Aera Security Review

Auditors

Gerard Persoon, Lead Security Researcher

D-Nice, Lead Security Researcher

Matt Solomon, Security Researcher

Pashov Krum, Associate Security Researcher

Report prepared by: Pablo Misirov

September 22, 2023

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

Aera is a treasury management protocol that attempts to address existing shortcomings with controlling treasury funds. Aera supports:

- Robust Asset Selection No need to plan strategies. Just pick assets.
- Efficient Purchasing Remove bureaucracy from DAOs.
- Decentralized Active Management Market aware & tailored to your protocol.

Disclaimer: This security review does not guarantee against a hack. It is a snapshot in time of aera-contracts-v2 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 10 days in total, Aera engaged with Spearbit to review the aera-contracts-v2 protocol. In this period of time a total of **87** issues were found.

Summary

Project Name	Aera
Repository	aera-contracts-v2
Commit	2adbc0...e02c
Type of Project	Financial Modeling, DeFi
Audit Timeline	Aug 14 - Aug 25
Two week fix period	Aug 25 - September 8

Issues Found

Severity	Count	Fixed	Acknowledged
Critical Risk	0	0	0
High Risk	1	1	0
Medium Risk	20	14	6
Low Risk	18	15	3
Gas Optimizations	17	14	3
Informational	31	26	5
Total	**87**	**70**	**17**

5 Findings

5.1 High Risk

5.1.1 Funds of negligent `owner` can be accessed by `guardian`

Severity: High Risk

Context: AeraVaultV2.sol#L169-L224

Description: Funds get added to the vault via `deposit()`. The `owner` has to set an allowance to be able to use `deposit()`. Assume the `owner()` has accidentally set an allowance that is larger than is used for `deposit()`, for example an unlimited allowance, for example for USDC. Assume there is also an allowlist entry so the guardian can use `safeTransferFrom` for USDC. Note: with ERC4626 there are two additional functions that can use the allowance: `withdraw()` and `redeem()`.

In that situation the guardian can use a `submit()` call to transfer additional USDC from the `owner`, either to the vault or to another address, for example his own address. So he can effectively steal from the `owner`. A variation on this is that the guardian could frontrun the `deposit()` by the `owner`, transfering the USDC out and causing `deposit()` to fail. This will likely be detected and contracts with the guardian will probably be used to resolve this. But still its probably better to prevent this.

```
function deposit(AssetValue[] calldata amounts) ... {
    ...
    // Interactions: transfer asset from owner to vault.
    assetValue.asset.safeTransferFrom(owner(), address(this), assetValue.value);
    ...
}
```

Recommendation: Consider disallowing the underlying selector of `safeTransferFrom`, which is `transferFrom()`. This could be done generally or more specific from the `owner`. Because the owner can change hooks, this check needs to live in the vault, not in a hook, to ensure it's permanent. The last check requires checking a third parameter from the call data. Similarly disallow `withdraw()` and `redeem()`. Note: that parameters of these functions are ordered differently.

Another approach would be that the `owner` uses `safeTransfer()` to send assets to the vault. However then there is no check that the tokens are on the allowed list, there is no `event` and there is no verification by hooks.

Aera: Fixed in PR 184.

Spearbit: Verified. If an owner change happens, the previous owner would be still vulnerable to this, so it'd be a good idea to require them to reset any allowances prior to owner changes.

Aera: We will include this caveat in our documentation.

Spearbit: Acknowledged.

5.2 Medium Risk

5.2.1 The `guardian` can use his own MEV opportunities

Severity: Medium Risk

Context: AeraVaultV2.sol#L411-L464

Description: When a guardian calls `submit()` he can create MEV opportunities. As the guardian knows about this, he can make use of this by frontrunning / backrunning / sandwiching the trades, for example by doing the opposite trade. If he use a smart contract to call `submit()`, it is straightforward to add transactions. He can also do this by bundling transactions and sending them to flashbots, which is far more difficult to detect. The slippage is limited by the `DailyExecutionLoss` checks in the hooks contract.

Recommendation: Monitor occurances of large slippage.

Aera: Aware, doesn't violate our current trust model and guardian/slippage will be monitored.

Spearbit: Acknowledged.

5.2.2 Multiple-token-addresses can circumvent checks

Severity: Medium Risk

Context: AeraVaultV2.sol#L169-L224, AeraVaultAssetRegistry.sol#L186-L226

Description: There are special tokens that can be address via separate addresses, see weird-erc20 multiple-token-addresses. If both these addresses are added via `addAsset()`, they would both be accepted. Their balance would be counted twice in `value()` and would thus artificially inflate the `value()` and would allow circumventing `DailyExecutionLoss` checks from the hooks contract.

```
function addAsset(AssetInformation calldata asset) ... {
    ...
    // Requirements: check that asset is not already present.
    if (asset.asset == _assets[i].asset) {
        revert Aera__AssetIsAlreadyRegistered(i);
    }
    ...
}
```

These special tokens will also circumvent the checks in `deposit()`, but because `safeTransferFrom()` will be done twice, there is no additional risk.

```
function deposit(AssetValue[] calldata amounts)
    ...
    for (uint256 i = 0; i < numAmounts;) {
        assetValue = amounts[i];
        ...
        for (uint256 j = 0; j < numAmounts;) {
            // Requirements: check that no duplicate assets are provided.
            if (i != j && assetValue.asset == amounts[j].asset) {
                revert Aera__AssetIsDuplicated(assetValue.asset);
            }
            ...
            assetValue.asset.safeTransferFrom( owner(), address(this), assetValue.value );
            ...
        }
    ...
 }
```

Recommendation: Carefully select the tokens that are registered.

Aera: We have a diligent whitelisting process for assets.

Spearbit: Acknowledged.

5.2.3 Beware of CREATE2 dangers with replaceable contracts possible via arbitrary factory

Severity: Medium Risk

Context: AeraVaultV2Factory.sol#L63-L71, AeraVaultV2Factory.sol#L118

Description: The `AeraVaultV2` contract is designed to be deployed via `AeraVaultV2Factory` using `CREATE2` for purposes of deterministic deployment. This is a valid use case, however, there are additional risks potentially exposed by `CREATE2` such as the possibility of replaceable or metamorphic contracts by those able to deploy with the factory, specifically when arbitrary code deployment is allowed. This condition is currently met.

Additionally, the deployed contract requires a `SELFDESTRUCT`, which may be obfuscated via `DELEGATECALL` or `CALLCODE`. This latter condition is not met in the current codebase. However, with the arbitrary code deployment, it may be trivial for the owner to deploy transient contracts that contain the necessary `SELFDESTRUCT`, as the underlying contents would be hidden within the creation code that most users may not be tracking. Most users would likely consider deployments from the factory to be their litmus test for safety.

When all conditions are met, transient vault contracts could be deployed and replaced, even with completely different bytecode while retaining their address. This could open up novel attacks, where vaults are deployed by the factory, and users load those vaults up with assets, which contain all the expected security guarantees. Then, if via some mechanic a `SELFDESTRUCT` occurs, the factory owner or anyone that can deploy via it could replace that bytecode with any they wish, and simply claim any assets that were in the vault.

Recommendation: It would be ideal to eliminate the possibility of this completely, by separating out the arbitrary deployment logic from the `AeraVaultV2Factory`. There should be a factory that only deploys `AeraVaultV2`, `AeraVaultAssetRegistry` and `AeraVaultHooks` contract, with the necessary `creationCode` derived there.

If the arbitrary deployer is required, it should be its own separate deployed contract, albeit anything from that arbitrary factory could be transient without thorough on-chain analysis, which would be necessary for any contracts deployed via it.

Aera: We are removing arbitrary deployment logic from the contracts. *"Atomic deployments can simplify instance setup and reduce risk"*.

Spearbit: Fixed.

5.2.4 Guardian can rapidly drain `2 * maxDailyExecutionLoss (%) * vaultValue`

Severity: Medium Risk

Context: AeraVaultHooks.sol#L215-L243

Description: Days are discrete intervals computed by `floor(block.timestamp / 86400)`. This means a guardian can quickly drain `2 * maxDailyExecutionLoss (%) * currentVaultValue` by waiting until a block where `block.timestamp % 86400 >= (86400 - x)`, where `x` seconds is the chain's block time. This lets them drain `maxDailyExecutionLoss` at that block, and drain it again at the next block when the new day starts.

The ability to do this in consecutive blocks gives the owner very little time to react and determine if this was malicious, particularly on L2s and other chains where block times can be2 seconds or less. This is exacerbated by multi-block MEV, if the guardian can propose consecutive blocks to guarantee the owner cannot react in time.

Assuming the guardian either directly or indirectly earns accrued fees, then at any given day transition a guardian is economically incentivized to carry out this attack if `2 * maxDailyExecutionLoss (%) * currentVaultValue` is greater than the lifetime amount the guardian can expect to earn by being honest. Similarly, intra-day, a guardian is incentivized to directly transfer out funds to themself if `maxDailyExecutionLoss (%) * currentVaultValue` is greater than the expected amount of lifetime fees.

Recommendation: One mitigation is to accept this risk and set the `maxDailyExecutionLoss` and `fee` values such that the guardian is not incentivized to carry out the attack. The lifetime amount of accrued fees can be estimated by `vaultLifespan * averageVaultValue * fee (%)`, where `vaultLifespan` is the average amount of time in seconds the guardian will manage the vault, `averageVaultValue` is an estimation of the average value during that

time, and `fee` is the fee in vault storage. Therefore, a guardian is not economically incentivized to steal funds in this way when: `2 * maxDailyExecutionLoss (%) * maxExpectedValue < vaultLifespan * averageVaultValue * fee (%)`, where `maxExpectedValue` is the estimated maximum vault value that may occur at any point the vault's lifespan.

A related approach is to accept the risk and set `maxDailyExecutionLoss` to half of the true desired value. Both this and the prior mitigation should include monitoring and documentation to make this attack easier to detect.

Another mitigation is to change how the `maxDailyExecutionLoss` threshold is tracked to something more robust. One approach is to track the max value seen in the last 24 hours (`maxValue`), the timestamp at which the value was seen (`maxValueTimestamp`), and comparing against that as follows:

- In the `beforeSubmit` hook, if `block.timestamp - 24 hours > maxValueTimestamp`, then the 24 hours window elapsed so we save `maxValue = custody.value()`.
- Otherwise, `if (custody.value() > maxValue)`, then save `maxValue = custody.value()`.
- Then the `afterSubmit` hook compares the current `custody.value()` to `maxValue` without having to consider days anymore.
- This approach also removes the need for the `cumulativeDailyMultiplier` and the `if (currentDay == day)` branch.

