/0b57e988377afa84b52727de0f718b9e265e7bb1/src/policies/lending/StethLiquidityVault.sol#L233).
- **Recommendation**: Change the calculations to take into account that stETH/USD price feed reports in 8 decimals.

2. [LOW] `safeTransferFrom` and `safeTransfer` not present

- As `pairToken` and `rewardToken` tokens might not be standard ERC20 tokens that revert on failure (they may return false and silently pass as opposed to more common way of reverting) or they may not return a boolean like USDT, consider utilizing a `SafeERC20` (https://github.com/OpenZeppelin/openzeppelin-

contracts/blob/master/contracts/token/ERC20/utils/SafeERC20.sol) library for token transfers throughout the contract to account for non ERC20 compliant tokens.
- **Recommendation**: Use `safeTransfer` and `safeTransferFrom` instead of `transfer` and `transferFrom` while handling tokens other than `OHM` , especially in the abstract contract as the tokens are not known beforehand. See these related (https://github.com/code-423n4/2021-08-notional-findings/issues/68) C4 issues (https://github.com/code-423n4/2022-01-trader-joe-findings/issues/12).

3. [HIGH] Faulty math in `_canDeposit` leads to Denial of Service

- Due to faulty math, the sanity check `_canDeposit` (https://github.com/OlympusDAO/bophades/blob/ss-liq-vault/src/policies/lending/abstracts/SingleSidedLiquidityVault.sol#L317) checks if `ohmMinted - ohmBurned + amount_ > LIMIT` in both branches as the variable `activeOhm` evaluates to `ohmMinted - ohmBurned` in each of the branches. As Solidity >0.8 reverts when a `uint` is evaluated as negative, the states in which `ohmBurned > ohmMinted` results in `_canDeposit` reverting. As the function `deposit` is the only path that can increase the variable `ohmMinted` and as `deposit` utilizes `_canDeposit` sanity check, once the contract gets into a state in which `ohmBurned > ohmMinted` which can occur naturally due to price fluctations in the Balancer pool, no further deposits are possible after that point, resulting in denial of service. As the only way to resume operations in such scenario is upgrading/replacing the contract, see issue 8 [INFORMATIONAL] No emergency protections present.
- **Recommendation**: Fix the faulty math.

4. [MED] `deposit` does not conform to checks-effects-interactions pattern

- As can be seen here (https://github.com/OlympusDAO/bophades/blob/ss-liq-vault/src/policies/lending/abstracts/SingleSidedLiquidityVault.sol#L166), state changes happen after `transfer` and `transferFrom` calls to `pairToken` in `deposit` . This is against the safe pattern of checks-effects-interactions, and it can lead to reentrancy attacks utilizing a `pairToken` with callback capabilities (e.g an ERC777 or a modified ERC20). Even though the `nonReentrant` modifier is used in both external functions of the contract, this modifier can only prevent reentrancy attacks caused by calling these functions mid-function, yet the attacker can manipulate the state by either tampering with the liquidity pool or dumping `OHM` directly into the contract in the callback.
    - **A) Tampering with the liquidity pool**
      The attacker can bypass the check `_isPoolSafe` by having an initially safe LP pool state to pass `_isPoolSafe` , and then proceeding to manipulate the pool price in the callback from `pairToken.transferFrom` call before `_deposit` , resulting in a deposit to an unsafe pool. The current Balancer implementation is not likely to be susceptible to this as all external state-changing functions of

Balancer has the `nonReentrant` modifier, meaning that any price manipulation during the transaction would lock the Balancer vault, resulting in revert in the `_deposit` call. Yet, further implementations utilizing different liquidity pool designs might be susceptible to bypassing of `_isPoolSafe`, if the underlying liquidity pools don't implement a mutex in the necessary places.

- **B) Dumping `OHM` directly into contract during execution**
  As the variable `ohmMinted` is increased by the difference in the `OHM` balance before and after the `transferFrom` calls, the attacker can utilize the callback to directly send `OHM` into the contract to inflate `unusedOhm`, causing `ohmMinted` to increase less than what is supposed to. This path can be utilized to force issue 3 by inflating the difference between `ohmBurned` and `ohmMinted`. Other than causing denial of service by issue 3 and tampering with the variables `ohmMinted` or `pairTokenDeposits`, no direct exploitation possibility for this path was found in the audit.

- **Recommendation**: Refactor `deposit` so that it conforms to CEI pattern.

