Aegis Vault Security Review

Pashov Audit Group

Conducted by: Said, btk, Shaka

August 12th 2024 - August 18th 2024

Contents

1. About Pashov Audit Group

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

2. Disclaimer

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

3. Introduction

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

4. About Aegis Vault

Aegis Vault automates the conversion of one type of token into another using advanced strategies. It integrates with Uniswap V3 pools to optimize liquidity and allows users to withdraw their funds in different ways depending on their preferences.

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 - 7f8badf33b62725acdd837ddac1d4819736b2ed1

fixes review commit hash - 1171f4c4cf9578d4addc00905a5085bd7299a4ff

Scope

The following smart contracts were in scope of the audit:

- `AegisVault`
- `AegisVaultERC20`
- `AegisVaultFactory`
- `AegisVaultCore`
- `Constants`
- `ERC20Initializable`
- `SymbolLib`
- `UV3Math`
- `PctMath`

7. Executive Summary

Over the course of the security review, Said, btk, Shaka engaged with Solo labs to review Aegis Vault. In this period of time a total of **11** issues were uncovered.

Protocol Summary

Protocol Name	Aegis Vault
Repository	https://github.com/labs-solo/aegis-vault
Date	August 12th 2024 - August 18th 2024
Protocol Type	Liquidity management

Findings Count

Severity	**Amount**
Medium	6
Low	5
Total Findings	**11**

Summary of Findings

ID	Title	Severity	Status
[M-01]	DoS for actions performing volatility check	Medium	Acknowledged
[M-02]	Claims from the Merkl distributor can fail	Medium	Resolved
[M-03]	Revert if re-depositing into ICHI vaults with an amount exceeding deposit0Max/deposit1Max	Medium	Acknowledged
[M-04]	Revert if attempting to re-deposit a zero amount	Medium	Resolved
[M-05]	performUpkeep could revert	Medium	Resolved
[M-06]	Bypassing MIN_INITIAL_DEPOSIT	Medium	Resolved
[L-01]	Owner can sweep tokens in certain cases	Low	Acknowledged
[L-02]	Using tokens with hooks	Low	Acknowledged
[L-03]	AegisVaultFactory does not check if ICHI vaults are single-sided	Low	Resolved
[L-04]	Users can avoid paying target token fees	Low	Acknowledged
[L-05]	Lack of minted shares slippage inside performUpkeep	Low	Resolved

8. Findings

8.1. Medium Findings

[M-01] DoS for actions performing volatility check

Severity

Impact: Medium

Likelihood: Medium

Description

The `_checkVolatility` function in `AegisVaultCore.sol` requires that the pending holdings ratio of deposit and target vaults are not above a certain threshold (set initially to 1%).

```
require(
        _pendingHoldingsRatio
          (depositVault, depositSpotPrice) <= pendingThreshold
            && _pendingHoldingsRatio
              (targetVault, targetSpotPrice) <= pendingThreshold,
        "PDT"
    );
```

This internal function is executed on deposit, rebalance, and withdrawal of type `TargetOnly`.

A malicious actor could perform a DoS over these actions by depositing tokens in one of the ICHI vaults before the transaction, causing it to revert, and withdrawing them after the transaction is reverted.

Note also that withdrawals in the ICHI vaults withdraw tokens proportionally from the Uniswap pools and from the unused funds in the vaults. This means that even after the withdrawal of the malicious actor, the pending holdings ratio could still be above the threshold, causing further transactions to revert.