This `maxValue` approach helps because there's now no longer a well-defined point at which this attack can be executed in consecutive blocks. A downside is that it's sensitive to intra-day price swings: If the guardian performs an action when prices are high, a large `maxValue` is saved off. If prices crash, actions can no longer be taken during that day since they all will result in the `maxDailyExecutionLoss` threshold being exceeded, simply due to price changes.

An alternative approach is to use rolling windows based off a number of observations or a given duration, but this may add a lot of complexity.

Lastly, instead of requiring 24 hours as the duration, it can be a constructor parameter in the hook contract. The longer the duration used to compute losses, the less likely it is that a transition between them will satisfy `2 * maxDailyExecutionLoss (%) * currentVaultValue < vaultLifespan * averageVaultValue * fee (%)`. As durations increase, the `2` constant factor can effectively be dropped.

Aera: Given our trust model, the resolution we will take is to set a value that is appropriately small.

Spearbit: Acknowledged.

5.2.5 Numeraire tokens with low decimals can cause inaccurate pricing of other tokens

Severity: Medium Risk

Context: AeraVaultAssetRegistry.sol#L366-L370, AeraVaultAssetRegistry.sol#L382-L386

Description: All prices are scaled to have the same number of decimals as the numeraire on L382–386. When the numeraire has less decimals, prices are scaled down with `price = price / (10 ** (oracleDecimals - numeraireDecimals))`. This causes precision loss in pricing tokens, which can result in accurate vault values.

The precision lost is most severe when dealing with low-decimal tokens. Below are two examples quantifying this precision loss for a vault with GUSD (2 decimals) and WETH (18 decimals) and another vault with GUSD and USDC (6 decimals), where GUSD is the numeraire in each case. The below data returned from Chainlink oracles on 21-Aug-2023 is used for these examples.

oracle	decimals	price	address
GUSD/ETH	18	599219798568989	0x96d15851CBac05aEe4EFD9eA3a3DD9BDEeC9fC28
GUSD/USD	8	100095281	0xa89f5d2365ce98B3cD68012b6f503ab1416245Fc
USDC/USD	8	99999700	0x8fFfFfd4AfB6115b954Bd326cbe7B4BA576818f6

With GUSD and WETH:

- Take the inverse of the GUSD/ETH price to get the WETH price denominated in GUSD.
- The WETH/GUSD price therefore is `1e18 ** 2 / 599219798568989 = 1668836714654828989087`. We can verify this by dividing the result by 1e18 since it's an 18 decimal oracle, returning the expected USD price of $1,668.8367146548.
- Since `numeraireDecimals < oracleDecimals` is true (`2 < 18`), the price is scaled down by `10 ** (18-2)`. This gives `1668836714654828989087 / 10**16 = 166883`, which is a price of $1,668.83
- Therefore the current WETH price is off from the "true" WETH price by $0.0067146548 (0.0004023554%), which is likely negligible in most scenarios.

With GUSD and USDC:

- Chainlink does not have a direct GUSD/USDC oracle so computing that price must pass through USD, meaning there would be a chainlink oracle wrapper that computes the USDC/GUSD price as:
  - `(USDC / USD) * (USD / GUSD)`
  - `(USDC / USD) * (1e8 ** 2 / (GUSD / USD))`
  - `99999700 * 99904809`
  - `9990450928557300` which has 16 decimals. Sanity check: `9990450928557300 / 1e16 = $0.9990450929` which checks out since both are stable coins pegged to $1.
- This price is then scaled down to 2 decimals, giving `9990450928557300 / 10 ** (16-2) = 99`, which is $0.99.
- This is an error of $0.0090450929, or 0.905%, which is much more significant than the prior case.

This GUSD + USDC vault may be the worst realistic case, as they are two mainstream tokens with low decimals, and that introduces ~1% error into the pricing. Because the error comes from truncating digits, the result is an underestimation of value by 1%. In the worst case, where a vault with GUSD numeraire is primarily composed of USDC, the vault value, and consequently accrued fees, can be 1% less than expected.

Recommendation: Instead of scaling all prices to the same number of decimals as the numeraire, scale all prices, including the numeraire's, up to 18 decimals.

Aera: Fixed in PR 160 and PR 181.

Spearbit: Verified.

5.2.6 Guardians can artificially increase accrued fees for free

Severity: Medium Risk

Context: AeraVaultV2.sol#L411-L464

Description: All accounting in the `submit()` method is done using `token.balanceOf()` calls, which allows the guardian can artificially increase total accrued fees for free as follows:

1. Guardian transfers some token (such as USDC) directly to vault. This token must be in the asset registry and the guardian must have the ability to transfer the token out of the vault.
2. Guardian calls `submit` with a single operation that transfers out the USDC sent in the previous step.
3. Within `submit`, fees are accrued in `_reserveFees()`. This uses the inflated USDC balance from step 1 to compute the now-inflated vault value. This increases the values of `fees[feeRecipient]` and `feesTotal` accordingly.
4. Next `hooks.beforeSubmit()` saves off this inflated value.
5. The operation to transfer out USDC is executed.

6. As long as the amount of USDC sent in (and consequently removed) does not increase the vault value by more than `maxDailyExecutionLoss`, the vault balance decrease is allowed. This allows the guardian to increase accrued fees by `maxDailyExecutionLoss` per day.

Below is a test which demonstrates this that can be added to `Submit.t.sol`. Run `forge test --match-test test_submit_success_ -vvv` to run both, which contains `console.log` statements showing that the fees accrued are 10% higher in the attack test—`maxDailyExecutionLoss` is 10% in the test—for the above reason. This can be repeated every day by modifying the `numberOfDays` variable to increase fees.

```
pragma solidity 0.8.21;

import "../TestBaseAeraVaultV2.sol";
import {IERC20Metadata} from "@openzeppelin/IERC20Metadata.sol";
import {console2} from "forge-std/Test.sol";

contract SubmitTest is TestBaseAeraVaultV2 {
    Operation[] public operations;

    function setUp() public override {
        super.setUp();

        // Zero out all token balances for simplicity.
        for (uint256 i = 0; i < assets.length; i++) {
            deal(address(assets[i]), address(vault), 0);
        }

        // Give vault 1000 USDC, where USDC is erc20Assets[1].
        uint8 decimals = IERC20Metadata(address(erc20Assets[1])).decimals();
        uint256 vaultBalance = 1000 * 10 ** decimals;
        deal(address(erc20Assets[1]), address(vault), vaultBalance);
    }

    function logState(string memory name) internal view {
        console2.log("\n", name);
        console2.log("fees accrued         ", vault.fees(vault.feeRecipient()));
        console2.log("Vault USDC balance   ", erc20Assets[1].balanceOf(address(vault)));
        console2.log("Guardian USDC balance", erc20Assets[1].balanceOf(_GUARDIAN));
    }

    function logProperties() internal view {
        uint256 numERC20Assets = erc20Assets.length;
        console2.log("PROPERTIES");
        console2.log("numERC20Assets", numERC20Assets);
        console2.log("maxDailyExecutionLoss", hooks.maxDailyExecutionLoss());
        console2.log("fee", vault.fee());
    }

    function submitOperations() internal {
        // Add target/selector for all operations.
        for (uint256 i = 0; i < operations.length; i++) {
            hooks.addTargetSighash(
                operations[i].target, IERC20.transfer.selector
            );
        }

        // Execute the operation.
        vm.expectEmit(true, true, true, true, address(vault));
        emit Submitted(vault.owner(), operations);

        vm.prank(_GUARDIAN);
        vault.submit(operations);
    }
```

```
    function test_submit_success_baseline() public {
        // --- Baseline ---
        // Log properties of the test
        logProperties();

        // Submit the operations and log state before and after. Operation is a no-op.
        logState("INITIAL CONDITIONS");
        operations.push(
            Operation({
                target: address(erc20Assets[1]),
                value: 0,
                data: abi.encodeWithSignature(
                    "transfer(address,uint256)", makeAddr("dummyRecipient"), 0
                )
            })
        );

        vm.warp(block.timestamp + 1 days); // Warp so fees accrue.

        submitOperations();
        logState("FINAL CONDITIONS");
    }

    function test_submit_success_attack() public {
        // --- Attack ---
        // Log properties of the test
        logProperties();

        // Compute an amount we can send to the vault and then withdraw
        // without triggering the `maxDailyExecutionLoss` revert.
        uint256 vaultBalance = erc20Assets[1].balanceOf(address(vault));
        uint256 amount = vaultBalance * hooks.maxDailyExecutionLoss() / 1e18;

        // Give that amount to the guardian
        deal(address(erc20Assets[1]), _GUARDIAN, amount);

        // Send that amount out as the only operation
        operations.push(
            Operation({
                target: address(erc20Assets[1]),
                value: 0,
                data: abi.encodeWithSignature(
                    "transfer(address,uint256)", _GUARDIAN, amount
                )
            })
        );

        // Submit the operations. Immediately before submitting, simulate a transfer
        // of `amount` USDC directly to the vault.
        logState("INITIAL CONDITIONS");
        uint256 numberOfDays = 1;
        for (uint256 i = 0; i < numberOfDays; i++) {
            vm.warp(block.timestamp + 1 days); // Warp so fees accrue.
            vm.prank(_GUARDIAN);
            erc20Assets[1].transfer(address(vault), amount);
            submitOperations();
        }
        logState("FINAL CONDITIONS");
    }
}
```

Note that this is similar in nature to the guardian stealing `maxDailyExecutionLoss` of tokens directly from the vault. However, this should be harder to detect as it's less direct and the total assets held by the vault remains unchanged.

Recommendation: There are a few approaches to consider:

1. Use internal accounting instead of `balanceOf()` calls. This means tokens transferred directly to the vault are not counted towards the vault value immediately. The difference between the internal balance and true balance can be used to determine the appropriate course of action. It may be desirable to only allow the owner to sync mismatched balances.
2. The check can done off-chain by setting up monitoring to look for unexpected fee increases.
3. Prevent the guardian from directly transferring out tokens with a hook that verifies the destination addresses for transfers and only allows some addresses. In practice this may be difficult to enforce as there are many ways to transfer value out.

Aera: We will take approach 2. We will be monitoring fees in the front-end and there's less chance of human error here.

Spearbit: Acknowledged.

5.2.7 `try/catch` statement may be manipulated to force usage of stale prices

Severity: Medium Risk

Context: AeraVaultV2.sol#L562-L567

Description: By default, calls forward 63/64ths of remaining gas. A `try/catch` call will fail and end up in the `catch` block if the call runs out of gas, which leaves the 1/64th of gas for the caller (AeraVaultV2) to finish execution.

If stale prices are more favorable than current prices to a user, they may be able to choose a gas limit to force the `spotPrices` call to revert—and therefore a stale price gets used—even if the oracle is behaving normally. The remaining 1/64th of gas needs to be sufficiently high for the rest of the execution to succeed.

