Omo Security Review

Pashov Audit Group

Conducted by: 0xunforgiven, Hals, Ch_301

January 25th 2025 - January 31st 2025

Contents

1. About Pashov Audit Group 5
2. Disclaimer 5
3. Introduction 5
4. About Omo 5
5. Risk Classification 6
   5.1. Impact 6
   5.2. Likelihood 6
   5.3. Action required for severity levels 7
6. Security Assessment Summary 7
7. Executive Summary 8
8. Findings 13
   8.1. Critical Findings 13
      [C-01] Users can duple their first deposit in the vault 13
      [C-02] Passing the wrong value to _burn() function 14
      [C-03] Users can't redeem funds due to malicious manipulation of _totalAssets 14
      [C-04] Funds get locked in the vault 15
      [C-05] The users are not able to redeem their funds 16
      [C-06] borrow() should decrease totalAssets value 17
      [C-07] Unrestricted OmoAgent.onERC721Received() allows permanent DoS and stuck funds 18
   8.2. High Findings 20
      [H-01] Anyone can redeem from users and take the funds 20
      [H-02] The transferFrom() in OmoRouter.sol has approval vulnerabilities 21
      [H-03] OmoVault malicious accounts can permanently disable the vault 22
      [H-04] Cross-vault misattribution leads to incorrect OmoVaults valuation 23
      [H-05] OmoAgent.topOffToAgent incorrect position retrieval logic 24

26

[H-06] OmoOracle incorrect token decimals handling in getUSDValue and convertUSDToToken
[H-07] OmoOracle wrong use of Uniswap interface to get pool address 27
[H-08] OmoOracle getLiquidityAmounts() uses spot price making it manipulatable 28
[H-09] OmoAgent: getPositionValue() doesn't consider deposited assets 29
[H-10] _validateSignature() does not properly handle address(0) 29
[H-11] Pending withdrawal tokens in redeem() affect share price 30
[H-12] getPositionValue() does not consider accumulated fees 31
8.3. Medium Findings 33
[M-01] Funds not always in vault lead to share price calculation mess 33
[M-02] Entry point contract cannot interact with OmoAgent.sol functions 33
[M-03] OmoOracle.convertUSDToToken() : missing price staleness validation 34
[M-04] OmoOracle.getUSDValue price feed updates may be incorrectly marked stale 35
[M-05] OmoAgent.setPositionManager change results in loss of deposited positions 36
[M-06] Incorrect handling of stETH token in OmoRouter.deposit() function 36
[M-07] Missing slippage protection in deposit(), mint(), and redeem() functions 38
[M-08] Unrestricted access in OmoRouter.transfer() function 39
[M-09] OmoRouter.approve() function allows unauthorized token approvals 39
[M-10] OmoRouter.mint function unusable due to missing asset handling 40
[M-11] OmoRouter: pendingRedeemRequest() calls a non-existent function 41
42

[M-12] OmoRouter registerAccount does not validate msg.sender as owner
[M-13] OmoVault/OmoRouter: no way to remove whitelist restriction 43
[M-14] OmoAgent removeAgent should not remove agent with positive debt 44
[M-15] Unbounded loops can lead to out-of-gas errors in contracts 45
[M-16] OmoVault does not enforce supplyCap 45
[M-17] OmoVault first depositor can inflate share price by donating 47
[M-18] OmoAgent: topOffToAgent() does not process and remove all positions 47
8.4. Low Findings 49
[L-01] Not all tokens return a boolean value 49
[L-02] Protocol can't handle fee-on-transfer tokens 49
[L-03] Hardcoded address for entry point 50
[L-04] Users can't withdraw native tokens from the Dynamic Account 50
[L-05] OmoVault.redeem() : Incorrect owner recorded in redemption requests 50
[L-06] Lack of cancellation mechanism for unfulfilled redemption requests in OmoVault 51
[L-07] Missing functionality to unregister malicious agents in OmoVault 51
[L-08] Agent cannot call dynamic account through the entry point 52
[L-09] OmoAgent.addAgent() : lack of agent whitelist validation 52
[L-10] Missing registered agent validation in OmoVault.borrow() function 53
[L-11] OmoVault.borrow missing receiver validation enables unauthorized transfers 54
[L-12] Users unable to redeem assets if removed from OmoRouter whitelist 54
[L-13] Ineffective whitelisting in OmoRouter contract functions 55

[L-14] OmoVault.borrow() : missing limits and tracking mechanism 56
[L-15] DynamicAccount incorrect argument in validateSignature 56

1. About Pashov Audit Group

Pashov Audit Group consists of multiple teams of some of the best smart contract security researchers in the space. Having a combined reported security vulnerabilities count of over 1000, the group strives to create the absolute very best audit journey possible - although 100% security can never be guaranteed, we do guarantee the best efforts of our experienced researchers for your blockchain protocol. Check our previous work here or reach out on Twitter @pashovkrum.

2. Disclaimer

A smart contract security review can never verify the complete absence of vulnerabilities. This is a time, resource and expertise bound effort where we try to find as many vulnerabilities as possible. We can not guarantee 100% security after the review or even if the review will find any problems with your smart contracts. Subsequent security reviews, bug bounty programs and on-chain monitoring are strongly recommended.

3. Introduction

A time-boxed security review of the **omo-protocol/omo-mvp** repository was done by **Pashov Audit Group**, with a focus on the security aspects of the application's smart contracts implementation.

4. About Omo

Omo is a liquidity pool system built on ERC4626-compliant vaults, featuring modular components like OmoVault, AgentSetter, OmoStaking, OmoVaultFactory, and OmoRouter for asset management, staking, and permission control. Its architecture emphasizes modularity, gas optimization, and flexible configuration, enabling independent vault operations and streamlined upgrades.

5. Risk Classification

Severity	Impact: High	Impact: Medium	Impact: Low
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

5.1. Impact

- High - leads to a significant material loss of assets in the protocol or significantly harms a group of users.
- Medium - only a small amount of funds can be lost (such as leakage of value) or a core functionality of the protocol is affected.
- Low - can lead to any kind of unexpected behavior with some of the protocol's functionalities that's not so critical.

5.2. Likelihood

- High - attack path is possible with reasonable assumptions that mimic on-chain conditions, and the cost of the attack is relatively low compared to the amount of funds that can be stolen or lost.
- Medium - only a conditionally incentivized attack vector, but still relatively likely.
- Low - has too many or too unlikely assumptions or requires a significant stake by the attacker with little or no incentive.

5.3. Action required for severity levels

- Critical - Must fix as soon as possible (if already deployed)
- High - Must fix (before deployment if not already deployed)
- Medium - Should fix
- Low - Could fix

6. Security Assessment Summary

review commit hash - c52b6d39ec34f640533c4fd30310e1fba31f5f39

fixes review commit hash - d2e1c2fb65d30591b41fd83771218f1445d1a8ef

Scope

The following smart contracts were in scope of the audit:

- `AgentSetter`
- `OmoRouter`
- `OmoVault`
- `OmoVaultFactory`
- `OmoOracle`
- `DynamicAccount`
- `DynamicAccountFactory`
- `OmoAgent`

7. Executive Summary

Over the course of the security review, 0xunforgiven, Hals, Ch_301 engaged with Omo to review Omo. In this period of time a total of **52** issues were uncovered.