Proof of concept

Copy the following code into `deposit.t.sol`:

```
function test_DoS_Deposit() public {
    // 1. Initial setup
    bytes32 depositorRoleId = aegisVaultCore.DEPOSITOR_ROLE();
    aegisVaultCore.grantRole(depositorRoleId, users.alice);

    vm.startPrank({ msgSender: users.alice });

    uint32 one_period = defaults.AEGIS_TWAP_PERIOD();
    skip(one_period);
    int24 finalTick = _swapToCrossTicks(true, 30);

    uint256 deposit0 = isToken0Deposit ? 1e16 : 0;
    uint256 deposit1 = isToken0Deposit ? 0 : 1e16;
    address to = users.alice;

    depositVault.deposit(deposit0, deposit1, to);
    targetVault.deposit(deposit1, deposit0, to);

    int24 baseLower = getClosestLowTick(-TICKS_100_PCT, tickSpacing);
    int24 baseUpper = getClosestLowTick(finalTick, tickSpacing);
    int24 limitUpper = getClosestHighTick
      (baseUpper + TICKS_100_PCT, tickSpacing);

    vm.startPrank({ msgSender: users.owner });
    depositVault.rebalance
      (baseLower, baseUpper, baseUpper + tickSpacing, limitUpper, 0);
    targetVault.rebalance
      (baseLower, baseUpper, baseUpper + tickSpacing, limitUpper, 0);

    vm.startPrank({ msgSender: users.alice });
    IERC20(address(depositVault)).approve({ spender: address
      (aegisVaultCore), amount: MAX_UINT256 });
    uint256 notTooLittleDepositToken = 510;

    // 2. Bob front-runs Alice's deposit, increasing the pending holdings ratio
    // in the deposit vault
    vm.startPrank({ msgSender: users.bob });
    uint256 ichiShares = depositVault.deposit(deposit0, deposit1, users.bob);

    // 3. Alice's deposit reverts
    vm.startPrank({ msgSender: users.alice });
    expectRevert("PDT");
    aegisVaultCore.deposit(notTooLittleDepositToken, 505, users.alice, true);

    // 4. Bob withdraws his deposit from the deposit vault
    vm.startPrank({ msgSender: users.bob });
    depositVault.withdraw(ichiShares, users.bob);

    // 5. Pending holdings ratio is still above the threshold, so deposits
    // continue to revert
    vm.startPrank({ msgSender: users.alice });
    expectRevert("PDT");
    aegisVaultCore.deposit(notTooLittleDepositToken, 505, users.alice, true);
}
```

Recommendations

Options:

1. It is recommended to evaluate if the impact of removing this check would be lower than the impact of the DoS attack and, if so, to remove it or, at least, allow the owner of the Aegis vault to disable it.

2. Consider increasing the `pendingThreshold` for low-liquidity vaults, making such attacks more costly. For instance, if the vaultValue is less than 1000e18, setting the `pendingThreshold` to 10% could be more appropriate.

3. Consider skipping the `_pendingHoldingsRatio` check for deposits, or using a separate, less restrictive `pendingThreshold` for `deposit` checks compared to the `pendingThreshold` used for rebalance checks.

Solo labs comments

Note that this type of DoS attack imposes a cost on the attacker. Deposits into ICHI Vaults are assessed using a pessimistic price (the less favorable of the spot price or TWAP), which means the attacker would incur some slippage if there has been any volatility in the pool. In addition to this inherent protection, the Aegis owner can also execute a manual rebalance by temporarily halting deposits and raising the pendingThreshold before rebalancing.

[M-02] Claims from the Merkl distributor can fail

Severity

Impact: Medium

Likelihood: Medium

Description

`AegisVaultCore.claim` allows the admin to claim rewards from the Merkl distributor contract and transfers the claimed tokens to the `recipient` address.

```
    IMerklDistributor(merkleDistributor).claim
(users, tokens, amounts, proofs);

    // Iterate over the list of token addresses and transfer the
// corresponding amounts to the recipient
    for (uint256 i = 0; i < tokens.length; i++) {
        IERC20(tokens[i]).safeTransfer(recipient, amounts[i]);
    }
```

As we can see, the amounts claimed for the tokens and the amounts transferred are the same. However, if we take a look at the code in the `Distributor.claim` function we can see that this is not always the case.

```
        uint256 toSend = amount - claimed[user][token].amount;
        claimed[user][token] = Claim(SafeCast.toUint208(amount), uint48
(block.timestamp), getMerkleRoot());

        IERC20(token).safeTransfer(user, toSend);
```

As leaves in the Merkle tree are composed by the keccak hash of the user address, the token address and the entitled amount, in order to offer multiple claims to the same user over the same token, a new leaf is created increasing the amount of the previous claim. So, in case that the Aegis vault claims a reward for a specific token, subsequent claims will cause the transaction to revert.