The `spotPrices` method loops over up to 50 assets, so there may be a crossover point at which this becomes possible for each method that calls this internal `_reserveFees` method, based on how much gas it needs to finish after the `spotPrices()` call. Examples of how this may be leveraged:

- If asset prices crash, a guardian may want to use stale price so they can accrue more fees, since values and prices would be higher than they should be.
- If asset prices spike, the owner may want to use stale prices so they can reduce the fees that accrue.

Currently, this manipulation is difficult to detect due to the silent `catch` block.

Recommendation: There are a few mitigations to consider:

1. Ensure that when an oracle call reverts for legitimate reasons, it returns data with a length greater than zero. Chainlink oracles are `view` methods that should never revert, but this behavior may change. Additionally, if custom "pass through" oracles are used to derive a price for a missing oracle, those may revert due to e.g. overflow. An example of proper handling that is present is the existing revert with an `Aera__OraclePriceIsInvalid` error when `price <= 0`. If valid oracle failure methods revert with data and bubble it up properly, `spotPrices()` can similarly bubble up any errors. Now the `try/catch` can discern the failure method: An out of gas revert has no revert data, so if the revert data is empty the call likely ran out of gas, and the vault can revert in that case. If the revert data is non-empty, it was likely a legitimate revert (e.g. overflow, etc.) so the vault can continue with the stale prices.
2. Emit an event in the `catch` block so there's a log of the failure. This allows users to more easily detect when the `catch` block executes, and they can check if the oracle was down at that time, or inspect the transaction trace to see if the `spotPrices` call ran out of gas. This allows users to determine if the failure was likely intentional, so they can decide whether any relationships need to be terminated. Additionally, document this attack vector so owners, guardians, and fee recipients know to lookout for it to ensure all parties are acting honestly.

3. Disallow the use of stale prices altogether. This may not be ideal, as there are legitimate use cases where the guardian may need to adjust the vault holdings even when oracles are down.

Aera: We will implement recommendations 1 & 2. Fixed in PR 193.

Spearbit: Fixed.

5.2.8 Using Chainlink price feeds on some L2s requires a sequencer check

Severity: Medium Risk

Context: AeraVaultAssetRegistry.sol#L372

Description: The `spotPrices` method in `AeraVaultAssetRegistry` makes use of Chainlink's `latestRoundData` API, which returns the latest price data from a price feed. The problem is that Aera's team has stated that the protocol will possibly be deployed to multiple EVM-compatible chains, one of which is Arbitrum for example. As you can see in the Chainlink docs this requires a check if the sequencer is currently up, and if it isn't the price shouldn't be used.

Recommendation: Follow the Chainlink docs here to add a sequencer check, but only in the cases where the protocol is deployed on an L2 - you can add a flag to indicate this.

Aera: Recommendation accepted. Fixed in PR 199.

Spearbit: Fixed. It is also worth noting that if the sequencer is down, then all code that calls `spotPrices` would revert, which can block important admin actions (maintenance of the vault for example). Still, I think there is no easy solution to this, as the sequencer won't be working and you can't use another fallback oracle on your chain.

5.2.9 Chainlink price feed oracle staleness is not checked

Severity: Medium Risk

Context: AeraVaultAssetRegistry.sol#L372

Description: The `spotPrices` method in `AeraVaultAssetRegistry` makes use of Chainlink's `latestRoundData` API, which returns the latest price data from a price feed. There is some input validation (checking if `answer` is `<= 0`) but it is insufficient, as there is no price staleness check. This can result in the application using an incorrect price when valuing the assets in a vault.

Recommendation: For each price feed oracle read the Chainlink documentation to understand the heartbeat (max update time), and configure the `AeraVaultAssetRegistry` to check that the `answer` received for a price feed was more recent than `PRICE_FEED_HEARTBEAT + 1 hour`, otherwise revert as the price is possibly stale.

Aera: Recommendation accepted. Fixed in PR 227.

Spearbit: Fixed.

5.2.10 Removal of assets in the registry can be DOSed

Severity: Medium Risk

Context: AeraVaultAssetRegistry.sol#L241

Description: The code in `AeraVaultAssetRegistry::removeAsset` has the following check:

```
if (IERC20(asset).balanceOf(custody) > 0) {
    revert Aera__AssetBalanceIsNotZero(asset);
}
```

Anyone can front-run calls to `removeAsset` by transferring 1 wei of `asset` to the `custody` address, making the check above revert every time. This is a common DoS attack vector (SCSVS::G4.5).

Removing this check would allow the accidental removal of a token and then leave the token balance outside of the protection of the hooks and the `DailyExecutionLoss` protection, as it is no longer a registered token.

Recommendation: Here are some possible solutions:

- Cache the transferred balances in the vault's storage, instead of using `balanceOf` (adds registration complexity).
- Swap any remaining token to a registered token (introduces other external dependencies).
- Allow removal if there is a value less than some threshold, for example 100 usd (requires calling the oracle for the token).

Aera: We will separately consider whether to make this deployment atomic as per the other issue. Fixed in PR 158.

Spearbit: Fixed.

5.2.11 Fees may never accrue in some scenarios

Severity: Medium Risk

Context: AeraVaultV2.sol#L575-L578

Description: This equation performs divisions before multiplications, resulting in precision loss. Additionally, since the divisions both round down, it's possible to have a scenario where fees never accrue because they round down to zero. After we rearrange this equation so all division occurs last (which minimizes the chance of fees rounding down to zero), we can demonstrate situations in which fees round down to zero as follows:

The condition for no fees to accrue is when: `10 ** feeTokenDecimals * lastValue * feeIndex * fee < ONE * lastFeeTokenPrice`. The main variable an attacker has control over is `feeIndex`, which is the time between fee accruals. If called frequently enough, fees will round down to zero and never occur.

The first way this can happen is due to the public `claim()` method. This method reverts when `fees[msg.sender] == 0` so it cannot be continuously called by an arbitrary attacker. However, there are still two scenarios where prior fee recipients can continually call this to prevent fee accrual.

1. If there are many past fee recipients with nonzero accrued fees, they can coordinate their calls to `claim()` to prevent more fees from accruing. They can only do that until they each claim, so it this attack would not last very long, and is unlikely.
2. If there are insufficient `feeToken`s to fully pay the claim, a single `feeRecipient` can call `claim()` each block. They have a nonzero balance to claim so fees will continue to accrue, and their available balance to claim will not be deducted. To stop this, more `feeToken` would need to be transferred to the vault so their next `claim()` fully pays out, preventing them from calling it again in the future. Alternatively, the guardian can submit a transaction that swaps some of a vault's token with the required amount of fee token.

Another way this can occur is if the owner continuously calls a method such as `execute` as a no-op just to accrue fees. However, the trust model of Aera is such that if the owner is acting in any way to interfere with fees received by guardian, the service relationship is effectively terminated and the guardian will stop submissions. Consequently, this is not a concern as the relationship will be terminated in that scenario.

The last scenario is the guardian inadvertently driving fees to zero through their active management, meaning we want to solve the above equation for `feeIndex`. Rearranging gives: `feeIndex < 1e18 * lastFeeTokenPrice / (10 ** feeTokenDecimals * lastValue * fee)`. When that is true, no fees will accrue.

Now we choose worst-case but realistic values for each variable to find bounds on `feeIndex`. We'll use GUSD (2 decimals) here.

- `lastValue = 5e6` gives $50,000 in GUSD which is a reasonably sized small vault.
- `fee = 1e7`, which is ~0.0315% (100x lower than the max fee).
- With GUSD as the numeraire and fee token, `lastFeeTokenPrice = 100`

Solving gives `feeIndex < 20,000`, meaning if the guardian executes actions more frequently than every 20,000 seconds (~5.55 hours) they will not accrue fees. This management frequency is plausible for an actively managed

vault. Increasing the fee by 10x to 0.315%, this becomes 2000 seconds (~33 minutes), and with the max fee it's 200 seconds (~3.3 minutes).

Re-solving with USDC (6 decimals, `50,000e6` value, `1e6` fee token price) for a `1e7` fee, we get a value of 2 seconds. In practice a guardian will not need to submit a transaction every 2 seconds, so it's not an issue in this case.

Scaling the numeraire up to 18 decimals would not mitigate this, because doing so scales both `lastFeeTokenPrice` (numerator) and `10 ** feeTokenDecimals` (denominator) proportionally, so they cancel out. Shrinking `lastValue` and `fee` values are the drivers that allow a larger `feeIndex` to cause no fee accrual.

Overall, this issue is not likely to occur for reasonably sized vaults for tokens that have 6+ decimals. The "insufficient fee token" scenario can be recovered from in a straightforward manner without too many fees lost. The "active guardian" scenario can only be recovered from by adding more assets (which may not be feasible if the owner has no other assets), or by starting a new vault with a larger fee (which may not be desirable for the owner), and therefore this may result in the guardian and owner unable to agree upon terms that work for certain combinations of tokens, fee, and vault size.

Recommendation: First, rearrange the equation so all divisions are performed last. This maximizes the result of the numerator, making this less likely to occur.

Then, there are a few additional mitigations to consider. Mitigations 2 or 3 both fully resolve the issue, obviating the need for mitigations 1 and 4:

1. Document this limitation:
    1. Ensure users are aware of how fees can be stolen in this manner, particularly for the "insufficient fee token" and "active guardian" scenarios.
    2. For the "insufficient fee token" scenario, document the solution of transferring more `feeToken` so an attacker's next `claim()` pays out.
    3. For the "active guardian" scenario, before deploying a vault, plug in expected values to ensure fees will accrue for those values even with an active guardian.
2. Change the division to round up instead of down, either by manually rounding up or by using a library like solmate.
3. Have `_reserveFees` be a no-op if accrued fees are zero, where the key part here is to not write an updated `lastFeeCheckpoint` to storage.
4. Ensure a minimum amount of time or blocks have elapsed before doing the next fee calculation. The longer time that must elapsed between fee accruals, the less likely this is to occur. This is an alternative to (3).

Aera: We will rearrange the equation so all divisions are performed last. We will also implement 3. Fixed in PR 160 and PR 156.

Spearbit: Fixed.

5.2.12 `finalize` may revert in some scenarios

Severity: Medium Risk

Context: #L341-L377

Description: There three ways in which the `finalize` method may revert:

1. It may exceed the block gas limit. This function does a lot, but the main source of risk comes from having tokens (such as Aave's aTokens) where the transfer cost is not static—it is dependent on user state in the protocol—and can cost upwards of 300k gas to deploy. You would need about all 50 tokens (where 50 is the maximum allowed number of assets) to have similarly expensive `transfer` costs to have a chance of this occurring.
2. If any single transfer fails, which can occur if:
    1. The vault holds receipt tokens that are collateral for a borrow position. The tokens would not be transferrable if transferring them would make the vault insolvent on that position.