Protocol Summary

Protocol Name	Omo
Repository	https://github.com/omo-protocol/omo-mvp
Date	January 25th 2025 - January 31st 2025
Protocol Type	Asset management

Findings Count

Severity	**Amount**
Critical	7
High	12
Medium	18
Low	15
Total Findings	**52**

Summary of Findings

ID	Title	Severity	Status
[C-01]	Users can duple their first deposit in the vault	Critical	Resolved
[C-02]	Passing the wrong value to _burn() function	Critical	Resolved
[C-03]	Users can't redeem funds due to malicious manipulation of _totalAssets	Critical	Resolved
[C-04]	Funds get locked in the vault	Critical	Resolved
[C-05]	The users are not able to redeem their funds	Critical	Resolved
[C-06]	borrow() should decrease totalAssets value	Critical	Resolved
[C-07]	Unrestricted OmoAgent.onERC721Received() allows permanent DoS and stuck funds	Critical	Resolved
[H-01]	Anyone can redeem from users and take the funds	High	Resolved
[H-02]	The transferFrom() in OmoRouter.sol has approval vulnerabilities	High	Resolved
[H-03]	OmoVault malicious accounts can permanently disable the vault	High	Resolved
[H-04]	Cross-vault misattribution leads to incorrect OmoVaults valuation	High	Resolved
[H-05]	OmoAgent.topOffToAgent incorrect position retrieval logic	High	Resolved

[H-06]	OmoOracle incorrect token decimals handling in getUSDValue and convertUSDToToken	High	Resolved
[H-07]	OmoOracle wrong use of Uniswap interface to get pool address	High	Resolved
[H-08]	OmoOracle getLiquidityAmounts() uses spot price making it manipulatable	High	Resolved
[H-09]	OmoAgent: getPositionValue() doesn't consider deposited assets	High	Resolved
[H-10]	_validateSignature() does not properly handle address(0)	High	Resolved
[H-11]	Pending withdrawal tokens in redeem() affect share price	High	Resolved
[H-12]	getPositionValue() does not consider accumulated fees	High	Resolved
[M-01]	Funds not always in vault lead to share price calculation mess	Medium	Resolved
[M-02]	Entry point contract cannot interact with OmoAgent.sol functions	Medium	Resolved
[M-03]	OmoOracle.convertUSDToToken() : missing price staleness validation	Medium	Resolved
[M-04]	OmoOracle.getUSDValue price feed updates may be incorrectly marked stale	Medium	Resolved
[M-05]	OmoAgent.setPositionManager change results in loss of deposited positions	Medium	Resolved
[M-06]	Incorrect handling of stETH token in OmoRouter.deposit() function	Medium	Resolved

[M-07]	Missing slippage protection in deposit(), mint(), and redeem() functions	Medium	Resolved
[M-08]	Unrestricted access in OmoRouter.transfer() function	Medium	Resolved
[M-09]	OmoRouter.approve() function allows unauthorized token approvals	Medium	Resolved
[M-10]	OmoRouter.mint function unusable due to missing asset handling	Medium	Resolved
[M-11]	OmoRouter: pendingRedeemRequest() calls a non-existent function	Medium	Resolved
[M-12]	OmoRouter registerAccount does not validate msg.sender as owner	Medium	Resolved
[M-13]	OmoVault/OmoRouter: no way to remove whitelist restriction	Medium	Resolved
[M-14]	OmoAgent removeAgent should not remove agent with positive debt	Medium	Resolved
[M-15]	Unbounded loops can lead to out-of-gas errors in contracts	Medium	Acknowledged
[M-16]	OmoVault does not enforce supplyCap	Medium	Resolved
[M-17]	OmoVault first depositor can inflate share price by donating	Medium	Resolved
[M-18]	OmoAgent: topOffToAgent() does not process and remove all positions	Medium	Resolved
[L-01]	Not all tokens return a boolean value	Low	Acknowledged
[L-02]	Protocol can't handle fee-on-transfer tokens	Low	Resolved

[L-03]	Hardcoded address for entry point	Low	Acknowledged
[L-04]	Users can't withdraw native tokens from the Dynamic Account	Low	Acknowledged
[L-05]	OmoVault.redeem() : Incorrect owner recorded in redemption requests	Low	Resolved
[L-06]	Lack of cancellation mechanism for unfulfilled redemption requests in OmoVault	Low	Acknowledged
[L-07]	Missing functionality to unregister malicious agents in OmoVault	Low	Acknowledged
[L-08]	Agent cannot call dynamic account through the entry point	Low	Resolved
[L-09]	OmoAgent.addAgent() : lack of agent whitelist validation	Low	Acknowledged
[L-10]	Missing registered agent validation in OmoVault.borrow() function	Low	Acknowledged
[L-11]	OmoVault.borrow missing receiver validation enables unauthorized transfers	Low	Acknowledged
[L-12]	Users unable to redeem assets if removed from OmoRouter whitelist	Low	Acknowledged
[L-13]	Ineffective whitelisting in OmoRouter contract functions	Low	Acknowledged
[L-14]	OmoVault.borrow() : missing limits and tracking mechanism	Low	Acknowledged
[L-15]	DynamicAccount incorrect argument in validateSignature	Low	Acknowledged

8. Findings

8.1. Critical Findings

[C-01] Users can duple their first deposit in the vault

Severity

Impact: High

Likelihood: High

Description

The `OmoVault.sol` contract implements some principles of the ERC7540 standard, Asynchronous redeem shares for assets. However, users are still able to withdraw directly using `ERC4626.sol#withdraw()`

Also, `_validateSignature()` function in `DynamicAccount.sol` contract accepts the user wallet as a signer, through the Entry Point. So, the user has the same privileges as the agent.

Malicious users can call `withdraw()` from the vault directly to receive their funds. Next, he will transfer all the funds and the UniswapV3 positions out of their Dynamic Account

As a result, he will duple his first deposit in the vault.

Recommendations

1- Override `withdraw()` function in `OmoVault.sol` and make it revert. 2- In `_validateSignature()` you need to check what function can be called if the signer is the owner

[C-02] Passing the wrong value to `_burn()` function

Severity

Impact: High

Likelihood: High

Description

The agent will call `topOff()` function in `OmoVault.sol` contract to transfer the asset to users who have filled the requests to redeem. The user shares are still not burned yet. `topOff()` will burn them here

```
File: OmoVault.sol

453:          _totalAssets -= agentBalance;
454:          _burn(address(this), agentBalance);
```

But it burns the wrong amount of share. because `agentBalance` is the amount of asset that gets transferred to the user.

Recommendations

```
File: OmoVault.sol

453:          _totalAssets -= agentBalance;
-454:          _burn(address(this), agentBalance);
+454:          _burn(address(this), record.shares);
```

[C-03] Users can't redeem funds due to malicious manipulation of `_totalAssets`

Severity

Impact: High

Likelihood: High

Description

The function `OmoVault.sol#topOff()` is only used by agents. It transfers the entire asset balance from the agent to the `record.receiver` address

```
File: OmoVault.sol

449:          asset.safeTransferFrom(msg.sender, address(this), agentBalance);
450:          asset.safeTransfer(record.receiver, agentBalance);
```

Regardless of the fact that we need to trust the agent will set his balance to the exact value. Malicious users can manipulate the value of `agentBalance` by transferring assets directly to the agent address (which will receive it back) But the critical value will be decreased by the wrong value `_totalAssets`.

Take this scenario: in case we have two users, Bob and Alice,

- Both had a deposit 20$
- Bob redeems his 20$,
- Before the agent calls `topOff()` function, Bob transfers to agent 20$
- Agent calls `topOff()` function
- Bob will receive at least 40$ the `_totalAssets` will be 0 Now, Alice will not be able to redeem her 20$ as `_totalAssets -= agentBalance;` will revert

Recommendations

```
File: OmoVault.sol

-449:          asset.safeTransferFrom(msg.sender, address(this), agentBalance);
+449:          asset.safeTransferFrom(msg.sender, address(this), record.assets);
-450:          asset.safeTransfer(record.receiver, agentBalance);
+450:          asset.safeTransfer(record.receiver, record.assets);
```

[C-04] Funds get locked in the vault

Severity

Impact: High

Likelihood: High

Description

In case:

- User X requests to redeem shares for assets and set `receiver` for Y
- Agent Z registered in the Dynamic Account of user X will remove liquidity from Uniswap V3
- Agent Z calls `OmoVault.sol#topOff()` function.