Additionally, for fee-on transfer tokens the amount received might be lower than the amount transferred by the Distributor contract. So for these kind of tokens the transaction will also revert in the first claim.

Proof of Concept

- Aegis vault is entitled to claim 100 XYZ tokens from the Merkle distributor.
- The admin calls the `claim` function and claims the 100 XYZ tokens.
- 50 more XYZ tokens are distributed to the Aegis vault, so the new leaf in the Merkle tree will be `keccak256(abi.encode(aegisVaultAddress, xyzAddress, 150e18))`.
- The admin calls the `claim` function again with the amount `150e18` and 50 XYZ tokens are transferred to the Aegis vault.
- `IERC20(xyzAddress).safeTransfer(recipient, 150e18)` reverts, as the Aegis vault only has 50 XYZ tokens.

Recommendations

```
+
+         // We could just transfer all available balance of tokens after claim, but ju
+
+         // targetVault, depositToken or targetToken, we ensure that only the increase
+        uint256[] balancesBefore = new uint256[](tokens.length);
+        for (uint256 i = 0; i < tokens.length; i++) {
+            balancesBefore[i] = IERC20(tokens[i]).balanceOf(address(this));
+        }

        IMerklDistributor(merkleDistributor).claim
          (users, tokens, amounts, proofs);

        // Iterate over the list of token addresses and transfer the
        // corresponding amounts to the recipient
        for (uint256 i = 0; i < tokens.length; i++) {
-            IERC20(tokens[i]).safeTransfer(recipient, amounts[i]);
+            uint256 balanceIncrease = IERC20(tokens[i]).balanceOf(address
+ (this)).sub(balancesBefore[i]);
+            IERC20(tokens[i]).safeTransfer(recipient, balanceIncrease);
        }
```

[M-03] Revert if re-depositing into ICHI vaults with an amount exceeding `deposit0Max`/`deposit1Max`

Severity

Impact: Medium

Likelihood: Medium

Description

The `withdraw` function allows users to choose `TargetOnly` as the withdrawal type. When this option is selected, users will receive only the target token, and the withdrawn deposit tokens from the `_withdrawAssetMix` operation will be redeposited into the deposit vault.

```
function _withdraw(
        uint256 aegisShares,
        uint256 aegisTotalSupply,
        address to,
        WithdrawSlippageData memory minSlippage,
        WithdrawType withdrawType
    )
        internal
        returns (
          WithdrawSlippageDatamemoryactualSlippage,
          uint256aegisSharesWithdrawn
        )
    {
        // ...

        } else if (withdrawType == WithdrawType.TargetOnly) {
            // NOTE: despite the possibility of a re-deposit on
            // WithdrawType.TargetOnly
            // we don't check _onlyDepositor as it's not a standard deposit
            // NOTE: we _checkVolatility before _withdrawAssetMix to avoid
            // oracle updates due to burns&mints in underlying pools
            // which could result in the ICHIVault re-deposit
            //(i.e. in _depositDepositToken) failing on the hysteresis check
            (
              uint256depositSpotPrice,
              uint256targetSpotPrice
            ) = _checkVolatility(

            // the AegisVault initially custodies both tokens, then transfers
            // the targetTokens to the recipient
            // and then re-deposits the depositTokens.
            // NOTE: the withdrawFee is charged identically across all
            // WithdrawTypes
            // so in the worst case of a TargetOnly withdrawal
            //(i.e. the user's position has no targetToken and only has depositToken)
            // the user would essentially be paying a withdrawFee for nothing in
            // return
>>>
  (actualSlippage.depositTokenAmount, actualSlippage.targetTokenAmount) =
                _withdrawAssetMix(aegisShares, address(this), aegisTotalSupply);

            targetToken.safeTransfer(to, actualSlippage.targetTokenAmount);

            // after _withdrawAssetMix we've temporarily burned all withdraw
            // aegisShares
            // however we then calculate the deposit aegisShares that would
            // normally be minted on the deposit below
            // NOTE:
            //(deposit aegisShares <=  withdraw aegisShares) is necessarily logically
            // so we effectively only need to burn the difference as calculated
            // in aegisSharesWithdrawn
            uint256 newTotalSupply = aegisTotalSupply.sub(aegisShares);

>>>         actualSlippage.depositSharesAmount =
                _depositDepositToken(
                  actualSlippage.depositTokenAmount,
                  true,
                  newTotalSupply,
                  depositSpotPrice,
                  targetSpotPrice
                );

            aegisSharesWithdrawn = aegisShares.sub
              (actualSlippage.depositSharesAmount);
        }
```