2. The vault is deny-listed, such as on USDC, USDT, or other tokens that support this feature.

Recommendation: The `finalize` method may be needed in emergency situations to recover all assets, so minimizing the chance of it reverting is ideal.

(1) Can be mitigated by being careful about which tokens are added to a vault. Before adding a new token to the registry and having the vault hold it, consider simulating a `finalize` call using a forge script to make sure the vault can still finalize. Additionally, the owner can instead call `withdraw` first to get out the majority of tokens, though an attacker can still grief the `finalize` call by sending small amounts of token before the call. This griefing can potentially be mitigated by setting a dust threshold and only transferring amounts above that threshold.

(2.1) Can be mitigated similarly, by calling `withdraw` to retrieve the majority of tokens. Alternatively, the owner can exit any positions first, then call `finalize`.

(2.2) In this situation the denylisted tokens cannot be withdrawn, so calling `withdraw` is the only option.

Document these limitations and workarounds so vault owners are prepared to call `finalize` and deal with any unexpected reverts.

Aera: Recommendation accepted. We will document these guidelines.

Spearbit: Acknowledged.

5.2.13 Malicious Tokens, Oracles and erc4626 could undo `ExecutionLoss` checks

Severity: Medium Risk

Context: AeraVaultHooks.sol

Description: Malicious Tokens, Oracles and erc4626 could undo `ExecutionLoss` checks.

All these malicious contracts could artificially inflate the token value or token balances. In cooperation with the guardian they could figure out the exact amount of required manipulation in for the following way:

- The gardian, as the first action of a submit, triggers an action at the malicious contract which retrieves the `value();`
- The gardian, as the first action of a submit, triggers an action at the malicious contract again, which inflates the result, just enough to pass the `ExecutionLoss` checks.

This way the guardian could compensate for losses or stolen funds. Also the fees could be inflated.

There will be a problem though when the owner tries to `withdraw()` or `finalize()` and exchange the tokens. Then it turns out the funds are not real.

Recommendation: The tokens, oracles and erc4626 contracts have to be selected carefully. This might not be trivial because the `owner` might not have the right knowledge to do that.

Possible solution: Have a function that `withdraw()`s the funds and verifies the received funds are real, for example by exchanging them for well known tokens. After this check the function could revert so the vault remains intact. The check function could be called periodically.

Aera: We agree. Our current policy is that the Aera protocol team maintains a global internal whitelist of ERC20, oracle and ERC4626 addresses that are carefully vetted. The owner will only select addresses from those. If they choose an address that is outside this list, the guardian will not be under any obligation to make submissions. We will implement whitelist compliance checks in our offchain monitoring.

Spearbit: Acknowledged.

5.2.14 Owner can circumvent `_checkReservedFees()` in `execute()`

Severity: Medium Risk

Context: AeraVaultV2.sol#L309-L339

Description: The function `_checkReservedFees()` in `execute()` makes sure the owner doesn't retrieve the last `feeTokens`, that are meant for the `feeRecipient`. However the `owner` can circumvent this check by setting an `allowance` for the `feeTokens`. The reduces the use of this check. Note: this could also grief the guardian if he isn't aware of this and sets allowances on the same token. In that situation `afterSubmit()` might revert.

```
function execute(Operation calldata operation) external override onlyOwner {
    ...
    _checkReservedFees(prevFeeTokenBalance);
    ...
}
```

Recommendation: Consider to disallow the owner to set an `allowance` for the `feeTokens`. Alternatively consider removing the `_checkReservedFees()` check, because it also has other downsides, see issue "Last resort function `execute()` could fail".

Aera: We will remove `_checkReservedFees` altogether and instead will expect that guardians will be claiming fees regularly. Fixed in PR 1821.

Spearbit: Fixed.

5.2.15 Last resort function `execute()` could fail

Severity: Medium Risk

Context: AeraVaultV2.sol

Description: The function `execute()` is a last resort function that can be used when other methods fail. However, the logic around fees can still revert, even though there is a `try/catch` in `_reserveFees()`. If these revert then the last resort doesn't work, which might lock funds in the vault. The main reasons for reverts could be:

- Buggy or malicious tokens.
- Deny list of a token, where `balanceOf` or `decimals` reverts (not likely).
- Interaction with ERC4626 contracts.

Note: `_reserveFees()` returns directly if `finalized` is set, but it might never reach this state due to issues with tokens, ERC4626 contracts or oracles.

```
function execute(Operation calldata operation) external override onlyOwner  {
    ...
    _reserveFees(); // could revert
    uint256 prevFeeTokenBalance =  assetRegistry.feeToken().balanceOf(address(this)); // could revert
    // actions
    _checkReservedFees(prevFeeTokenBalance); // could revert
    ...
}
function _reserveFees() internal {
    if (fee == 0 || paused() || finalized) {
        return;
    }
    ...
    try assetRegistry.spotPrices() returns (
        IAssetRegistry.AssetPriceReading[] memory erc20SpotPrices
    ) {
        (lastValue, lastFeeTokenPrice) = _value(erc20SpotPrices, feeToken);  // could revert
    } catch {}
    ...
```

```
    uint256 newFee = ( ((lastValue * feeIndex * fee) / ONE)
        * 10 ** IERC20Metadata(address(feeToken)).decimals()  // could revert
        ) / lastFeeTokenPrice;
}
function _checkReservedFees(uint256 prevFeeTokenBalance) internal view {
    uint256 feeTokenBalance = assetRegistry.feeToken().balanceOf(address(this)); // could revert
    ...
}
function _value(...) ... {
    ...
    AssetValue[] memory assetAmounts = _getHoldings(assets); // could revert
    (...) = _getSpotPricesAndUnits(assets, erc20SpotPrices); // could revert
    ...
    balance = IERC4626(address(assets[i].asset)).convertToAssets(assetAmounts[i].value);  // could
    ↪  revert
    ...
}
function _getHoldings(IAssetRegistry.AssetInformation[] memory assets) ... {
    ...
    ...  value: asset.asset.balanceOf(address(this))  // could revert
    ...
}
function _getSpotPricesAndUnits(...) ... {
    ...
    underlyingAsset = IERC4626(address(asset.asset)).asset(); // could revert
    ...
    assetUnits[i] = 10 ** IERC20Metadata(underlyingAsset).decimals(); // could revert
    ...
```

Recommendation: Carefully select the tokens and ERC4626 contracts, also see issue "Malicious Tokens, Oracles and erc4626 could undo `ExecutionLoss` checks".

Consider wrapping the fee logic in `try/catch`. Note this is more difficult to do if `modifiers` are used, see issues:

- "Ordering of `_reserveFees()`"
- "Ordering of `_checkReservedFees()`"

Consider removing the fee logic from `execute()`. The `_reserveFees()` will likely only add a small number of fees and is the main responsibility for the guardian to do. The `_checkReservedFees()` is easy to circumvent, also see issue "Owner can circumvent `_checkReservedFees()` in `execute()`".

Aera: Removed fee logic from `execute` in PR 173.

Spearbit: Verified. In theory `feeToken().balanceOf(address(this))` could also fail (which is also called in `_checkReservedFees()`). But this is very unlikely.

5.2.16 Life cycle of hooks contracts

Severity: Medium Risk

Context: AeraVaultHooks.sol#L33-L36

Description: A hooks contract can be switched to another hooks contract via `setHooks()`. The new hooks contract could also be a hooks contract that has been used previously. When switching to a new version, the variables from the old contract are no longer relevant. That includes:

- `currentDay`
- `cumulativeDailyMultiplier`
- `_pendingOwner`

This could allow exceeding the 'real' max daily loss, via reset of the `cumulativeDailyMultiplier`.

However, when switching to a previously used hooks contract, the old values of these variables are relevant again. The influence of `currentDay` and `cumulativeDailyMultiplier` is limited. However, an old `_pendingOwner` might be unexpected. Additionally, future versions of hooks might have other relevant variables.

```
contract AeraVaultHooks is IHooks, ERC165, Ownable2Step {
    ...
    uint256 public currentDay;
    uint256 public cumulativeDailyMultiplier;
    ...
}
```

Recommendation: Consider having a hook before deinstallation and right after installation. This can be used to clear and/or set variables.

`Owners` could pause their vaults during the update process until the `cumulativeDailyMultiplier` would effectively reset on the new hook as on the old, making for a smoother and safer transition.

Also see issues:

- "Owner can manipulate hooks"
- "Access to inactive hooks contracts"

Aera: Fixed in PR 161. Hooks now have a `decomission()` function.

Spearbit: Verified.

5.2.17 Lack of `assetRegistry.custody` check upon being set can lead to invalid vault values and permanent loss of funds

Severity: Medium Risk

Context: AeraVaultV2.sol#L127, AeraVaultV2.sol#L150, AeraVaultV2.sol#L800-L816

Description: When an `AeraVaultV2` is deployed, its constructor expects an `AeraVaultAssetRegistry` contract to be set, which tracks a list of registered ERC20 and ERC4626 assets that can be used within the vault, containing necessary data to complete operations with them. The vault currently only has some basic sanity checks when setting this registry, such as it being non-zero address and signalling implementation for the expected `IAssetRegistry` interface. This means, an `AeraVaultAssetRegistry` contract could be currently connected to the vault, where the asset registry was intended for a whole different vault altogether.

This is a one-shot operation and irreversible. General functionality is not affected until certain conditions are met, so vaults could mistakenly or unknowingly have this occur to them, and begin operating on vaults as normal by depositing funds into it. This could also be purposefully encouraged by a malicious vault guardian or even other third-party wishing to blackmail vault users.

The main danger condition possible by this is that a vault owner may deposit tokens initially registered in the registry. Since the registry points to a different vault though, those tokens can be unregistered, even though they are still in our vault. This would effectively blackhole their value from the vault (referred hereinafter as 'stealthed' tokens). This opens a number of possible attack scenarios depending on the controlling party of the registry (although ownership isn't a requirement to execute these scenarios aside from the blackmail one).

- Avoidance of fees by vault owners by 'stealthing' value.
- Exceeding max daily real execution loss by vault guardian, or even complete extraction of 'stealthed' tokens.
- Blackmail by a third party, as the simple `withdraw` function would be rendered inoperable on 'stealthed' tokens.
- Stuck funds pertaining to the unregistered tokens, if `finalize` is called, only reversible by an `execute` recovery, could occur under any ownership scenario.

These are some of the possible attacks by various actors upon one scenario of an invalid `custody` being in the registry contract. The attack surface is quite wide with the following functions being impacted which could lead to additional unaccounted attack scenarios:

- `finalize`
- `withdraw`
- `deposit`
- `submit`
- `execute`
- `claim`
- `setHooks`
- `pause`
- `setGuardianAndFeeRecipient`

A power user or recovery setup by authors provided to owners should be possible using `execute`, except in the case of a guardian using such a scenario to steal the 'stealthed' tokens.