But the current implementation will revert in the above case. Because `topOff()` checks if agent is authorized in the `record.receiver` Dynamic Account.

```
File: OmoVault.sol

427:         address userAccount = userToAccount[record.receiver];
428:         require(userAccount != address(0), "NO_REGISTERED_ACCOUNT");
429:
430:         // Check if agent is registered for this account
431:         OmoAgent agent = OmoAgent(payable(userAccount));
432:         require(agent.agents(0) == msg.sender, "UNAUTHORIZED_AGENT");
```

There is a high chance that the `userAccount` is a zero address or it is not registered. As a result, funds get locked in the vault until `record.receiver` creates a Dynamic Account. and add that the agent

Recommendations

```
File: OmoVault.sol

-427:          address userAccount = userToAccount[record.receiver];
+427:          address userAccount = userToAccount[record.owner];
428:         require(userAccount != address(0), "NO_REGISTERED_ACCOUNT");
```

[C-05] The users are not able to redeem their funds

Severity

Impact: High

Likelihood: High

Description

Dynamic Accounts could have multiple agents, which are stored in `OmoAgent.sol` as library `OmoAgentStorage` under the `agents[]` mapping.

```
File: OmoAgent.sol

18:     struct Data {
19:         address manager;
20:         mapping(uint256 => address) agents;
```

Each agent has a unique id. However, in `OmoVault.sol` contract the `topOff()` function only agents that in `agents[]` can call it

```
File: OmoVault.sol

430:         // Check if agent is registered for this account
431:         OmoAgent agent = OmoAgent(payable(userAccount));
432:         require(agent.agents(0) == msg.sender, "UNAUTHORIZED_AGENT");
```

But it checks only the zero ID (probably it does not exist). With the current implementation, no agent can call `topOff()` and redeem the user's funds.

As a result, the users are not able to redeem their funds.

Recommendations

You should pass the agent ID, not a zero-value.

[C-06] `borrow()` should decrease `totalAssets` value

Severity

Impact: High

Likelihood: High

Description

The `OmoVault.sol` contract has a `borrow()` used only by the agent, which transfers assets to the Agent. It has no tracking system for those debts (it depends on off-chain security). Also, it does not update the `_totalAssets` variable

```
File: OmoVault.sol

403:     function borrow
  (uint256 assets, address receiver) external nonReentrant onlyAgent {
407:         address msgSender = msg.sender;
409:         require(assets != 0, "ZERO_ASSETS");
411:         asset.safeTransfer(receiver, assets);
414:     }
```

This will affect the `totalAssets()` calculation,because The current implementation computes the borrowed assets by the agent two times. But the `totalAssets()` should be the sum of all assets in the vault and all Uni V3 position values in DynamicAccounts.

Recommendations

```
File: OmoVault.sol

403:     function borrow
  (uint256 assets, address receiver) external nonReentrant onlyAgent {
407:         address msgSender = msg.sender;
409:         require(assets != 0, "ZERO_ASSETS");
+410:       _totalAssets =  _totalAssets - assets;
411:         asset.safeTransfer(receiver, assets);
414:     }
```

[C-07] Unrestricted `OmoAgent.onERC721Received()` allows permanent DoS and stuck funds

Severity

Impact: High

Likelihood: High

Description

- The `OmoAgent.onERC721Received()` function is intended to be invoked when an NFT is transferred to the contract using the `safeTransferFrom()` method, however, the `depositPosition()` function incorrectly transfers NFTs using `transferFrom()`, meaning that this hook is never triggered during legitimate deposits:

```
function onERC721Received(
        address,
        address,
        uint256 tokenId,
        bytes calldata
    ) external returns (bytes4) {
        // if (msg.sender == OmoAgentStorage.data().positionManager) {
        _addPositionId(tokenId);
        // }
        return this.onERC721Received.selector;
    }
```

- **Despite this**, `onERC721Received()` is callable by **any** address, leading to:

1. **Permanent DoS of `depositPosition()`**:
   - any attacker can call `onERC721Received()` with arbitrary token IDs.
   - these token IDs will be added to `ownedPositions` and `_positionIds`.
   - when an agent attempts to deposit a valid position ID, the transaction will revert with `"Position already tracked"`.
2. **Out-of-Gas (OOG) due to storage bloat**:
   - attackers can continuously add token IDs, inflating `_positionIds` and `ownedPositions` mappings.
   - this will cause an **OOG error** when looping over `_positionIds` in `topOffToAgent()` and `getPositionValue()`.
3. **Complete breakdown of protocol functionalities**:
   - **Stuck funds**: agents will be unable to retrieve deposited positions due to `topOffToAgent()` failing.
   - **Vault disruption**: if this dynamic account is registered in `OmoVault`, calls to `OmoVault.getAssets()` will fail, permanently disabling the vault, as there's no functionality to remove malicious or compromised agents from the `OmoVault` contract.

Recommendation

Remove `depositPosition()` from `OmoAgent.onERC721Received()`, as the logic of adding an NFT position is handled by `depositPosition()` function.

8.2. High Findings

[H-01] Anyone can redeem from users and take the funds

Severity

Impact: High

Likelihood: Medium

Description

The `redeem()` function in `OmoRouter.sol` contract is triggered by the user to request redeem shares for the equivalent assets. `owner` should set approval for the router address to transfer the shares

```
File: OmoRouter.sol
     function redeem
       (uint256 vaultId,uint256 shares, address receiver,address owner)
     /*code*/
         // Transfer shares from owner to router first
         IOmoVault(vault).transferFrom(owner, address(this), shares);
```

An attacker can drain any allowance for the router address in any vault shares (in case chains have the possibility of front-run, MEV bots can just front-run users `redeem()` transactions) and wait for an agent to do the rest (send the funds).

Recommendations

The owner should only be `msg.sender`

```
File: OmoRouter.sol
-     function redeem
- (uint256 vaultId,uint256 shares, address receiver,address owner)
+     function redeem(uint256 vaultId,uint256 shares, address receiver)
     /*code*/
         // Transfer shares from owner to router first
-         IOmoVault(vault).transferFrom(msg.sender, address(this), shares);
+         IOmoVault(vault).transferFrom(owner, address(this), shares);
```

[H-02] The `transferFrom()` in `OmoRouter.sol` has approval vulnerabilities

Severity

Impact: Medium

Likelihood: High

Description

The `transferFrom()` function in `OmoRouter.sol` contract transfers a certain amount of shares from a sender `from` to a recipient address `to`.

```
File: OmoRouter.sol

File: OmoRouter.sol
     function transferFrom(
       uint256vaultId,
       addressfrom,
       addressto,
       uint256amount
     ) external returns (bool
         address vault = vaultFactory.getVault(vaultId);
         return IOmoVault(vault).transferFrom(from, to, amount);
     }
```

It directly sub-calls to `OmoVault.sol#transferFrom()`. The user needs to call `Approve()` in the vault to allow the router to spend on behalf of `msg.sender` and send it to the recipient address.

Malicious users can steal any amount in `allowance[][]` for the router address from other users or can front-run user transactions that call `transferFrom()#OmoRouter.sol` and transfer all the approved value by the user to his address.

Recommendations

```
File: OmoRouter.sol

-     function transferFrom
- (uint256 vaultId,address from,address to,uint256 amount) external returns (bool) {
+     function transferFrom
+ (uint256 vaultId,address to,uint256 amount) external returns (bool) {
         address vault = vaultFactory.getVault(vaultId);

-         return IOmoVault(vault).transferFrom(from, to, amount);
+         return IOmoVault(vault).transferFrom(msg.sender, to, amount);
     }
```

[H-03] `OmoVault` malicious accounts can permanently disable the vault

Severity

Impact: High

Likelihood: Medium

Description

- The `OmoVault` contract allows whitelisted users to register accounts (`OmoAgent`) without requiring approval from the vault owner or manager. A malicious actor can exploit this by registering an `OmoAgent` account that is configured with an invalid oracle for price retrieval, this would cause the `OmoVault.totalAssets()` function to revert whenever it is called, leading to a **permanent denial of service (DoS) for the vault**:

```
function totalAssets() public view virtual override returns (uint256) {
        //...
        // Sum up values from all registered accounts
        for (uint256 i = 0; i < accountList.length; i++) {
            address account = accountList[i];
            if (registeredAccounts[account]) {
                IDynamicAccount dynamicAcc = IDynamicAccount(account);
                accountHoldings += dynamicAcc.getPositionValue(address(asset));
            }
        }
        //...
    }
```

- Since `totalAssets()` is used to calculate the share value by `deposit()`, `mint()` and `redeem()` functions, this would make the vault unusable as there is **no implemented method to remove malicious accounts (`OmoAgent`) from the vault**, preventing recovery from such an attack.