However, this does not account for the scenario where the returned `actualSlippage.depositTokenAmount` from both ICHI vaults exceeds `deposit0Max`/`deposit1Max`. When this occurs, the user’s withdraw operation may unexpectedly revert due to attempting to re-deposit an amount that exceeds the deposit ICHI vault's `deposit0Max`/`deposit1Max`.

```
function deposit(
        uint256 deposit0,
        uint256 deposit1,
        address to
    ) external override nonReentrant returns (uint256 shares) {
        require(allowToken0 || deposit0 == 0, "IV.deposit: token0 not allowed");
        require(allowToken1 || deposit1 == 0, "IV.deposit: token1 not allowed");
        require
          (deposit0 > 0 || deposit1 > 0, "IV.deposit: deposits must be > 0");
>>>     require(
  deposit0<deposit0Max&&deposit1<deposit1Max,
  "IV.deposit:depositstoolarge"
);
        require(to != NULL_ADDRESS && to != address(this), "IV.deposit: to");

        // ...
    }
```

Recommendations

When the `actualSlippage.depositTokenAmount` to be re-deposited exceeds `deposit0Max`/`deposit1Max`, break down the single deposit operation into multiple transactions to avoid unexpected reverts.

Solo labs comments

The depositMax0 and depositMax1 settings are configured at the ICHI Vault level by the vault owner. Aegis adheres to these limits by reverting any transactions that attempt to deposit into ICHI Vaults with established depositMax0 or depositMax1 constraints. This applies to both direct token deposits and re-deposits during TargetOnly withdrawals. However, Aegis depositors can still withdraw using other available withdrawal options, ensuring their funds are not locked.