5. [MED] Truncations and unsafe casts in the rewards logic can lead to weird states such as infinite rewards

- PoC Foundry test can be found here. (https://gist.github.com/okkothejawa/09204b0c16a82644da9a42aa8a9dbd40)
- As a `MasterChef` variation, the rewards logic of SSLV includes potential truncations (division results in 0 in Solidity if numerator is smaller than denominator as Solidity does not support floats) and unsafe casts (casting a `int` to an `uint` is dangerous as underflow/overflow in casting is not patched after Solidity >0.8).
  While our research didn't yield a path that is directly exploitable, we found a interesting enough weird state that is accessible through a valid path in which a first depositor deposits a large amount of `pairToken` so that `totalLP` starts large, resulting in a small `accumulatedRewardsPerShare`. Then a second depositor makes a deposit of usual `1e18`, and proceeds to withdraw only `1`. As rewards are calculated as in the snippet below in `internalRewardsForToken`, once `userRewardDebts[user_][rewardToken.token]` is larger than `int256((lpPositions[user_] * accumulatedRewardsPerShare) / 1e18)` the outermost `int` value becomes negative, underflowing `uint` and resulting in infinite rewards value. This behavior happens as the withdrawn amount truncates when its subtracted from the `userRewardDebts` in `_withdrawUpdateRewardState` (https://github.com/OlympusDAO/bophades/blob/ss-liq-vault/src/policies/lending/abstracts/SingleSidedLiquidityVault.sol#L499) due to both withdrawn LP amount and rewards per share being small and their product is smaller than `1e18`, which leads to `userRewardDebts` staying the same while `int256((lpPositions[user_] * accumulatedRewardsPerShare) / 1e18)` is decreased.

```
uint256(int256((lpPositions[user_] * accumulatedRewardsPerShare) / 1e18) - user
```

- Even though the above scenario highlights the problems associated with truncations and unsafe casts, it is not practically exploitable as the contract wouldn't have an infinite amount of rewards and the fee calculation would overflow and revert. We extensively studied the contract to see if an arbitrary negative number can be reached through the means explained above, yet we couldn't find such a path. Yet, this does not mean such a path does not exist, and thus usage of potential truncations and unsafe casts should be minimized in order to prevent potential rewards drainage paths.
- **Recommendation**: Minimize truncations and unsafe casts.

6. [LOW] Add sanity check when setting variables THRESHOLD and FEE, specially THRESHOLD since it breaks `_isPoolSafe`

- The function `setThreshold` should be bound within the parameters to stay compliant with the comment above, specially since it cannot be bigger than `PRECISION` , which is hardcoded to the value of 1000, and accidentally adding another zero accidentally when using this function would break `_isPoolSafe` and revert and all the functions that depend on this check to function.
- **Recommendation**: Implement sanity checks in potentially contract breaking setters.

7. [INFO] No emergency protections present

- This module does not have emergency protections present, and in case of a critical event there should be a way to prevent withdraws and other sensitive functions.
- There is also no way to initialize the contract with a pre-set mapping of LP positions, thus upgrading the contract with a new one as a response to a critical event is tricky. An emergency migration function can be designed as the following:
  1. Withdraw LP tokens from Aura.
  2. Transfer LP tokens to the new implementation.
  3. Initialize the new contract with the old state mappings.
  4. Deposit the LP tokens into Aura again.
  5. Resume operation as usual.
- **Recommendation**: Consider implementing permissioned emergency migration and pause/unpause functions.

8. [LOW] Event not emitted due to dead code in `withdraw`

- In line 228 (https://github.com/OlympusDAO/bophades/blob/0b57e988377afa84b52727de0f718b9e265e7bb1/src/policie

s/lending/abstracts/SingleSidedLiquidityVault.sol#L228) of `SingleSidedLiquidityVault.sol` there is a return statement that prevents the posterior event in line 230 (https://github.com/OlympusDAO/bophades/blob/0b57e988377afa84b52727de0f718b9e265e7bb1/src/policies/lending/abstracts/SingleSidedLiquidityVault.sol#L230) to not be emitted.
- **Recommendation**: Swap the return line with the line firing the event.

9. [INFO] Insufficient test coverage

- Upon inspection, the contract's test unit doesn't simulate the correct functionality of production, and has the following issues:
    - The withdraw testing only applies a condition of warping time, but has no test of a condition without passing time.
    - No fuzzing of deposit and withdraw amount inputs, leading to problems.
- Mock contracts do not provide the correct functionality compared to mainnet deployment environment and can't be verified:
    - `MockAuraRewardPool` and `BalancerMockVault` mock by naive token minting. This can not provide sufficient verification of Vault's external function calls.
- **Recommendation:** Extend the test coverage and improve the accuracy of mocks.

10. [INFO] Claim reward functions does not return value (https://github.com/OlympusDAO/bophades/blob/0b57e988377afa84b52727de0f718b9e265e7bb1/src/policies/lending/abstracts/SingleSidedLiquidityVault.sol#L527-L551)

- The functions `_claimInternalRewards` and `_claimExternalRewards` are declared to return uint256, but does not return anything.
- **Recommendation:** Function should return claimed tokens amount or be declared without return.