Recommendations

- Possible mitigation for this issue:

1. Restrict the ability to register accounts in the vault to **only the vault owner or manager**.
2. Implement a function to allow vault owner/manager to **remove malicious or compromised accounts** from the system.
3. Modify `totalAssets()` to **handle oracle failures gracefully**, ensuring that a single failing oracle does not cause the entire function to revert.

[H-04] Cross-vault misattribution leads to incorrect `OmoVaults` valuation

Severity

Impact: High

Likelihood: Medium

Description

- The `OmoAgent` agents will interact with multiple `OmoVault` vaults, borrowing assets and depositing liquidity positions back into the dynamic account (`OmoAgent`), however, the `OmoVault.totalAssets()` function accounts for **all** liquidity positions deposited in the `OmoAgent`, without tracking whether these positions were generated from assets borrowed from that specific vault.

```
function totalAssets() public view virtual override returns (uint256) {
        uint256 vaultHoldings = _totalAssets;
        uint256 accountHoldings;

        // Sum up values from all registered accounts
        for (uint256 i = 0; i < accountList.length; i++) {
            address account = accountList[i];
            if (registeredAccounts[account]) {
                IDynamicAccount dynamicAcc = IDynamicAccount(account);
                accountHoldings += dynamicAcc.getPositionValue(address(asset));
            }
        }

        return vaultHoldings + accountHoldings;
    }
```

- This leads to **overvaluation** of vault shares, as `totalAssets()` will include the value of liquidity positions that originated from borrowed assets of **different vaults**, inflating the share price and creating an inaccurate valuation of the vault's holdings.

Recommendation

1. Implement a tracking mechanism in the `OmoAgent` contract to associate deposited liquidity positions with their **corresponding vault (`vaultId`)**, ensuring that each position is linked to the vault from which the assets used to open it were borrowed.
2. Modify `OmoVault.totalAssets()` to **only account for positions created from assets borrowed from that specific vault** rather than summing all positions in the `OmoAgent`.

[H-05] `OmoAgent.topOffToAgent` incorrect position retrieval logic

Severity

Impact: High

Likelihood: Medium

Description

- The `OmoAgent.topOffToAgent()` function allows an agent to retrieve their deposited position NFT, however, there are multiple issues in the current implementation:

```
function topOffToAgent(uint256 _agentId, address asset) external onlyAgent
  (_agentId) {
        INonfungiblePositionManager nftManager = INonfungiblePositionManager
          (OmoAgentStorage.data().positionManager);
        for (uint256 i = 0; i < OmoAgentStorage.data
          ()._positionIds.length; i++) {
            uint256 tokenId = OmoAgentStorage.data()._positionIds[i];
            require(nftManager.getApproved(tokenId) == OmoAgentStorage.data
              ().agents[_agentId], "Not approved");

            nftManager.transferFrom(address(this), OmoAgentStorage.data
              ().agents[_agentId], tokenId);
            _removePositionId(tokenId);
        }

        uint256 amount = IERC20(asset).balanceOf(address(this));
        if(amount > 0) {
            IERC20(asset).transfer(OmoAgentStorage.data
              ().agents[_agentId], amount);
        }

        emit TopOffToAgent(_agentId);
    }
```

1. **Incorrect loop-based validation:**

   - The function loops over **all the positions owned by the contract** and checks whether the caller (agent) is approved on each of the owned positions.
   - If the agent is not approved on any one of them, the **call reverts**.
   - This assumption is inaccurate because **multiple agents interact with the `DynamicAccount`**, and there is **no tracking mechanism** to associate each deposited position with the agent who deposited it.

2. **Unnecessary approval check:**

   - The function verifies that the **agent is approved on the position ID** before transferring it back,however, there is **no implemented function in the contract that allows the DynamicAccount to make external calls** to grant NFT approvals to specific agents.
   - This check **cannot be enforced correctly** and **prevents agents from retrieving their deposited positions**.

Recommendation

- Implement tracking for each deposited position by **mapping each position ID to the agent who deposited it**.
- Remove the `nftManager.getApproved(tokenId) == OmoAgentStorage.data().agents[_agentId]` check, as it is unnecessary when the position is being **transferred back to its original depositor**.

[H-06] OmoOracle incorrect token decimals handling in `getUSDValue` and `convertUSDToToken`

Severity

Impact: High

Likelihood: Medium

Description

The `getUSDValue()` function in the OmoOracle contract does not consider token decimals when converting amounts. It assumes all tokens have the same number of decimals (1e18), which can lead to incorrect conversions and amounts calculations. The function uses the following approach:

```
function getUSDValue(address token, uint256 amount) public view returns
      (uint256) {
        --snip--
        uint8 decimals = priceFeed.decimals();
        uint256 normalizedPrice = uint256(price) * 1e18 / 10**decimals;

        return (amount * normalizedPrice) / 1e18;
    }
```

Recommendations

The correct way to handle asset amount conversions, considering their decimals, is to adjust for the respective decimals of each token involved in the conversion. The formula should look like this:

```
amountUSD = (amountToken * priceTokenToUSD * decimalsUSD) /
  (decimalsToken * 1e18);
```

[H-07] OmoOracle wrong use of Uniswap interface to get pool address

Severity

Impact: High

Likelihood: Medium

Description

The function `getLiquidityAmounts()` incorrectly uses the Uniswap interface to obtain the pool address:

```
function getLiquidityAmounts(
        address positionManager,
        uint256 tokenId,
        uint128 liquidity
    ) internal view returns (uint256 amount0, uint256 amount1) {
        if (liquidity == 0) return (0, 0); // Handle zero liquidity case
        INonfungiblePositionManager nftManager = INonfungiblePositionManager
          (positionManager);

        --snip--

        IUniswapV3Pool pool = IUniswapV3Pool(nftManager.factory
        //()); //Wrong pool address
        (uint160 sqrtPriceX96,,,,,,) = pool.slot0();

        // Calculate amounts using UniswapV3 math
        (amount0, amount1) = LiquidityAmounts.getAmountsForLiquidity(
            sqrtPriceX96,
            TickMath.getSqrtRatioAtTick(tickLower),
            TickMath.getSqrtRatioAtTick(tickUpper),
            liquidity
        );
    }
```

Recommendations

Use correct interface to get pool address

To resolve this, get the pool that belongs to `tokenId`, you need to catch the `token0`, `token1` and `fee` from this call `nftManager.positions(tokenId);` and use it like this

```
File: OmoOracle.sol#getLiquidityAmounts()

         INonfungiblePositionManager nftManager = INonfungiblePositionManager
           (positionManager);
         /*code*/

-        IUniswapV3Pool pool = IUniswapV3Pool(nftManager.factory());
+        address poolAddress = IUniswapV3Factory(nftManager.factory()).getPool
+ (token0, token1, fee);
+        IUniswapV3Pool pool = IUniswapV3Pool(poolAddress);
        (uint160 sqrtPriceX96,,,,,,) = pool.slot0();
```

[H-08] OmoOracle `getLiquidityAmounts()` uses spot price making it manipulatable

Severity

Impact: High

Likelihood: Medium

Description

`getPositionValue()` / `getLiquidityAmounts()` in `OmoOracle` rely on the spot price on Uniswap to calculate token amounts in a position. It makes the system vulnerable to **price manipulation**.

```
function getLiquidityAmounts(
        address positionManager,
        uint256 tokenId,
        uint128 liquidity
    ) internal view returns (uint256 amount0, uint256 amount1) {

--snip--

        IUniswapV3Pool pool = IUniswapV3Pool(nftManager.factory());
        (uint160 sqrtPriceX96,,,,,,) = pool.slot0();

        // Calculate amounts using UniswapV3 math
        (amount0, amount1) = LiquidityAmounts.getAmountsForLiquidity(
            sqrtPriceX96,
            TickMath.getSqrtRatioAtTick(tickLower),
            TickMath.getSqrtRatioAtTick(tickUpper),
            liquidity
        );
    }
```

Recommendations

Use Time-Weighted Average Price (TWAP) Instead

[H-09] OmoAgent: `getPositionValue()` doesn't consider deposited assets

Severity

Impact: High

Likelihood: Medium

Description

Function `totalAssets()` calls `getPositionValue()` for all the dynamic accounts to calculate the total assets. In the `OmoAgent` contract, the `getPositionValue()` function does not take into account the deposited assets when calculating the position value. This leads to an incorrect calculation of `totalAssets()` in `OmoVault`, which can result in potential loss of funds for users due to inaccurate asset tracking.
Note: agents can deposit assets (not just positions) into `OmoAgent` by `depositAssets()` function

Recommendations

Modify `getPositionValue()` to include the value of deposited assets when computing the total position value.