[M-04] Revert if attempting to re-deposit a zero amount

Severity

Impact: Medium

Likelihood: Medium

Description

The `withdraw` function allows users to choose `TargetOnly` as the withdrawal type. When this option is selected, users will receive only the target token, and the withdrawn deposit tokens from the `_withdrawAssetMix` operation will be redeposited into the deposit vault.

```
function _withdraw(
        uint256 aegisShares,
        uint256 aegisTotalSupply,
        address to,
        WithdrawSlippageData memory minSlippage,
        WithdrawType withdrawType
    )
        internal
        returns (
          WithdrawSlippageDatamemoryactualSlippage,
          uint256aegisSharesWithdrawn
        )
    {
        // ...

        } else if (withdrawType == WithdrawType.TargetOnly) {
            // NOTE: despite the possibility of a re-deposit on
            // WithdrawType.TargetOnly
            // we don't check _onlyDepositor as it's not a standard deposit
            // NOTE: we _checkVolatility before _withdrawAssetMix to avoid
            // oracle updates due to burns&mints in underlying pools
            // which could result in the ICHIVault re-deposit
            //(i.e. in _depositDepositToken) failing on the hysteresis check
            (
              uint256depositSpotPrice,
              uint256targetSpotPrice
            ) = _checkVolatility(

            // the AegisVault initially custodies both tokens, then transfers
            // the targetTokens to the recipient
            // and then re-deposits the depositTokens.
            // NOTE: the withdrawFee is charged identically across all
            // WithdrawTypes
            // so in the worst case of a TargetOnly withdrawal
            //(i.e. the user's position has no targetToken and only has depositToken)
            // the user would essentially be paying a withdrawFee for nothing in
            // return
>>>
  (actualSlippage.depositTokenAmount, actualSlippage.targetTokenAmount) =
                _withdrawAssetMix(aegisShares, address(this), aegisTotalSupply);

            targetToken.safeTransfer(to, actualSlippage.targetTokenAmount);

            // after _withdrawAssetMix we've temporarily burned all withdraw
            // aegisShares
            // however we then calculate the deposit aegisShares that would
            // normally be minted on the deposit below
            // NOTE:
            //(deposit aegisShares <=  withdraw aegisShares) is necessarily logically
            // so we effectively only need to burn the difference as calculated
            // in aegisSharesWithdrawn
            uint256 newTotalSupply = aegisTotalSupply.sub(aegisShares);

>>>         actualSlippage.depositSharesAmount =
                _depositDepositToken(
                  actualSlippage.depositTokenAmount,
                  true,
                  newTotalSupply,
                  depositSpotPrice,
                  targetSpotPrice
                );

            aegisSharesWithdrawn = aegisShares.sub
              (actualSlippage.depositSharesAmount);
        }
```

However, this does not consider the scenario where the ICHI vaults return only `actualSlippage.targetTokenAmount` and no `actualSlippage.depositTokenAmount` (equal to 0). This can happen when both ICHI vaults provide liquidity to UniV3 ticks that are currently out of the active range. When this occurs, the operation will revert because a zero amount is provided for both tokens in the ICHI vault deposit operation.

```
function deposit(
        uint256 deposit0,
        uint256 deposit1,
        address to
    ) external override nonReentrant returns (uint256 shares) {
        require(allowToken0 || deposit0 == 0, "IV.deposit: token0 not allowed");
        require(allowToken1 || deposit1 == 0, "IV.deposit: token1 not allowed");
>>>     require
  (deposit0 > 0 || deposit1 > 0, "IV.deposit: deposits must be > 0");
        require(
          deposit0<deposit0Max&&deposit1<deposit1Max,
          "IV.deposit:depositstoolarge"
        );
        require(to != NULL_ADDRESS && to != address(this), "IV.deposit: to");

        // ...
    }
```

Recommendations

Consider to add condition, do deposit operation only if `actualSlippage.depositTokenAmount` greater than 0.

```
function _withdraw(
        uint256 aegisShares,
        uint256 aegisTotalSupply,
        address to,
        WithdrawSlippageData memory minSlippage,
        WithdrawType withdrawType
    )
        internal
        returns (
          WithdrawSlippageDatamemoryactualSlippage,
          uint256aegisSharesWithdrawn
        )
    {
        // ...

        } else if (withdrawType == WithdrawType.TargetOnly) {
            // ...
            (
              uint256depositSpotPrice,
              uint256targetSpotPrice
            ) = _checkVolatility(

            // ...
            (
              actualSlippage.depositTokenAmount,
              actualSlippage.targetTokenAmount
            ) =
                _withdrawAssetMix(aegisShares, address(this), aegisTotalSupply);

            targetToken.safeTransfer(to, actualSlippage.targetTokenAmount);

            // ...
            uint256 newTotalSupply = aegisTotalSupply.sub(aegisShares);

+           if (actualSlippage.depositTokenAmount > 0) {
                  actualSlippage.depositSharesAmount =
                      _depositDepositToken(
                        actualSlippage.depositTokenAmount,
                        true,
                        newTotalSupply,
                        depositSpotPrice,
                        targetSpotPrice
                      );
+           }
            aegisSharesWithdrawn = aegisShares.sub
              (actualSlippage.depositSharesAmount);
        }
```