Recommendation: Consider an atomic deployment mechanism for these contracts as specified in *"Atomic deployments can simplify instance setup and reduce risk"*. A proper implementation would fix this, however it could entail a significant overhaul to the contracts.

For the current contracts and deployment design, add a requirement within `_checkAssetRegistryAddress` that the asset registry candidate has this vault set as its `custody` variable, following confirmation of the appropriate `IAssetRegistry` interface being supported.

```
 function _checkAssetRegistryAddress(address newAssetRegistry) internal view {
        ...
+       // confirmed registry interface, let's check we are the intended vault
+       address registryCustody = address((newAssetRegistry).custody())
+       if (registryCustody != address(this))
+               revert Aera__AssetRegistryHasInvalidCustody(newAssetRegistry, registryCustody);
    }
```

A custom error should be added to signify failure on this check and the `custody` variable exposed in the `IAssetRegistry` interface. The team could also consider creating an external getter of the `custody` variable that simply returns its corresponding address.

This may seem problematic at first glance, since we expect the `custody` variable to be set to that of a newly deployed contract via its constructor, however, this can be accomplished due to it being deployed via `CREATE2`, and the address can be precomputed and set in the registry prior to its deployment. This will require the vault interface (currently `ICustody`) check to be skipped by the registry, as the vault would not be deployed yet.

```
 function setCustody(address newCustody) external onlyOwner {
...
        // Requirements: check that there is an ICustody contract at the new address.
        if (newCustody == address(0)) {
            revert Aera__CustodyIsZeroAddress();
        }
-       if (
-           !ERC165Checker.supportsInterface(
-               newCustody, type(ICustody).interfaceId
-           )
-       ) {
-           revert Aera__CustodyIsNotValid(newCustody);
-       }

        // Effects: set custody address.
        custody = newCustody;
...
```

Aera: Recommendation accepted. Fixed in PR 151.

Spearbit: Verified.

5.2.18 Lack of `custody` address check in `setHooks` function can lead to most functions reverting while unpaused, until properly reset hook

Severity: Medium Risk

Context: AeraVaultV2.sol#L288-L306, AeraVaultV2.sol#L818-L829

Description: `AeraVaultV2` depends on a `AeraVaultHooks` contract as an upgradeable protection layer for vault owners. When a new hook is set, the current checks include ensuring that it's a non-zero address and that it supports the `IHooks` interface. This in turn, could allow an invalid `AeraVaultHooks` contract to be set for the vault, where the `custody` variable within the hook contract is not actually set, to the setting vault contract.

This would render any hook dependent functions to revert and not work even in an unpaused state, as the vault would lack the access control necessary to actually use those hooks.

These functions include

- deposit
- withdraw
- execute
- finalize
- submit

Fees could keep accruing during this time if unpaused, so the vault could be being charged fees, even though it's in an invalid state where most of its core functionality doesn't work.

This is reversible and fixable even with a deployed vault, by resetting to a proper hook that has the appropriate `custody` address set, with the `setHook` call.

Recommendation: To avoid the possibility of such a scenario and even entering into this invalid state, mistakenly or purposefully, the requirements section within `setHooks` via the internal `_checkHooksAddress` call should also include a check of the `custody` variable being equal to the vault's address within the hook contract about to be set, following ERC165 detection of the expected `IHooks` interface on the given `newHooks` address.

Consider the following as a candidate for the code revision

```
 function _checkHooksAddress(address newHooks) internal view {
        ...
+       // confirmed hooks interface, let's check we are the intended vault
+       address hooksCustody = address(IHooks(newHooks).custody())
+       if (hooksCustody != address(this))
+               revert Aera__HookHasInvalidCustody(newHooks, hooksCustody);
    }
```

Note: getter for `custody` will need to be exposed in `IHooks`, where it should be available as it's a public variable. Also likely a new custom error code introduced `Aera__HookHasInvalidCustody(address, address)`

Aera: Fixed in PR 155.

Spearbit: Verified.

5.2.19 Ordering of `_reserveFees()`

Severity: Medium Risk

Context: AeraVaultV2.sol#L227-L266, AeraVaultV2.sol#L627-L668, AeraVaultV2.sol#L731-L760

Description: Assume `_reserveFees()` hasn't been called for a relatively long period. Then when `withdraw()` is called, it calls `_checkWithdrawRequest()`. Function `_checkWithdrawRequest()` calls `_getHoldings()`, which returns the available value for `feeToken`, excluding the reserved fees (`feeTotal`). As `feeTotal` hasn't been updated yet, this allows for possibly withdrawing too much `feeToken`, leaving the vault insolvent.

The update of the `feeTotal` via `_reserveFees()` only happens after `_checkWithdrawRequest()`. Note: the amount of insolvency depends on how long ago the previous call to `_reserveFees()` was and only occurs if the bulk of the `feeToken` is withdrawn, so in practice this would be a small amount. Note: set to medium risk as this doesn't fit the invariants of Aera

```
function withdraw(AssetValue[] calldata amounts) ... {
    ...
    // Requirements: check the withdraw request.
    _checkWithdrawRequest(assets, amounts);
    // Effects: reserve fees for fee recipient.
    _reserveFees();
    ...
}
function _checkWithdrawRequest(...) ... {
    ...
    AssetValue[] memory assetAmounts = _getHoldings(assets);
     ...
    for (uint256 i = 0; i < numAmounts;) {
    ...
        if (assetAmounts[assetIndex].value < assetValue.value) {
            revert Aera__AmountExceedsAvailable(...);
        }
        ...
    }
    ...
}
function _getHoldings(IAssetRegistry.AssetInformation[] memory assets) ... {
    ...
    for (uint256 i = 0; i < numAssets;) {
        ...
        if (asset.asset == feeToken) {
            if (assetAmounts[i].value > feeTotal) {
                assetAmounts[i].value -= feeTotal;
            } else {
                assetAmounts[i].value = 0;
            }
        }
        ...
    }
    ...
}
```

Recommendation: Consider making a `modifier` that calls `_reserveFees()`, which reduces ordering mistakes. Note consider combining this with the recommendations from issue: "Ordering of `_checkReservedFees()`".

Alternatively, in function `withdraw()` call `_reserveFees()` as the first thing, as shown below:

```
 function withdraw(AssetValue[] calldata amounts) ... {
     ...
+    _reserveFees();
     _checkWithdrawRequest(assets, amounts);
-    _reserveFees();
     ...
}
```

Aera: Fixed in PR 143.

Spearbit: Verified.

5.2.20 Disable `renounceOwnership()`

Severity: Medium Risk

Context: Ownable.sol#L61-L63, Ownable2Step.sol#L44-L47, AeraVaultV2Factory.sol#L11, AeraVaultV2.sol#L18-L23, AeraVaultHooks.sol#L18, AeraVaultAssetRegistry.sol#L15

Description: The contracts `AeraVaultV2Factory`, `AeraVaultV2`, `AeraVaultHooks` and `AeraVaultAssetRegistry` inherit from `Ownable2Step`, which inherits from `Ownable`.

This means the function `renounceOwnership()` is present in all four contracts. If the function `renounceOwnership()` of `AeraVaultV2` would be accidentally called, then `withdraw()`, `finalize()` and `execute()` would no longer be possible and the funds would be locked in the vault. Removing the `owner` in `AeraVaultV2Factory`, `AeraVaultHooks` and `AeraVaultAssetRegistry` also reduces the functionality.

```
abstract contract Ownable is Context {
    function renounceOwnership() public virtual onlyOwner {
        _transferOwnership(address(0));
    }
}
abstract contract Ownable2Step is Ownable {
    function _transferOwnership(address newOwner) internal virtual override {
        delete _pendingOwner;
        super._transferOwnership(newOwner);
    }
}
```

Recommendation: Disable `renounceOwnership()`, for example by overriding it, in `AeraVaultV2Factory`, `AeraVaultV2`, `AeraVaultHooks` and `AeraVaultAssetRegistry`.

Aera: It has no significant consequence in hooks (hooks can be reset), asset registry (funds can still be withdrawn with `execute`) or factory (a new factory can be deployed). We will override `renounceOwnership` in the Aera vault with a revert. Fixed in PR 142.

Spearbit: Fixed.

5.3 Low Risk

5.3.1 Precision loss when computing `custody.value()`

Severity: Low Risk

Context: AeraVaultV2.sol#L590-L622

Description: The `custody.value()` calculation, used when computing `newMultiplier`, has divisions occurring before the final value is computed, resulting in precision error.

Recommendation: The `custody.value()` precision loss requires a larger refactor to fully mitigate because it would require a different way of handling prices, such as returning non-normalized prices and lazily normalizing in the `afterSubmit` hook. Because the precision error in this instance results in a more conservative computation, consider whether the added complexity is worth the precision.

Aera: Acknowledged. We are happy to accept this small precision loss as it is conservative for the vault value.

Spearbit: Acknowledged.

5.3.2 `execute` function should have `nonReentrant` guard

Severity: Low Risk

Context: AeraVaultV2.sol#L309-L313

Description: `execute` is a privileged function allowing owners to execute just about any call originating from the vault, aside from the hooks address being a target for reasons of security. Currently, it is trivial to perform reentrancy into it due to this flexibility and it has no reentrancy guards, even though other functions are protected by `nonReentrant`.

Recommendation: Add the `nonReentrant` modifier to `execute` to mitigate reentrancy into this function, and keep it a more predictable single operation function.

Aera: Recommendation accepted. Fixed in PR 200.

Spearbit: Fixed.

5.3.3 Disallow `execute` and `submit` operations targeting vault itself

Severity: Low Risk

Context: AeraVaultV2.sol#L309-L318

Description: The vault could `execute` or `submit` operations targeting itself. One of the possibilities this allows is to transfer the ownership of the vault to itself and accept it via `execute`. This would effectively brick the contract and any associated value still within. This would require 2 transactions to be done by an owner, but since all the operations would be essentially trusted targets (e.g. the vault itself), it may happen under social engineering scenarios. Note: for `submit` to work, the transaction should also be allowlisted.

Recommendation: Check to make sure that the vault itself is not a target of an `execute` or `submit` operation , and revert if it is. Update `submit` in a similar way.

```
 function execute(Operation calldata operation)
...
        if (operation.target == address(hooks)) {
            revert Aera__ExecuteTargetIsHooksAddress();
        }

+       // Requirements: check that the target contract is not vault itself.
+       if (operation.target == address(this)) {
+           revert Aera__ExecuteTargetIsVaultAddress();
+       }
```

Aera: Recommendation accepted. Fixed in PR 201.

Spearbit: Fixed.