11. [INFO] `_accumulateInternalRewards` does not make state changes and can be set to view (https://github.com/OlympusDAO/bophades/blob/ss-liq-vault/src/policies/lending/abstracts/SingleSidedLiquidityVault.sol#L385)

- **Recommendation:** Set function to be `view` .

12. [MED] Possible reentrancy in `claimRewards`

- The claim function makes several external calls, presented for external token accumulation in _accumulateExternalRewards (https://github.com/OlympusDAO/bophades/blob/0b57e988377afa84b52727de0f718b9e265e7bb1/src/policies/lending/StethLiquidityVault.sol#L171-L195):

```
182     auraPool.rewardsPool.getReward(address(this), true);
```

- While `withdraw` function is protected against reentrancy that allows claiming rewards, `claimRewards` can be claimed.
- **Recommendation:** Function `claimRewards` should have `nonReentrant` to ensure reentrancy safety.

13. [INFO] Tokens rewarding is not accounted for tokens less or more than 18 decimals

- The vault will miscalculate reward token amount if the token's decimal is not 18. The calculation occurs in following functions: _updateInternalRewardState (https://github.com/OlympusDAO/bophades/blob/0b57e988377afa84b52727de0f718b9e265e7bb1/src/policies/lending/abstracts/SingleSidedLiquidityVault.sol#L407), _updateExternalRewardState (https://github.com/OlympusDAO/bophades/blob/0b57e988377afa84b52727de0f718b9e265e7bb1/src/policies/lending/abstracts/SingleSidedLiquidityVault.sol#L415), _depositUpdateRewardDebts (https://github.com/OlympusDAO/bophades/blob/0b57e988377afa84b52727de0f718b9e265e7bb1/src/policies/lending/abstracts/SingleSidedLiquidityVault.sol#L453), _withdrawUpdateRewardState (https://github.com/OlympusDAO/bophades/blob/0b57e988377afa84b52727de0f718b9e265e7bb1/src/policies/lending/abstracts/SingleSidedLiquidityVault.sol#L484), `internalRewardsForToken` and `externalRewardsForToken`
- **Recommendation:** Make sure all reward tokens have 18 decimals, or extend the logic to support decimals other than 18.

14. [LOW] `withdraw` insufficient amount input checking (https://github.com/OlympusDAO/bophades/blob/0b57e988377afa84b52727de0f718b9e265e7bb1/src/policies/lending/abstracts/SingleSidedLiquidityVault.sol#L202)

- The function doesn't fully ensure the correctness of passed inputs:
  - A user is able to call withdraw with zero amount in `lpAmount_` and `minTokenAmounts_`.
- `withdraw` makes external calls to Balancer and Aura, checks current token balance and makes update states to reward tokens. Which shouldn't occur.
- **Recommendation:** Add sanity checks for `lpAmount_` and `minTokenAmounts_`.

15. [HIGH] Vault receiving reward tokens outside `_accumulateExternalRewards` from AURA pool can't be accounted and claimed

(https://github.com/OlympusDAO/bophades/blob/0b57e988377afa84b52727de0f718b9e265e7bb1/src/policies/lending/StethLiquidityVault.sol#L171-L195)

- The accumulation is done by checking current's balance before and after pool call rewarded upon updating reward state when depositing (https://github.com/OlympusDAO/bophades/blob/0b57e988377afa84b52727de0f718b9e265e7bb1/src/policies/lending/abstracts/SingleSidedLiquidityVault.sol#L430) and withdrawing (https://github.com/OlympusDAO/bophades/blob/0b57e988377afa84b52727de0f718b9e265e7bb1/src/policies/lending/abstracts/SingleSidedLiquidityVault.sol#L490).
- According to the Aura's contract `BaseRewardPool` implementation, `getReward` can be called by anyone, passing the vault's address:

```
function getReward(address _account, bool _claimExtras) public updateReward
    uint256 reward = earned(_account);
    if (reward > 0) {
        rewards[_account] = 0;
        rewardToken.safeTransfer(_account, reward);
        IDeposit(operator).rewardClaimed(pid, _account, reward);
        emit RewardPaid(_account, reward);
    }

    //also get rewards from linked rewards
    if(_claimExtras){
        for(uint i=0; i < extraRewards.length; i++){
            IRewards(extraRewards[i]).getReward(_account);
        }
    }
    return true;
}
```

- Assuming this is the correct pool, `getReward()` call can be triggered by anyone which will result in rewards will be transferred to the vault without getting recorded as it won't be accounted as a balance change in the function `_accumulateExternalRewards`.
- The vault also does not have a general token sweep function, and the only way to transfer the rewards out are through reading the recorded `accumulatedExternalRewards` values, thus the rewards will be stuck in the contract.
- This path can be used as a griefing attack, as it results in monetary loss for both the users of SSLV and the protocol in the form of potential fees.
- **Recommendation:** Change accumulation logic so that it covers rewards accumulation happening outside of `_accumulateExternalRewards`.