[H-10] `_validateSignature()` does not properly handle `address(0)`

Severity

Impact: High

Likelihood: Medium

Description

The current implementation of `_validateSignature()` uses `ECDSA.recover()` to extract the signer address, which can return `address(0)` for invalid

signatures:

```
function _validateSignature(
        UserOperation calldata userOp,
        bytes32 userOpHash
    ) internal virtual override returns (uint256 validationData) {
        bytes32 hash = ECDSA.toEthSignedMessageHash(userOpHash);
        address signer = ECDSA.recover(hash, userOp.signature);

        // Check if signer is admin (user wallet)
        if (isAdmin(signer)) return 0;

        // Check if signer is an authorized agent
        if (OmoAgent(payable(address(this))).agents(0) == signer) {
            return 0;
        }
        return SIG_VALIDATION_FAILED;
    }
```

However, if `agent(0)` is not set (i.e., `address(0)`), the function might incorrectly pass the signature validity check even though the signature is invalid. This is because it doesn't account for the possibility that the recovered address could be `address(0)` and mistakenly treats it as a valid signer. This could lead to security issues where invalid signatures are accepted.

Recommendations

Explicitly check for `address(0)` in the recovered address and ensure that no invalid signatures are accepted. If the recovered address is `address(0)`, the signature should be considered invalid.

Update the `DynamicAccount._validateSignature()` function to check that the recovered signer is not `address(0)`:

```
function _validateSignature(
       UserOperation calldata userOp,
       bytes32 userOpHash
   ) internal virtual override returns (uint256 validationData) {
       bytes32 hash = ECDSA.toEthSignedMessageHash(userOpHash);
       address signer = ECDSA.recover(hash, userOp.signature);
+       if (signer == address(0)) return SIG_VALIDATION_FAILED;
       //...
   }
```

[H-11] Pending withdrawal tokens in `redeem()` affect share price

Severity

Impact: High

Likelihood: Medium

Description

Pending withdrawal tokens in the `redeem()` function are still considered in the share price calculations. Since `totalSupply` and `_totalAssets` are not updated to reflect the pending withdrawals:

```
// Transfer shares from owner to vault
        balanceOf[owner] -= shares;
        balanceOf[address(this)] += shares;
```

The share price would be incorrect for profit/loss that happens later. When the pending withdrawals are finalized (by calling `topOff`), the share price will adjust to account for the withdrawals that were not previously considered. This could lead to inconsistencies in share valuation, causing wrong fund distributions among users over time.

Recommendations

Update the share price calculation to exclude pending withdrawal tokens until they are finalized. This will ensure that `totalSupply` and `totalAssets` accurately reflect the current state of the vault and provide an accurate share price.

[H-12] `getPositionValue()` does not consider accumulated fees

Severity

Impact: High

Likelihood: Medium

Description

The function `getPositionValue()` only retrieves the liquidity token amounts using `getLiquidityAmounts()` and converts them into the quote token value. But it does not account for accumulated fees.

```
function getPositionValue(
        address positionManager,
        uint256 tokenId,
        address quoteToken
    ) external view override returns (uint256) {
        // Get position details
        (
          ,
          ,
          addresstoken0,
          addresstoken1,
          ,
          ,
          ,
          uint128liquidity,
          ,
          ,
          ,

        ) = INonfungiblePositionManager(positionManager

        // Get amounts
        (uint256 amount0, uint256 amount1) = getLiquidityAmounts
          (positionManager, tokenId, liquidity);

        // Convert to quote token value
        uint256 value0 = convertTokenValue(token0, amount0, quoteToken);
        uint256 value1 = convertTokenValue(token1, amount1, quoteToken);

        return value0 + value1;
    }
```

Note: Uniswap V3 positions earn fees over time, and these are claimable but not included in the liquidity calculation.

Recommendations

- Include Accumulated Fees in Position Value Calculation
  - You can use `tokensOwed0` and `tokensOwed1` in response of `positions(tokenId)` to calculate how many uncollected tokens are owed to the position, as of the last computation

- Use `collect` function to collect fees for a specific position Uniswap Doc

8.3. Medium Findings

[M-01] Funds not always in vault lead to share price calculation mess

Severity

Impact: High

Likelihood: Low

Description

At any point, the funds could be outside of `OmoVault.sol` and still not a UniswapV3 position in dynamic accounts. the funds could be in the agent address, in dynamic accounts as tokens (not UniV3 positions) or UniswapV3 positions but still owned by the agent.

This will affect the return value of the total assets in the vault from `OmoVault.so#totalAssets()` and it will corrupt all the logic of deposit/mint and redeem.

Recommendations

The only fix I can find (but not efficient) is to lock the vault in this case.

[M-02] Entry point contract cannot interact with `OmoAgent.sol` functions

Severity

Impact: Medium

Likelihood: Medium

Description

Most of the functions in `OmoAgent.sol` contract will be triggered by the entry point contract e.g.`borrowAsset()`, `repayAsset()`, `topOffToAgent()` and more. But they have an `onlyAgent` modifier

```
modifier onlyAgent(uint256 _id) {
        require(msg.sender == OmoAgentStorage.data().agents[_id], "Only agent");
        _;
    }
```

This will prevent the standard's intended behavior of ERC4337 (limiting interactions with the EntryPoint).

Recommendations

The struct `Data` in the library `OmoAgentStorage` should have the address of `EntryPoint` and create a check similar to `_requireFromEntryPoint()`

[M-03] `OmoOracle.convertUSDToToken()` : missing price staleness validation

Severity

Impact: Medium

Likelihood: Medium

Description

The `OmoOracle.convertUSDToToken()` function fetches the price of `toToken` from the price feed but does not check if the price is fresh, which would result in using a stale price of the `quoteToken` leading to incorrect valuation of assets held by agents:

```
function convertUSDToToken
  (uint256 usdAmount, address token) public view returns (uint256) {
        //...
        (, int256 price,,,) = priceFeed.latestRoundData();

        if (price <= 0) revert InvalidPriceFeed();

        //...
    }
```

Recommendations

Implement a price staleness check similar to `getUSDValue()`, ensuring that `block.timestamp - updatedAt` is within an acceptable range based on the heartbeat of the price feed.

[M-04] `OmoOracle.getUSDValue` price feed updates may be incorrectly marked stale

Severity

Impact: Medium

Likelihood: Medium

Description

The `OmoOracle.getUSDValue()` function enforces a hardcoded staleness check where the price is considered stale if `block.timestamp - updatedAt > 1 hours`, however, different price feeds have different heartbeats, and price updates do not necessarily occur within a fixed hourly interval.
Instead, aggregators update the price based on deviation thresholds or when the last update time exceeds their specified heartbeat, while the current implementation may cause unnecessary reverts for valid price feeds that have a longer update interval (heartbeat > 1 hour):

```
function getUSDValue(address token, uint256 amount) public view returns
  (uint256) {
        //...
    if (block.timestamp - updatedAt > 1 hours) revert StalePrice();
        //...
    }
```

Recommendations

Modify the staleness check to compare against the specific price feed's configured heartbeat rather than a fixed 1-hour threshold.

[M-05] `OmoAgent.setPositionManager` change results in loss of deposited positions

Severity

Impact: Medium

Likelihood: Medium

Description

The `setPositionManager()` function allows the manager to update the `positionManager` address, however, changing the position manager without proper handling will result in all previously deposited positions being lost, since deposited positions are tied to the previous position manager, they will become inaccessible after the update:

```
function setPositionManager(address _positionManager) external onlyManager {
        OmoAgentStorage.data().positionManager = _positionManager;
    }
```

Recommendation

To prevent loss of deposited positions when updating the position manager:

- Implement a check to ensure that no positions are currently deposited before allowing an update.
- Alternatively, track each deposited position along with the `positionManager` address used at the time of deposit, so that previously deposited positions remain accessible.