5.3.4 Loop `removeAsset()` compares `asset` to itself

Severity: Low Risk

Context: AeraVaultAssetRegistry.sol#L229-L298

Description: Function `removeAsset()` does a loop over all assets to doublecheck if the `asset`, located at `oldAssetIndex`, can be deleted. Logically it should skip comparing to itself.

Luckily the present conditions prevent any harm, but future changes in the code might introduce issues.

```
function removeAsset(address asset) external override onlyOwner {
    ...
    if (_assets[oldAssetIndex].isERC4626) {
        numYieldAssets--;
    } else {   // now we know oldAssetIndex isn't ERC4626
        for (uint256 i = 0; i < numAssets; i++) {
            if (
                _assets[i].isERC4626                // so this will be false for oldAssetIndex
                    && IERC4626(address(_assets[i].asset)).asset() == asset   // the underlying asset
                    ↪  of oldAssetIndex shouldn't be asset either
            ) {
                revert Aera__AssetIsUnderlyingAssetOfERC4626( address(_assets[i].asset));
            }
        }
        ...
    }
    ...
}
```

Recommendation: Consider changing the code to:

```
 if (
+     i != oldAssetIndex &&
      _assets[i].isERC4626
      && IERC4626(address(_assets[i].asset)).asset() == asset
) { ... }
```

Aera: Recommendation accepted. Fixed in PR 202.

Spearbit: Fixed.

5.3.5 Use `msg.sender` as opposed to `owner()` in methods that have an `onlyOwner` modifier

Severity: Low Risk

Context: AeraVaultV2.sol#L169, AeraVaultV2.sol#L227, AeraVaultV2.sol#L309, AeraVaultV2.sol#L342

Description: Retrieving the `owner` is relatively expensive as it requires an `SLOAD`. In functions that are protected by an `onlyOwner` modifier, it is verified that `msg.sender` is the `owner`. Using `msg.sender` is a lot cheaper than using `owner`

Recommendation: Consider changing `owner()` to `msg.sender` in the referenced methods that use `onlyOwner`. This additionally resolves an edge case, where `execute()` could potentially call `acceptOwnership()`, and the `Executed` event would emit the new owner, instead of the correct original executing `owner`, which we get via `msg.sender`.

Aera: Recommendation accepted. Fixed in PR 208.

Spearbit: Fixed.

5.3.6 `_getSpotPricesAndUnits()` can be simplified and made safer

Severity: Low Risk

Context: AeraVaultV2.sol#L675-L726

Description: The function `_getSpotPricesAndUnits()` contains code duplication that can be reduced. This makes the code easier to understand and maintain. Also it doesn't explicitly check the asset has been found.

Recommendation: Consider changing the code as shown below. For safety an extra check could be added to make sure the asset is found.

```
function _getSpotPricesAndUnits(... ) ... {
    ...
    for (uint256 i = 0; i < numAssets;) {
        asset = assets[i];
        address tofind = (asset.isERC4626 ?  IERC4626(address(asset.asset)).asset() :  tofind =
        ↪  asset.asset ) ;
        for (uint256 j; j < numERC20SpotPrices;) {
            if (tofind == address(erc20SpotPrices[j].asset))
                break;
            unchecked { j++; } // gas savings
        }
        if (j >= numERC20SpotPrices) // possibly extra check not found
            revert ...;
        spotPrices[i] = erc20SpotPrices[j].spotPrice;
        assetUnits[i] = 10 ** IERC20Metadata(tofind).decimals();

        unchecked {
            i++; // gas savings
        }
    }
}
```

Aera: Fixed in PR 210. Check not implemented because it's won't happen in practice.

Spearbit: Verified. If the asset isn't found it will revert anyway so no risk.

5.3.7 `holdings()` and `value()` can be called while calls are being executed

Severity: Low Risk

Context: AeraVaultV2.sol#L497-L511

Description: The function `holdings()` and `value()` could be (indirectly) called while calls are being executed via `execute()` or `submit()`. At that moment the tokens are in flux so its important not to have anything that relies on the values. Otherwise, there could be an issue like a read-only reentrancy.

Potential situation: the vault is used as collateral for something and could be liquidated under certain situations (perhaps a bit far fetched).

```
function holdings() public view override returns (AssetValue[] memory) {
    ...
}
function value() external view override returns (uint256 vaultValue) {
    ...
}
```

Recommendation: If it is anticipated that something could rely on the value of the vault consider doing the following:

- check the value of the `nonReentrant` modifier in `holdings()` and `value()`, possibly allow reentracy from the hooks contract.

Note: it is still possible to retrieve the underlying values like token holdings, but that is unlikely to be used.

Depending on the reentrancy check, `value()` and `holdings()` may not be accessible from the hooks contract. `value()` is currently used and `holdings()` might be relevant for future versions of hooks. This could also be solved by passing the information from `value()` and possibly `holdings()` to the `before` and `after` hooks.

Also see issues:

- "Storing `_beforeValue` and `_beforeBalance` is relative expensive"
- "Expensive functions `_reserveFees()` and `value()` called both but are very similar"

Aera: We are far from these types of external integrations for Aera so will wait until we see how hooks develop in practice to take further action.

Spearbit: Acknowledged.

5.3.8 Precision loss when computing `newMultiplier`

Severity: Low Risk

Context: AeraVaultHooks.sol#L227-L234

Description: When computing `newMultiplier`, divisions occur before multiplications in the `if (currentDay == day)` block of `afterSubmit`.

The precision loss is not a big issue here because the precision loss results in a smaller value of `newMultiplier`. A smaller value is more conservative as it's more likely to revert from the `newMultiplier < ONE - maxDailyExecutionLoss` check.

Recommendation: Move all multiplications last with:

```
uint256 newMultiplier;
if (currentDay != day) {
    newMultiplier = custody.value() * ONE / _beforeValue;
} else {
    newMultiplier = cumulativeDailyMultiplier * custody.value() / _beforeValue;
}
```

Aera: Recommendation accepted. Fixed in PR 209.

Spearbit: Since there were previously two recommendations here (remove precision loss from `newMultiplier` computation itself, and remove precision loss from `custody.value()`), I've split this into two issues. This issue will be labeled Fixed, and *"Precision loss when computing `custody.value()`"* is the new issue which I've labeled as Acknowledged.

5.3.9 `_value` may revert for non-conforming ERC-4626 tokens

Severity: Low Risk

Context: AeraVaultV2.sol#L606-L608

Description: Per ERC-4626, `convertToAssets` must never revert. However, tokens don't always follow spec—if it does revert, `_reserveFees()` also reverts, and the vault will be stuck until the offending token is removed from the asset registry.

Recommendation: Only add ERC-4626 assets that have been verified to accurately conform to the ERC-4626 specification.

Aera: We will have strict asset whitelisting criteria for all whitelisted assets.

Spearbit: Acknowledged.

5.3.10 Add stronger validation on oracle addresses

Severity: Low Risk

Context: AeraVaultAssetRegistry.sol#L429-L432

Description: The oracle for an ERC20 token is validated by comparing the oracle address against the zero address. Consequently, invalid addresses—such as EOAs or contracts that don't conform to the expected interface—can be added as oracles, which will result in reverting transactions. In this case, the asset needs to be removed then re-added, as there is no way to directly update an oracle address.

Recommendation: A stronger oracle validation would be to call `latestRoundData` and verify that a valid and up to date value is returned, which would reduce the likelihood of adding an invalid oracle.

Aera: Recommendation accepted. Fixed in PR 215.

Spearbit: Fixed.

5.3.11 Native tokens could be temporarily inaccessible

Severity: Low Risk

Context: AeraVaultV2.sol#L590-L622, AeraVaultHooks.sol#L183-L243

Description: Via `submit()` any native tokens (ETH) that is left in the Vault can no longer be used due to the checks in `afterSubmit()`. Also `submit()` cannot easily know how much native tokens (ETH) are present after external calls, see issue "Inherent limits of `submit()`", so there are bound to be some native tokens (ETH) left in the vault (assuming native tokens (ETH) are being used at all). Furthermore, the native tokens (ETH) are not counted in the `value()` calculation so they add to the `DailyExecutionLoss`. The function `_reserveFees()` also doesn't take native tokens (ETH) into account.

The native tokens (ETH) can be rescued though by the `owner` via `execute()`.

```
function _value(...) ... {
    ...
    for (uint256 i = 0; i < numAssets;) {
        if (assets[i].isERC4626) {
            balance = IERC4626(address(assets[i].asset)).convertToAssets(
                assetAmounts[i].value
            );
        } else {
            balance = assetAmounts[i].value;
        }
        vaultValue += (balance * spotPrices[i]) / assetUnits[i];  // native tokens are not counted
        ...
    }
}
function beforeSubmit(Operation[] calldata operations) ... {
    ...
    _beforeBalance = address(custody).balance;
}
function afterSubmit(Operation[] calldata operations) ... {
    ...
    if (address(custody).balance < _beforeBalance) { // cannot use any ETH that is left in the vault
    ↪  from previous submit
        revert Aera__ETHBalanceIsDecreased();
    }
    ...
}
```

Recommendation: In function `submit()`: swap any left over native tokens (ETH) back to its wrapped version (WETH). This should be done right after all operations and before the calls to `_checkReserved-`

`Fees(prevFeeTokenBalance);` and `hooks.afterSubmit(operations)`. The WETH address is already known in 'AeraVaultV2. WETH should be added as an asset in the registry and shouldn't be allowed to be removed.

Aera: Fixed in PR 238 , with the following changes: In vault:

- check that WETH is part of the asset registry in the vault constructor
- perform the conversion to `WETH` at the end of `submit`

In asset registry:

- provide WETH as an asset to asset registry constructor
- add it as an asset
- prevent its removal in `removeAsset`
- we will confirm that it is added as an ERC20 asset
- NOTE: It could or couldn't have an oracle depending on if it's used as the numeraire

Spearbit: Verified.

5.3.12 Contracts can't rescue mistakenly sent ERC20 tokens or ETH

Severity: Low Risk

Context: AeraVaultAssetRegistry.sol,AeraVaultHooks.sol

Description: The `AeraVaultAssetRegistry` and `AeraVaultHooks` contracts currently have no way to rescue mistakenly sent ERC20 tokens to them, as well as ETH that is received through the following ways:

1. A contract selfdestructs and sends ETH to an aera contract.
2. An aera contract is set as the block's fee recipient.
3. An aera contract is set as the recipient for a staked ETH withdrawal.
4. Address of an aera contract is precomputed and ETH is sent before it's deployed.

Recommendation: Add a way to rescue ERC20 tokens and ETH that has been mistakenly sent to either the `AeraVaultAssetRegistry` or `AeraVaultHooks` contracts.

Aera: We will add a sweep function to both contracts that can be called by the owner. Fixed in PR 218.

Spearbit: Fixed.