[M-05] `performUpkeep` could revert

Severity

Impact: High

Likelihood: Low

Description

When `performUpkeep` is called, it will withdraw a percentage of shares from the deposit vault and then deposit the withdrawn tokens into the deposit and target vaults accordingly.

```
function _rebalance(uint256 rebalancePct) private {
        require
          (rebalancePct > 0 && rebalancePct <= maxRebalanceThreshold, "IIN");

        // don't do rebalance during high volatility
        (uint256 depositSpotPrice,) = _checkVolatility();

        // don't rebalance if there are too few target tokens in the
        // depositVault
        require(_targetTokenRatio
          (depositSpotPrice) >= excessTargetTokenThreshold, "ITT");

        // calculate how many shares to withdraw from the depositVault
        uint256 toWithdraw = PctMath.pctOf(depositVault.balanceOf(address
          (this)), rebalancePct);

        // withdraw shares from the depositVault
>>>     depositVault.withdraw(toWithdraw, address(this));

        // take a portion of targetToken as fees before depositing them to the
        // targetVault
        uint256 fees = _distributeTargetTokenFees();

        // get pending token amounts
        uint256 depositTokenBalance = depositToken.balanceOf(address(this));
        uint256 targetTokenBalance = targetToken.balanceOf(address(this));

        // @audit - this could also revert when one of them is empty
        // deposit pending depositToken to the depositVault
>>>     depositVault.deposit(
  isInverted?0:depositTokenBalance,
  isInverted?depositTokenBalance:0,
  address
)

        // deposit pending targetToken to the targetVault
>>>     targetVault.deposit(
  isInverted?targetTokenBalance:0,
  isInverted?0:targetTokenBalance,
  address
)

        emit Rebalance(
          rebalancePct,
          toWithdraw,
          depositTokenBalance,
          targetTokenBalance,
          fees
        );
    }
```

However, this does not account for the scenario where all withdrawn assets from the ICHI vaults consist solely of `targetToken`. This can happen when both ICHI vaults provide liquidity to UniV3 ticks that are currently out of the active range. When this occurs, the operation will revert because a zero amount is provided for both tokens in the ICHI vault's `depositVault.deposit` operation.

Recommendations

When `depositTokenBalance` is 0, skip the deposit operation. `targetTokenBalance` check is not needed, as the `_targetTokenRatio` check should ensure a non-zero `targetTokenBalance`.

[M-06] Bypassing `MIN_INITIAL_DEPOSIT`

Severity

Impact: High

Likelihood: Low

Description

When users first deposit into the Aegis vault (`aegisTotalSupply` is 0), the operation will check if `aegisShares` minted is greater than `MIN_INITIAL_DEPOSIT`. This is designed to prevent attacks from first depositors.

```
function __deposit(
        uint256 depositSharesAmount,
        uint256 aegisTotalSupply,
        uint256 depositSpotPrice,
        uint256 targetSpotPrice
    )
        private
        returns (uint256 aegisShares)
    {
        // ...
        if (aegisTotalSupply == 0) {
            // better to use userDepositValue than userValueContribution here,
            // because userValueContribution could be very small
            aegisShares = ctx.userDepositValue;
            // simplified check for initial deposit
            // @audit - can this exploited by withdrawing immediately until
            // aegisShares lower than minimum
>>>         require(aegisShares >= MIN_INITIAL_DEPOSIT, "VTS");
        } else {
            /// @dev invariant check for guaranteed safety, since at this point
            // the AegisVault has received any ICHIVault shares
            /// either from a depositToken deposit or for a depositVault share
            // transfer therefore given existing prior deposits
            /// the user's contribution to the whole must necessarily be less
            // than the whole i.e. PRECISION i.e. 1e18
            require(ctx.userValueContribution < PRECISION, "VTS");
            aegisShares = _mulDiv(
              ctx.userValueContribution,
              aegisTotalSupply,
              PRECISION.sub
            )
        }
    }
```

However, this can be easily bypassed by immediately withdrawing shares until it becomes feasible to perform a first depositor attack since there is no check in the withdraw operation to ensure that the remaining shares or total supply are greater than `MIN_INITIAL_DEPOSIT`.