[M-06] Incorrect handling of `stETH` token in `OmoRouter.deposit()` function

Severity

Impact: Medium

Likelihood: Medium

Description

- The `OmoRouter.deposit()` function doesn't correctly handle the `stETH` token, which is one of the LST tokens supported as an underlying asset in `OmoVaults`, the issue arises because `stETH` transfers **1-2 wei less than the amount specified during a `safeTransferFrom` operation**.

- This leads to the `OmoRouter` contract approving and attempting to deposit more tokens than it has received, causing the subsequent `OmoVault.deposit()` call to revert due to insufficient contract balance (received is less than deposited).

```
function deposit(
        uint256 vaultId,
        uint256 assets,
        address receiver
    ) external onlyWhitelisted returns (uint256 shares) {
        address vault = vaultFactory.getVault(vaultId);
        if (!isVaultEnabled[msg.sender][vault]) revert VaultNotEnabled();

        // Get the underlying token using IERC4626
        ERC20 token = ERC20(IERC4626(vault).asset());

        // Transfer tokens from user to router
        token.safeTransferFrom(msg.sender, address(this), assets);

        // Approve vault to spend tokens
        token.safeApprove(vault, assets);

        // Deposit into vault
        shares = IOmoVault(vault).deposit(assets, receiver);

        return shares;
    }
```

- The same issue exists in:
  - `OmoVault.deposit()` function when whitelisted users interact directly with the function.
  - `OmoVault.topOff()` and `OmoAgent.repayAsset()` functions.

Recommendations

Update `OmoRouter.deposit()` to deposit the actual amount received:

```
function deposit(
        uint256 vaultId,
        uint256 assets,
        address receiver
    ) external onlyWhitelisted returns (uint256 shares) {
        address vault = vaultFactory.getVault(vaultId);
        if (!isVaultEnabled[msg.sender][vault]) revert VaultNotEnabled();

        // Get the underlying token using IERC4626
        ERC20 token = ERC20(IERC4626(vault).asset());

        // Transfer tokens from user to router
+       uint256 balanceBefore = token.balanceOf(address(this));
        token.safeTransferFrom(msg.sender, address(this), assets);
+       uint256 amountToDeposit = token.balanceOf(address
+ (this)) - balanceBefore;
        // Approve vault to spend tokens
-       token.safeApprove(vault, assets);
+       token.safeApprove(vault, amountToDeposit);

        // Deposit into vault
-       shares = IOmoVault(vault).deposit(assets, receiver);
+       shares = IOmoVault(vault).deposit(amountToDeposit, receiver);

        return shares;
    }
```

[M-07] Missing slippage protection in `deposit()`, `mint()`, and `redeem()` functions

Severity

Impact: Medium

Likelihood: Medium

Description

- The `deposit()`, `mint()`, and `redeem()` functions in the `OmoRouter` contract lack a slippage mechanism to safeguard against significant deviations in the assets received or shares minted, which would result in users getting less assets/shares and incur unexpected losses during these operations.
- The same issue exists when whitelisted users interact directly with the `OmoVault.deposit()`, `OmoVault.mint()`, and `OmoVault.redeem()` functions.

Recommendation

Implement slippage protection mechanisms in the vault contract itself rather than in the `OmoRouter` contract, by introducing a parameter for acceptable slippage (minimum assets received or maximum shares burnt when redeeming) that users can specify when interacting with the vault.

[M-08] Unrestricted access in `OmoRouter.transfer()` function

Severity

Impact: Medium

Likelihood: Medium

Description

The `OmoRouter.transfer()` function allows any user to transfer any amount of vault shares from the router balance to themselves, which could lead to token theft if the router has any balance:

```
function transfer(
        uint256 vaultId,
        address to,
        uint256 amount
    ) external returns (bool) {
        address vault = vaultFactory.getVault(vaultId);
        return IOmoVault(vault).transfer(to, amount);
    }
```

Recommendations

Restrict the `transfer()` function to the contract owner or manager, or it can be removed.

[M-09] `OmoRouter.approve()` function allows unauthorized token approvals

Severity

Impact: Medium

Likelihood: Medium

Description

- The `OmoRouter.approve()` function allows any user to approve themselves on the vault share tokens of the router contract, so users can grant themselves approvals, potentially leading to unauthorized token transfers or misuse (if the router has any shares balance):

```
function approve(
        uint256 vaultId,
        address spender,
        uint256 amount
    ) external returns (bool) {
        address vault = vaultFactory.getVault(vaultId);
        return IOmoVault(vault).approve(spender, amount);
    }
```

Recommendation

Remove the `approve()` function from the `OmoRouter` contract.

[M-10] `OmoRouter.mint` function unusable due to missing asset handling

Severity

Impact: Medium

Likelihood: Medium

Description

The `OmoRouter.mint()` function is intended to call `IOmoVault(vault).mint()`, where the vault pulls assets from the router and mints shares for the `receiver, however, the implementation is incomplete and lacks key steps to facilitate this process:

1. The function does not pull assets from the user.
2. The function does not approve the vault to spend the deposited assets.

as a result, the vault will be unable to pull the required assets, rendering the `mint()` function unusable.

```
// OmoRouter
  function mint(
        uint256 vaultId,
        uint256 shares,
        address receiver
    ) external onlyWhitelisted returns (uint256 assets) {
        address vault = vaultFactory.getVault(vaultId);
        if (!isVaultEnabled[msg.sender][vault]) revert VaultNotEnabled();
        return IOmoVault(vault).mint(shares, receiver);
    }
```

Recommendation

Update `OmoRouter.mint()` function to pull the assets that will be deposited from the user, and approve the `OmoVault` to spend the deposited assets before calling `IOmoVault(vault).mint()`:

```
function mint(
    uint256 vaultId,
    uint256 shares,
    address receiver
) external onlyWhitelisted returns (uint256 assets) {
    address vault = vaultFactory.getVault(vaultId);
    if (!isVaultEnabled[msg.sender][vault]) revert VaultNotEnabled();

+   ERC20 token = ERC20(IERC4626(vault).asset());

+   uint256 initialBalance = token.balanceOf(address(this));
+   token.safeTransferFrom(msg.sender, address(this), shares);
+   uint256 receivedAssets = token.balanceOf(address(this)) - initialBalance;

+   token.safeApprove(vault, receivedAssets);

    return IOmoVault(vault).mint(shares, receiver);
}
```

[M-11] OmoRouter: `pendingRedeemRequest()` calls a non-existent function

Severity

Impact: Medium

Likelihood: Medium

Description

The function `pendingRedeemRequest()` tries to call `OmoVault.pendingRedeemRequest()`, but this function **does not exist in OmoVault**. This will cause the transaction to revert.

```
function pendingRedeemRequest(
        uint256 vaultId,
        uint256 requestId,
        address owner
    ) external view returns (uint256 pendingShares) {
        address vault = vaultFactory.getVault(vaultId);
        return IOmoVault(vault).pendingRedeemRequest(requestId, owner);
    }
```

Recommendations

add `pendingRedeemRequest` function to `OmoVault`.

[M-12] OmoRouter `registerAccount` does not validate `msg.sender` as owner

Severity

Impact: Medium

Likelihood: Medium

Description

The `registerAccount()` function does not check if `msg.sender` is the owner of the `account` being registered. This means another user can call it and register the account in `OmoVault` for himself.

```
function registerAccount
      (uint256 vaultId, address account) external onlyWhitelisted {

        address[] memory accountAddress = accountFactory.getAllAccounts();

        bool accountExists = false;

        for (uint256 i = 0; i < accountAddress.length; i++) {

            if (accountAddress[i] == account) {

                accountExists = true;

                break;

            }

        }

        if (!accountExists) {

            revert AccountNotRegistered();

        }

        IOmoVault(vaultFactory.getVault(vaultId)).registerAccount
          (msg.sender, account);

    }
```

`OmoVault.registerAccount` function:

```
function registerAccount
      (address user, address account) external onlyRouterOrOwner() {

        if (!registeredAccounts[account]) {

            registeredAccounts[account] = true;

            userToAccount[user] = account;

            accountList.push(account);

            emit AccountRegistered(user, account);

        }

    }
```

Recomendations

Validate `msg.sender` before registering account in vault.