5.3.13 Add a stronger than just a zero address check for `weth`

Severity: Low Risk

Context: AeraVaultV2Factory.sol#L44-L46

Description: The constructor of `AeraVaultV2Factory` checks if the `weth_` argument is the zero address to do input validation, but a stronger check would be to verify if the `weth_` address has code, or even if it returns a value when calling `balanceOf`. While more expensive, this is protecting you from deploying the contract with an EOA address for example.

Recommendation: In the constructor of `AeraVaultV2Factory` add a check if calling `balanceOf` of the `weth_`-argument returns a value or just check if the account in the given address has code.

Aera: We will perform the `balanceOf` check. Fixed in PR 213.

Spearbit: Fixed.

5.3.14 Function `submit()` doesn't have `whenHooksSet`

Severity: Low Risk

Context: AeraVaultV2.sol

Description: The function `submit()` doesn't have the modifier `whenHooksSet`, while other functions like `deposit()`, withdraw(), and finalize()do have this. It isn't absolutely necessary because initially the contract is paused andresume()can only be called when the hooks are set. In turnsubmit()‘ can't be run in the paused state.

However, for consistency and to prevent future mistakes with the pause logic, it would be good to also have the `whenHooksSet` in function `submit()`.

```
function deposit(...) ...  whenHooksSet ... {
}
function withdraw(...) ... whenHooksSet ... {
}
function finalize() ... whenHooksSet  ... {
}
function submit(...) ... whenNotPaused ... {
}
function resume() ... whenHooksSet ... {
    ...
    _unpause();
}
```

Recommendation: Consider adding the `whenHooksSet` modifier to function `submit()`.

Aera: Recommendation accepted. Fixed in PR 212.

Spearbit: Fixed.

5.3.15 In edge case `resume()` might be inaccessible

Severity: Low Risk

Context: AeraVaultV2.sol#L342-L408

Description: The function `finalize` calls `hooks.beforeFinalize()`. Depending on the code of the hooks contract, and non trivial code at the guardian, this could potentially do a `reentrant` call to `pause()`. Note: `finalized` isn't set yet. After function `finalize`, `finalized` will be set and `resume()` can no longer be called because it has the `modifier whenNotFinalized`.

However this doesn't have consequences as the only useful function that relies on `whenNotPaused` is `submit()` and that can't run either due to `whenNotFinalized` and isn't relevant anymore after `finalize`.

```
function finalize() nonReentrant ... whenNotFinalized {
   ...
    hooks.beforeFinalize(); // external call, could do reentrant call to pause()
    // Effects: mark the vault as finalized.
    finalized = true;
    ...
}
function pause() ... whenNotFinalized { // no nonReentrant modifier
    ...
}
function resume() ... whenNotFinalized {
    ...
}
```

Recommendation: Determine if it is useful to be able to call `pause()` from the hooks contract. There might be cases if an inconsistency is detected, although a revert might be more appropriate then. Otherwise consider adding

the `nonReentrant` to `pause()`. Doublecheck the usefulness of `whenNotFinalized` at `resume()`. This function can only be used after `pause()` has been called.

Aera: We will mark `pause` as `nonReentrant` as for now pausing from a hook is very unlikely as it would require the guardian to be controlled by the hook which we do not envision.

We will keep the `whenNotFinalized` check on `resume` as it prevents the `lastFeeCheckpoint` to be modified which could affect accounting in the UI for us. Fixed in PR 219.

Spearbit: Fixed.

5.3.16 Access to inactive hooks contracts

Severity: Low Risk

Context: AeraVaultV2.sol#L309-L339, AeraVaultV2.sol#L411-L464

Description: The functions `execute()` and `submit()` prevent calling into the current hooks contract. However if a hooks contract is changed, then the protection doesn't work for the old version of the hooks contract. As that old version of the hooks contract could potentially be used again, this might give too much access.

The calls from `submit()` are also protected by an allow list, so that would also require manipulating the allow list. Also the potential functions that can called, are only the `before` and `after` hooks, which is harmless in the current hooks contract when the hooks contract is not in use.

The calls from `execute()` can only be done by the `owner` which already has rights on the hooks contract so doesn't introduce additional risks (unless the `owner` role is further separated.)

```
function execute(Operation calldata operation) external override onlyOwner {
   ...
    if (operation.target == address(hooks)) {  // only works for current version of hooks
        revert Aera__ExecuteTargetIsHooksAddress();
    }
    ...
}
function submit(Operation[] calldata operations) ... {
    ...
    for (uint256 i = 0; i < numOperations;) {
        ...
        if (operation.target == hooksAddress) { // only works for current version of hooks
            revert Aera__SubmitTargetIsHooksAddress();
        }
        ...
    }
}
```

Recommendation: Consider deleting the `custody` variable of the hooks contract in a deinstallation hook and perhaps setting it again in an installation hook. See issue "Life cycle of hooks contracts".

Alternatively keep a mapping of all (previous) hook contracts in the vault contract.

Aera: Recommendation of deleting the `custody` variable is accepted. This will be covered and tracked in our solution to *"Life cycle of hooks contracts"*. Fixed in PR 161.

Spearbit: Verified.

5.3.17 Owner can manipulate hooks

Severity: Low Risk

Context: AeraVaultV2.sol#L289-L306

Description: Hooks protect the effect of the following `owner` functions: `deposit()`, `withdraw()` and `finalize()`. The `owner` can manipulate hooks in the following ways, which circumvents the use of the hooks:

- call `setHooks()` with a less restrictive hook, call one of the protected functions and restore the hook via `setHooks()`;
- call `setHooks()` reentrant from a `hooks` contract to an `owner` contract to change the `after` hook.

As the `owner` has a lot possibilities anyway, in practice the hooks will be used to protect him against mistakes, so he normally wouldn't change the hooks. But an `owner` might be tricked into signing/executing unwanted transactions.

```
function setHooks(address newHooks)  external override onlyOwner whenNotFinalized {
    ...
    hooks = IHooks(newHooks);
    ...
  }
```

Recommendation: If you want to further improve the use of hooks, consider adding a `nonReentrant` modifier to `setHooks()`. Also consider having a hook before deinstallation and right after installation. This could perhaps be used to install a time lock.

Aera: Added decomission hook in PR 161. Added a `nonReentrant` modifier to `setHooks` in PR 220.

Spearbit: Verified as the suggestion have largely been implemented, however the primairy risk is still present as the owner has all the rights.

5.3.18 The `constructor` of `AeraVaultV2` uses old value of `owner`

Severity: Low Risk

Context: AeraVaultV2.sol#L117-L166, AeraVaultV2.sol#L777-L798

Description: The `constructor` of `AeraVaultV2` use the old value of `owner` to compare to `guardian_` and `feeRecipient_`. After these checks the `owner` is changed via `_transferOwnership(owner_)`. This make these checks not effective.

Also see issues:

- "Separation of roles".
- "Comments for `_transferOwnership` not accurate"

```
contract AeraVaultV2 is
    constructor(...) ... {
        ...
        _checkGuardianAddress(guardian_); // uses old owner
        _checkFeeRecipientAddress(feeRecipient_); // uses old owner
        ...
        _transferOwnership(owner_); // installs new owner
        ...
    }
   function _checkGuardianAddress(address newGuardian) internal view {
       ...
       if (newGuardian == owner()) {
           revert Aera__GuardianIsOwner();
       }
   }
    function _checkFeeRecipientAddress(address newFeeRecipient) internal view {
       if (newFeeRecipient == owner()) {
           revert Aera__FeeRecipientIsOwner();
       }
    }
}
```

Recommendation: Add a parameter to `_checkGuardianAddress()` and `_checkFeeRecipientAddress()` to supply the owner to be used in the comparison.

Aera: Fixed in PR 222.

Spearbit: Verified.

5.4 Gas Optimization

5.4.1 Redundant `unavailableFee` variable within `claim` scope

Severity: Gas Optimization

Context: AeraVaultV2.sol#L482-L493

Description: On L482 a `unavailableFee` variable is introduced. It is used only at the end of the scope, for an event emission. L484 is equivalent to L482 and the `reservedFee` variable following it will always share the same value.

```
uint256 unavailableFee = reservedFee - availableFee; // L482
...
reservedFee -= availableFee;  // L484 reservedFee now has the same value as unavailableFee
...
emit Claimed(msg.sender, availableFee, unavailableFee);
```

Recommendation: Drop the redundant `unavailableFee` declaration, and simply use `reservedFee`also for the event emission in its place, which will save some gas and avoid confusion for the need of the same value being stored in 2 different variables.

```
- uint256 unavailableFee = reservedFee - availableFee;
  feeTotal -= availableFee;
  reservedFee -= availableFee;

...
  // Log the claim.
- emit Claimed(msg.sender, availableFee, unavailableFee);
+ emit Claimed(msg.sender, availableFee, reservedFee);
```

Aera: Recommendation accepted. Fixed in PR 203.

Spearbit: Fixed.

5.4.2 Compiler settings may be changed to reduce bytecode size and gas usage

Severity: Gas Optimization

Context: foundry.toml#L1-L11

Description: Both existing foundry profiles do not use the solidity optimizer or the via-ir pipeline.

Using the standard optimizer can result in significant reductions to both bytecode size and gas usage. The optimizer's behavior can be configured by adjusting the number of runs which "specifies roughly how often each opcode of the deployed code will be executed across the life-time of the contract". A value of 1 produces less bytecode but more expensive code, but typically still cheaper than without the optimizer. A larger values increases bytecode size to produce more gas efficient code.

The standard optimization pipeline is complex, and may be a source of bugs. Using the via-ir pipeline provides similar benefits, but with less risk. Note that via-ir also introduces a few semantic changes.

Recommendation: Consider using the via-ir pipeline with the optimizer enabled to reduce bytecode size (which reduces deployment costs) and runtime gas usage. Be sure to run the test suite against contracts compiled with via-ir to ensure expected contract behavior is not impacted by the semantic changes.

Aera: Recommendation accepted. Fixed in PR 232.

Spearbit: Fixed.

5.4.3 Possibility for unchecked block optimizations

Severity: Gas Optimization

Context: AeraVaultV2.sol#L537-L538, AeraVaultAssetRegistry.sol#L114, AeraVaultAssetRegistry.sol#L252, AeraVaultAssetRegistry.sol#L264

Description: There are cases where arithmetic occurs whose underflow protection is already checked by a preceding logical precondition or overflow which is covered similarly and simply by being unattainable due to gas limits, such as the case of single increments in loops.

Recommendation: Rewrite these operations in unchecked blocks which will provide gas savings by removing these redundant or unreachable checks.