Recommendations

Check the new total supply after the `withdraw` operation. If it is not zero and is lower than `MIN_INITIAL_DEPOSIT`, revert the operation.

```
function _withdraw(
        uint256 aegisShares,
        uint256 aegisTotalSupply,
        address to,
        WithdrawSlippageData memory minSlippage,
        WithdrawType withdrawType
    )
        internal
        returns (
          WithdrawSlippageDatamemoryactualSlippage,
          uint256aegisSharesWithdrawn
        )
    {
        // ...

+      uint256 newAegisTotalSupply = aegisTotalSupply - aegisSharesWithdrawn;
+      require
+ (newAegisTotalSupply == 0  || newAegisTotalSupply >= MIN_INITIAL_DEPOSIT, "VTS");
        _checkWithdrawSlippage(actualSlippage, minSlippage);
        emit Withdraw(msg.sender, to, aegisShares, actualSlippage);
    }
```

8.2. Low Findings

[L-01] Owner can sweep tokens in certain cases

The AegisVaultCore includes a `sweepExtraTokens()` function designed to recover non-vault tokens from the contract:

```
function sweepExtraTokens
      (address _token, address _recipient) external override {
        _onlyAdmin();
        require(
            _token != address(depositVault) && _token != address
              (targetVault) && _token != address(depositToken)
                && _token != address(targetToken),
            "WTK"
        );

        IERC20 token = IERC20(_token);
        uint256 tokenBalance = token.balanceOf(address(this));
        require(tokenBalance > 0, "ZBL");
        token.safeTransfer(_recipient, tokenBalance);

        emit SweepExtraTokens(msg.sender, _token, _recipient, tokenBalance);
    }
```

When the owner attempts to rescue a vault token, the function throws a "WTK" error. However, the owner can bypass this check if a token has multiple entry points. For example, this was the case with the TUSD token when its secondary entry point was still active. This vulnerability remains a risk as other tokens with multiple entry points may exist. While there is no direct fix for this issue, it is important to document that the owner can withdraw all tokens in cases where tokens have multiple entry points.

Solo labs comments

While the Aegis owner cannot recover deposit and target tokens when they are implemented as standard ERC20 tokens, recovery could potentially be possible if these tokens have multiple entry points. However, this scenario is highly unlikely, as such tokens are rare and not typically used in ICHI vaults. If the need for an Aegis contract that handles such tokens ever arises, it will be evaluated on a case-by-case basis.

[L-02] Using tokens with hooks

On deposits to the Aegis vault, there is a check on volatility that also fetches the spot prices that are used for the calculation of the shares received by the depositor. This check happens before the deposit is made, which breaks the checks-effects-interactions pattern.

If the deposit token implements hooks on transfer, the depositor could manipulate the spot prices in the Uniswap pools, but the old prices would still be used for the calculation of the shares received.

The same principle applies to withdrawals of the type `TargetOnly`.

Consider moving the check on volatility after the tokens are transferred and/or document that tokens implementing hooks on transfer should not be used in the protocol.

Solo labs comments

Aegis will not be used with tokens that have transfer hooks that could potentially lead to price manipulation in the pool.

[L-03] AegisVaultFactory does not check if ICHI vaults are single-sided

The statement of expected behavior says the following

> vaults are single-sided, meaning the Deposit Vault only accepts deposit tokens and the Target Vault only accepts target tokens

However, `AegisVaultFactory.createAegisVault` does not perform any checks to ensure that the ICHI vaults are single-sided. This could lead to unexpected behavior if the same ICHI vault is used as the deposit vault for both token0 and token1, as the Aegis vault could be constantly rebalancing between the two tokens.