[M-13] OmoVault/OmoRouter: no way to remove whitelist restriction

Severity

Impact: Medium

Likelihood: Medium

Description

Right now, the `onlyWhitelisted` modifier only allows approved addresses to call vault functions. The documentation says that the whitelist system will be removed in the future, but there is no functionality to do this in the code.

Recommendations

A simple `whitelistEnabled` flag can let the owner turn off the whitelist when needed:

```
bool public whitelistEnabled = true;

modifier onlyWhitelisted() {
    require(!whitelistEnabled || whitelisted[msg.sender], "Not whitelisted");
    _;
}

// Owner can turn off the whitelist
function disableWhitelist() external onlyOwner {
    whitelistEnabled = false;
}
```

[M-14] OmoAgent `removeAgent` should not remove agent with positive debt

Severity

Impact: Medium

Likelihood: Medium

Description

The `removeAgent()` function currently allows the removal of an agent even if the agent has a positive debt. This could lead to an inconsistent or incorrect state in contract:

```
function removeAgent(uint256 _id) external onlyManager {
        delete OmoAgentStorage.data().agents[_id];
    }
```

Recommendations

Check `agentDebts` before removing an agent.

[M-15] Unbounded loops can lead to out-of-gas errors in contracts

Severity

Impact: Medium

Likelihood: Medium

Description

1. The `totalAssets()` function in `OmoVault.sol` contains an unbounded loop that iterates through all registered accounts:
2. `OmoRouter.registerAccount` loops through all accounts, making it susceptible to a DoS attack via createAccount().
3. `OmoVaulFactory.getVaultId` loops through all vaults...
4. `OmoAgent.topOffToAgent` has unbounded loop when tries to remove `positionId`
5. `OmoAgent._removeTokenAddress` loops through all token addresses
6. `AgentSetter.removeAgent` loops through all `whitelistedAgents` which is unbounded

Recommendation

Consider adding maximum limits or use mappings instead of arrays for storage

[M-16] OmoVault does not enforce supplyCap

Severity

Impact: Low

Likelihood: High

Description

The `OmoVault.sol` contract has a `supplyCap` state variable and setter function, but this cap is not enforced in the `deposit()` or `mint()` functions.

```
function deposit(
        uint256 assets,
        address receiver
    ) public virtual override onlyWhitelisted returns (uint256 shares) {
        address msgSender = msg.sender;

        // Check for rounding error since we round down in previewDeposit.
        require((shares = _convertToShares(assets, false)) != 0, "ZERO_SHARES");
        // require(supplyCap >= totalAssets() + assets, "SUPPLY_CAP_EXCEEDED");
        // Need to transfer before minting or ERC777s could reenter.
        asset.safeTransferFrom(msgSender, address(this), assets);

        _totalAssets += assets;

        _mint(receiver, shares);

        emit Deposit(msgSender, receiver, assets, shares);
    }
```

```
function mint(
        uint256 shares,
        address receiver
    ) public virtual override onlyWhitelisted returns (uint256 assets) {
        address msgSender = msg.sender;

        assets = _convertToAssets
        //(shares, true); // No need to check for rounding error, previewMint rounds u

        // Need to transfer before minting or ERC777s could reenter.
        asset.safeTransferFrom(msgSender, address(this), assets);

        _totalAssets += assets;

        _mint(receiver, shares);

        emit Deposit(msgSender, receiver, assets, shares);
    }
```

Recommendations

Implement the supply cap check in both `deposit()` and `mint()` functions

[M-17] OmoVault first depositor can inflate share price by donating

Severity

Impact: High

Likelihood: Low

Description

`OmoVault` is vulenarable to the *inflation attack*: The function `_convertToShares()` determines share price using `totalAssets()`, which includes both `vaultHoldings` and `accountHoldings`. A malicious user can inflate share price by donating assets to their own registered account, increasing their `accountHoldings`.

```
function totalAssets() public view virtual override returns (uint256) {
        uint256 vaultHoldings = _totalAssets;
        uint256 accountHoldings;

        // Sum up values from all registered accounts
        for (uint256 i = 0; i < accountList.length; i++) {
            address account = accountList[i];
            if (registeredAccounts[account]) {
                IDynamicAccount dynamicAcc = IDynamicAccount(account);
                accountHoldings += dynamicAcc.getPositionValue(address(asset));
            }
        }

        return vaultHoldings + accountHoldings;
    }
```

Recommendations

Different approaches (like creating dead shares) can be found on this thread: link

[M-18] OmoAgent: `topOffToAgent()` does not process and remove all positions

Severity

Impact: Medium

Likelihood: Medium

Description

In the `topOffToAgent()` function, positions are removed from the `_positionIds` array while iterating over it.

```
for (uint256 i = 0; i < OmoAgentStorage.data
          ()._positionIds.length; i++) {
            uint256 tokenId = OmoAgentStorage.data()._positionIds[i];
            require(nftManager.getApproved(tokenId) == OmoAgentStorage.data
              ().agents[_agentId], "Not approved");

            nftManager.transferFrom(address(this), OmoAgentStorage.data
              ().agents[_agentId], tokenId);
            _removePositionId(tokenId);
        }
```

As the loop index `i` increases, calling `_removePositionId()` would shift the array and as result iteration will not go over all the items.

Recommendations

Transfer tokens inside the loop and delete remove the positions in another loop.

8.4. Low Findings

[L-01] Not all tokens return a boolean value

The protocol will support stable tokens. some of them will not return a boolean value, e.g.USDT.

This will lead `depositAssets()` function in `OmoAgent.sol` contract to keep reverting due to the require check for every `transferFrom()` call.

```
File: OmoAgent.sol

     function depositAssets(
       addressasset,
       uint256amount,
       uint256_agentId
     ) external onlyAgent(_agentId
        require(IERC20(asset).transferFrom(msg.sender, address
          (this), amount), "Transfer failed");
```

[L-02] Protocol can't handle fee-on-transfer tokens

The protocol will support stable tokens e.g.USDT, USDC. some of them are fee-on-transfer tokens.

However, The `deposit()` function in the `OmoRouter.sol` contract, Transfers tokens from the user to the router and then will be deposited into the vault.

```
File: OmoRouter.sol#deposit()

         token.safeTransferFrom(msg.sender, address(this), assets);
         token.safeApprove(vault, assets);
         shares = IOmoVault(vault).deposit(assets, receiver);
```

The issue here is the `assets` value will be not available here.

This issue can be fixed as follows: checking the contract balance before and after, and passing the difference to the `deposit()` function.

[L-03] Hardcoded address for entry point

The entry point address is hardcoded in the `DynamicAccountFactory.sol` contract.

```
File: DynamicAccountFactory.sol

        address public constant ENTRYPOINT_ADDRESS = 0x0000000071727De22E5E9d8BAf0edAc6
```

This `ENTRYPOINT_ADDRESS` address has nothing in it on the **hyper EVM** chain.

[L-04] Users can't withdraw native tokens from the Dynamic Account

The owner of Dynamic Account needs to send native tokens to his account, to allow EntryPoint to pay the necessary gas through `BaseAccount.sol#_payPrefund()` when it calls `validateUserOp()`, But, if the owner stops using this protocol he can't withdraw his native token. This can be resolved by creating a function to withdraw native tokens from the Dynamic Account.

[L-05] `OmoVault.redeem()` : Incorrect owner recorded in redemption requests

- The `OmoVault.redeem()` function allows a caller to create a redemption request on behalf of a share owner. However, the request incorrectly records the `owner` as the caller (`msg.sender`) instead of the actual owner of the shares being redeemed, which can lead to incorrect tracking of ownership and causing issues later when this address is required for validation if a redemption cancellation mechanism is implemented in the future:

```
function redeem(
        uint256 shares,
        address receiver,
        address owner
    ) public virtual override onlyWhitelisted returns (uint256 assets) {
        address msgSender = msg.sender;
        //...
        redemptionRecords[nextRedemptionId] = RedemptionRecord({
            owner: msgSender,
            receiver: receiver,
            shares: shares,
            assets: assets,
            settled: false
        });
        //...
    }
```

- Recommendation: update the redemption request to properly record the **real owner** of the shares instead of the caller (`msg.sender`).