Consider applying the following change to `AegisVaultFactory.sol`:

```
require(
            createAegisVaultParams.depositToken == dToken0
-               ? IICHIVault(createAegisVaultParams.depositVault).allowToken0
- () && IICHIVault(createAegisVaultParams.targetVault).allowToken1()
+               ? IICHIVault(createAegisVaultParams.depositVault).allowToken0
+ () && IICHIVault(createAegisVaultParams.targetVault).allowToken1() &&
+                 !IICHIVault(createAegisVaultParams.depositVault).allowToken1
+ () && !IICHIVault(createAegisVaultParams.targetVault).allowToken0()
-               : IICHIVault(createAegisVaultParams.depositVault).allowToken1
- () && IICHIVault(createAegisVaultParams.targetVault).allowToken0(),
+               : IICHIVault(createAegisVaultParams.depositVault).allowToken1
+ () && IICHIVault(createAegisVaultParams.targetVault).allowToken0() &&
+                 !IICHIVault(createAegisVaultParams.depositVault).allowToken0
+ () && !IICHIVault(createAegisVaultParams.targetVault).allowToken1(),
            "WDT"
        )
```

[L-04] Users can avoid paying target token fees

When `AegisVaultCore.performUpKeep` is called by the rebalancer keeper there is a fee that is applied over the target tokens being rebalanced, and the fee is sent to the `feeRecipient` address.

Users can avoid paying their share of the fee by calling the withdraw function before the rebalance is triggered and re-depositing their tokens after the rebalance is completed.

It is important to note that the profitability of the strategy depends on several factors, such as the base fee (initially 1%), the withdrawal fee (initially 0), the amount of target tokens rebalanced, the percentage of shares owned by the user over the total supply, the prices of the tokens in the underlying pools, the gas price, etc.

The most straightforward way to mitigate this issue would be to ensure that the base fee is not much higher than the withdrawal fee.

Solo labs comments

Both the baseFee and withdrawFee are set to 0% by default. If it becomes necessary to increase the baseFee for a specific Aegis contract, the Aegis admin can adjust the withdrawFee accordingly, taking this consideration into account.

[L-05] Lack of minted shares slippage inside `performUpkeep`

Inside the `performUpkeep` operation, it will withdraw a percentage of shares from the deposit vault (once the target ratio is met) and then deposit the tokens into the deposit and target ICHI vaults accordingly.

```
function _rebalance(uint256 rebalancePct) private {
        require
          (rebalancePct > 0 && rebalancePct <= maxRebalanceThreshold, "IIN");

        // don't do rebalance during high volatility
        (uint256 depositSpotPrice,) = _checkVolatility();

        // don't rebalance if there are too few target tokens in the
        // depositVault
        require(_targetTokenRatio
          (depositSpotPrice) >= excessTargetTokenThreshold, "ITT");

        // calculate how many shares to withdraw from the depositVault
        uint256 toWithdraw = PctMath.pctOf(depositVault.balanceOf(address
          (this)), rebalancePct);

        // withdraw shares from the depositVault
        depositVault.withdraw(toWithdraw, address(this));

        // take a portion of targetToken as fees before depositing them to the
        // targetVault
        uint256 fees = _distributeTargetTokenFees();

        // get pending token amounts
        uint256 depositTokenBalance = depositToken.balanceOf(address(this));
        uint256 targetTokenBalance = targetToken.balanceOf(address(this));

        // deposit pending depositToken to the depositVault
        depositVault.deposit(
          isInverted?0:depositTokenBalance,
          isInverted?depositTokenBalance:0,
          address
        )

        // deposit pending targetToken to the targetVault
        targetVault.deposit(
          isInverted?targetTokenBalance:0,
          isInverted?0:targetTokenBalance,
          address
        )

        emit Rebalance(
          rebalancePct,
          toWithdraw,
          depositTokenBalance,
          targetTokenBalance,
          fees
        );
    }
```

While the price of the tokens cannot be manipulated due to the volatility check, the price of ICHI vault shares can still be manipulated (due to donation or sandwich attacks). Rebalance operations could cause the Aegis vault to mint fewer shares than expected.

Add minimum minted shares to protect `performUpkeep` operation.