[L-06] Lack of cancellation mechanism for unfulfilled redemption requests in `OmoVault`

- The `OmoVault` contract implements an asynchronous redemption process where users create a redemption request to withdraw their deposited assets, these requests are fulfilled later by a whitelisted agent, however, the contract doesn't provide a mechanism for users to cancel their pending redemption requests if they remain unfulfilled, which would result in depositors' assets remain locked in the vault, as they do not hold their vault shares either.

- Additionaly, the `OmoVault` fails to align with the recommendations of **EIP-7540**, which suggests including a cancellation mechanism for unfulfilled redemption requests under its security considerations.

- Recommendation: implement a function to allow depositors to cancel their pending redemption requests if they have not yet been fulfilled.

[L-07] Missing functionality to unregister malicious agents in `OmoVault`

- The `OmoVault` contract allows agents to be registered, granting them specific privileges when interacting with the vault, however, there is no functionality to unregister an agent if they begin acting maliciously, which exposes the vault to potential risks, as malicious agents could manipulate the value of their deposited positions to intentionally overvaluating or undervaluating the vault vaule; affecting the vault's share value.

- Recommendation: implement a functionality to unregister agents, allowing the vault's owner to remove a registered agents if they act maliciously.

[L-08] Agent cannot call dynamic account through the entry point

Agent can't call Dynamic Account through the Entry Point because the `_validateSignature()` function in `DynamicAccount.sol`

```
File: DynamicAccount.sol
   function _validateSignature(
     /*code*/
        if (OmoAgent(payable(address(this))).agents(0) == signer) {
             return 0;
        }
```

Always check if the signer is the agent under the `id` zero. As a result, all the agent transactions will revert.

Recommendations:

Add a mechanism to check by signer address or pass the correct `id`

[L-09] `OmoAgent.addAgent()` : lack of agent whitelist validation

The `OmoAgent.addAgent()` function allows the manager to set agents for the dynamic account, these agents are expected to interact with the `OmoVault` contract when borrowing assets or performing `topOff()`, however, there is no

validation to check if the added agent **is a whitelisted agent in the `AgentSetter` contract**:

```
function addAgent(address _agent, uint256 _id) external onlyManager {
        OmoAgentStorage.data().agents[_id] = _agent;
    }
```

While the `OmoVault.onlyAgent` modifier ensures that only whitelisted agents interact with `borrow()` and `topOff()`, the lack of validation in `addAgent()` could result in non-whitelisted agents being assigned to the dynamic account.

Recommendation:

Modify the `OmoVault.addAgent()` function to validate that the `_agent` address is whitelisted in the `AgentSetter` contract before adding it to `OmoAgentStorage.data().agents`.

[L-10] Missing registered agent validation in `OmoVault.borrow()` function

The `OmoVault.borrow()` function is designed to allow whitelisted agents to borrow assets from the vault for reinvestment and yield generation, it accepts a `receiver` parameter, which determines the account that will receive the borrowed assets.

```
function borrow(
    uint256 assets,
    address receiver
) external nonReentrant onlyAgent {
    address msgSender = msg.sender;
    require(assets != 0, "ZERO_ASSETS");

    asset.safeTransfer(receiver, assets);

    emit Borrow(msgSender, receiver, assets);
}
```

While the function ensures that the caller is a whitelisted agent; **it does not verify whether the agent is registered as an authorized agent for the `receiver` account**, so any agent could borrow on behalf of another `DynamicAccount` even if they are not a registered agent for that account.

Since the protocol relies on `DynamicAccount` wallets, where agents manage assets on behalf of users, failing to enforce this validation could allow

unauthorized agents to execute financial operations on accounts they do not control.

Recommendations:

Implement a check in the `OmoVault.borrow()` function to ensure that the borrowing agent is a registered agent for the specified receiver.

[L-11] `OmoVault.borrow` missing receiver validation enables unauthorized transfers

The `OmoVault.borrow()` function allows whitelisted agents to borrow assets from the vault to reinvest and generate yield, the function accepts a `receiver` parameter, which determines the address that will receive the borrowed assets, however, there is no validation to ensure that the `receiver` is a registered account, which allows borrowing assets for unauthoized accounts:

```
function borrow(
        uint256 assets,
        address receiver
    ) external nonReentrant onlyAgent {
        address msgSender = msg.sender;
        require(assets != 0, "ZERO_ASSETS");

        asset.safeTransfer(receiver, assets);

        emit Borrow(msgSender, receiver, assets);
    }
```

Update the `OmoVault.borrow()` function to check that the receiver is a registered account before transferring borrowed assets.

[L-12] Users unable to redeem assets if removed from `OmoRouter` whitelist

The `OmoRouter.redeem()` function uses the `onlyWhitelisted` modifier to check if the user is whitelisted before allowing them to redeem their deposited assets from the vault, however, if a user is removed from the whitelist, they lose the ability to withdraw their previously deposited assets, which violates the principle of ensuring users can always access their funds:

```
function redeem(
        uint256 vaultId,
        uint256 shares,
        address receiver,
        address owner
    ) external onlyWhitelisted returns (uint256 assets){
    //...
    }
```

Recommendations:

Remove the `onlyWhitelisted` modifier from the `redeem()` function to ensure all users can redeem their deposited assets, regardless of their whitelist status:

```
function redeem(
        uint256 vaultId,
        uint256 shares,
        address receiver,
        address owner
-   ) external onlyWhitelisted returns (uint256 assets){
+   ) external  returns (uint256 assets){
    //...
    }
```

[L-13] Ineffective whitelisting in `OmoRouter` contract functions

- The `OmoRouter.deposit()` function restricts depositing in the vault to whitelisted users only, where these users are whitelisted by the router owner, however, the `receiver` address is not subjected to the same whitelist check, **allowing whitelisted users to deposit on behalf of non-whitelisted addresses**, which bypass the whitelisting mechanism.
- The same issue exists in the `OmoRouter.mint()` function, where a non-whitelisted `receiver` can indirectly benefit from the vault functionality through a whitelisted user acting on their behalf.

```
function deposit(
        uint256 vaultId,
        uint256 assets,
        address receiver
    ) external onlyWhitelisted returns (uint256 shares) {
        //...
        // Deposit into vault
        shares = IOmoVault(vault).deposit(assets, receiver);
        //...
    }
```

Recommendations:

Add a whitelist check for the `receiver` address in both the `deposit()` and `mint()` functions to ensure only whitelisted users can be specified as the `receiver`.

[L-14] `OmoVault.borrow()` : missing limits and tracking mechanism

The `OmoVault.borrow()` function is intended to allow whitelisted agents to borrow the vault's underlying asset, however, the current implementation has several issues:

1. The function allows whitelisted agents to withdraw any amount of the vault's assets without a defined limit, this enables an agent to withdraw the entire balance of the vault's underlying assets.

2. The contract doesn't track how much each agent has borrowed, which makes it impossible to manage or enforce repayment.

```
function borrow(
        uint256 assets,
        address receiver
    ) external nonReentrant onlyAgent {
        address msgSender = msg.sender;

        require(assets != 0, "ZERO_ASSETS");

        asset.safeTransfer(receiver, assets);

        emit Borrow(msgSender, receiver, assets);
    }
```

Recommendation:

- Introduce a maximum borrow limit for agents.
- Track the amount borrowed by each agent.

[L-15] DynamicAccount incorrect argument in `validateSignature`

The first argument in the `DynamicAccount._validateSignature` function is currently of type `UserOperation`. However, according to the expected signature in `BaseAccount`, (link) the first argument should be `PackedUserOperation`. This misalignment in function arguments could revert the transaction.

```
function _validateSignature(
        UserOperation calldata userOp,
        bytes32 userOpHash
    ) internal virtual override returns (uint256 validationData) {
    --snip--
    }
```

Recommendations:

```
function _validateSignature(
        PackedUserOperation calldata userOp,
        bytes32 userOpHash
    )
```