For AeraVaultV2.sol#L537-L538 we already have a precondition that makes sure we don't underflow

```
 function _getFeeIndex() internal view returns (uint256 feeIndex) {
        if (block.timestamp > lastFeeCheckpoint) {
+            unchecked {
                feeIndex = block.timestamp - lastFeeCheckpoint;
+           }
        }
...
```

For AeraVaultAssetRegistry.sol#L264

```
- for (uint256 i = 0; i < numAssets; i++) {
+ for (uint256 i = 0; i < numAssets;) {
    if (
        _assets[i].isERC4626
        && IERC4626(address(_assets[i].asset)).asset() == asset
    ) {
        revert Aera__AssetIsUnderlyingAssetOfERC4626(
            address(_assets[i].asset)
        );
    }
+   unchecked {
+       i++; // gas savings
+   }
}
```

AeraVaultAssetRegistry.sol#L114 and AeraVaultAssetRegistry.sol#L252 will follow a similar pattern as the last example, and the Aera team already uses a number of these unchecked increments where appropriate within loops so are familiarized with them.

Aera: Recommendation accepted. Fixed in PR 204.

Spearbit: Fixed.

5.4.4 Redundant `Pausable` modifiers

Severity: Gas Optimization

Context: AeraVaultV2.sol#L384, AeraVaultV2.sol#L391, AeraVaultV2.sol#L399, AeraVaultV2.sol#L407

Description: The `pause` and `resume` functions are guarded by the modifiers `whenNotPaused` and `whenPaused`, respectively, inherited from `Pausable`. They also internally call to `_pause` and `_unpause` which are guarded by these same modifiers. Hence, a redundant check occurs wasting gas.

Recommendation: Remove the `Pausable` modifiers guarding the external `pause` and `resume` variants, and instead just depend on the check done from the internal calls for their appropriate states. This can save about 200 gas on a successful call, it will however expend more gas on failed calls, which ideally shouldn't be submitted to the network anyways. Consider amending the interface with commentary on this behavior, that the functions expect to be in an appropriate paused/unpaused state and will revert otherwise and that these checks are handled by the internal OZ `Pausable` calls.

Example suggestions for AeraVaultV2.sol#L384, AeraVaultV2.sol#L391

```
+  /// @dev MUST be in paused state.
    /// @inheritdoc ICustody
    function resume()
        external
        override
        onlyOwner
-       whenPaused
...
+       // checks that contract is paused first
        _unpause();
    }
```

Aera: Recommendation accepted. Fixed in PR 205.

Spearbit: Fixed.

5.4.5 Redundant check that WETH is not the zero address

Severity: Gas Optimization

Context: AeraVaultV2.sol#L144-L146, AeraVaultV2Factory.sol#L44-L46

Description: The vault constructor verifies that the WETH address is not the zero address. The WETH address passed to the vault constructor comes from the factory, which stores the address as an `immutable` and verifies it at construction. Therefore, since vaults will only be deployed from the factory, the check within the vault is redundant.

Recommendation: If there is confidence that vaults will only be deployed from the factory, this check can be removed to save gas. However, the check may be kept if deployment approaches might change, or if there is a chance Aera may be forked by another team and used in unknown ways.

Aera: Recommendation accepted. (We will indeed continue deploying through the factory.) Fixed in PR 206.

Spearbit: Fixed.

5.4.6 Move check after `for` loop to save gas

Severity: Gas Optimization

Context: AeraVaultAssetRegistry.sol#L450

Description: The `!assetsToCheck[underlyingIndex].isERC4626` check in the `for` loop of `_checkUnderlyingAsset` in `AeraVaultAssetRegistry` can be moved to be after the loop to save some gas - by doing this you can also revert with a more appropriate custom error based.

```
for (; underlyingIndex < numAssets; underlyingIndex++) {
    if (
        !assetsToCheck[underlyingIndex].isERC4626
            && underlyingAsset
                == address(assetsToCheck[underlyingIndex].asset)
    ) {
        break;
    }
}
if (underlyingIndex == numAssets) {
    revert Aera__UnderlyingAssetIsNotRegistered(address(asset.asset), underlyingAsset);
}
```

Recommendation: Move the check after the `for` loop and use a concrete custom error for it, like the following code:

```
 for (; underlyingIndex < numAssets; underlyingIndex++) {
    if (
-        !assetsToCheck[underlyingIndex].isERC4626 &&
         underlyingAsset == address(assetsToCheck[underlyingIndex].asset)
    ) {
        break;
    }
}
if (underlyingIndex == numAssets) {
    revert Aera__UnderlyingAssetIsNotRegistered(address(asset.asset), underlyingAsset);
}
+if ( assetsToCheck[underlyingIndex].isERC4626)
+    revert Aera_UnderlyingAssetIsItselfERC4626();
```

Aera: Recommendation accepted. Fixed in PR 207.

Spearbit: Fixed.

5.4.7 Function `afterSubmit()` does storage writes that are not always necessary

Severity: Gas Optimization

Context: AeraVaultHooks.sol#L215-L283

Description: The function `afterSubmit()` does some storage writes that are not always necessary.

```
function afterSubmit(Operation[] calldata operations) ... {
    ...
    if (_beforeValue > 0) {
        ...
        cumulativeDailyMultiplier = newMultiplier;    // update might not be necessary
    }
    currentDay = day; // update might not be necessary
    ...
}
```

Recommendation: Consider changing the code so something like this:

```
uint256 newMultiplier;
uint256 currentMultiplier = cumulativeDailyMultiplier;

if (_beforeValue > 0) {
    // Initialize new cumulative multiplier with the current submit multiplier.
    newMultiplier = (custody.value() * ONE) / _beforeValue;
} else
    newMultiplier = ONE;

if (currentDay == day) {
     // Calculate total multiplier for today.
    newMultiplier = (currentMultiplier * newMultiplier) / ONE;
} else // Effects: reset day and prior vault value for the next submission.
    currentDay = day; // only update day when necessary

// Requirements: check that daily execution loss is within bounds.
if (newMultiplier < ONE - maxDailyExecutionLoss) {
    revert Aera__ExceedsMaxDailyExecutionLoss();
}
// Effects: update the daily multiplier.
if (currentMultiplier != newMultiplier)
    cumulativeDailyMultiplier = newMultiplier;  // only update cumulativeDailyMultiplier when necessary
```

Aera: Fixed in PR 216.

Spearbit: Verified

5.4.8 Expensive functions `_reserveFees()` and `value()` called both but are very similar

Severity: Gas Optimization

Context: AeraVaultV2.sol#L505-L511, AeraVaultV2.sol#L545-L583, AeraVaultV2.sol#L590-L622

Description: The functions `value()` and `_reserveFees()` are very similar, they both call `spotPrices()` and `_value()`, which costs a lot of gas. In the case of `submit()`, function `_reserveFees()` is called once and `value()` is called twice.

```
function value() external view override returns (uint256 vaultValue) {
    ... erc20SpotPrices =assetRegistry.spotPrices();
    IERC20 feeToken = assetRegistry.feeToken();
    (vaultValue,) = _value(erc20SpotPrices, feeToken);
}
function _reserveFees() internal {
    ...
    IERC20 feeToken = assetRegistry.feeToken();
    try assetRegistry.spotPrices() returns ( ... erc20SpotPrices ) {
        (lastValue, lastFeeTokenPrice) = _value(erc20SpotPrices, feeToken);
    } catch {}
    ...
}
```

Recommendation: Consider optimizing `submit()` by combining the `_reserveFees()` and the first call to `value()`, which is in the `beforeSubmit()` hook.

Also see issues:

- "`holdings()` and `value()` can be called while calls are being executed"
- "Storing `_beforeValue` and `_beforeBalance` is relative expensive"

Aera: We don't see an elegant way of doing this without changing the interface of the `beforeSubmit` hook. We acknowledge that there is duplication and wait until the transient TSTORE opcode is introduced and we can implement a caching based solution.

Spearbit: Acknowledged.

5.4.9 Storing `_beforeValue` and `_beforeBalance` is relative expensive

Severity: Gas Optimization

Context: AeraVaultHooks.sol#L43-L46, AeraVaultHooks.sol#L183-L243

Description: The hooks contracts stores values in `_beforeValue` and `_beforeBalance` in between the `beforeSubmit()` and `afterSubmit()`, which is relatively expensive. Although the idea is to use `TSTORE`/`TLOAD` in the future, these opcodes will not be available soon on all chains.

```
uint256 internal _beforeValue;
uint256 internal ;

function beforeSubmit(Operation[] calldata operations) ... {
    ...
    // Effects: remember current vault value and ETH balance for use in afterSubmit.
    _beforeValue = custody.value();
    _beforeBalance = address(custody).balance;
}
function afterSubmit(Operation[] calldata operations) ... {
    ... // user _beforeValue and _beforeBalance
}
```

Recommendation: Consider passing the values for `value()`, the native tokens balance and possibly `holdings()` to the before and after hooks. The `deposit`, `withdraw` and `finalize` hooks will have some extra overhead, but they are not called frequently. The `submit()` hooks could be optimized by combining `value()` and `_reserveFees()`.

Alternatively return the temporary values from the before hook and supply them in the calling function. This however leaks information about the hooks implementation, which is undesirable according to the project team.

Also see issues:

- "`holdings()` and `value()` can be called while calls are being executed"

- "Expensive functions `_reserveFees()` and `value()` called both but are very similar"

Aera: We do not want to leak implementation details from hooks at this time and the gas cost is acceptable with the current `submit` call frequency.

Spearbit: Acknowledged.

5.4.10 `maxDailyExecutionLoss_` can be `immutable`

Severity: Gas Optimization

Context: AeraVaultHooks.sol#L215-L243, AeraVaultHooks.sol#L289-L294, AeraVaultHooks.sol#L77-L121

Description: The variable `maxDailyExecutionLoss` is never updated (except in `afterFinalize()`), so it can be `immutable`. Also an `immutable` variable can be made for `ONE - maxDailyExecutionLoss`, which makes `afterSubmit()` somewhat cheaper to execute. In that case `maxDailyExecutionLoss` isn't even necessary anymore.

```
uint256 public maxDailyExecutionLoss;
constructor(...) {
    ...
    // Requirements: check if max daily execution loss is bounded.
    if (maxDailyExecutionLoss_ > ONE) {
        revert Aera__MaxDailyExecutionLossIsGreaterThanOne();
    }
    ...
    maxDailyExecutionLoss = maxDailyExecutionLoss_;
    ...
}
function afterSubmit(Operation[] calldata operations) ... {
    ...
    // Requirements: check that daily execution loss is within bounds.
    if (newMultiplier < ONE - maxDailyExecutionLoss) {
        revert Aera__ExceedsMaxDailyExecutionLoss();
    }
    ...
}
function afterFinalize() external override onlyCustody {
    ...
    maxDailyExecutionLoss = 0;
    ...
}
```

Recommendation: Consider changing the code to:

```
- uint256 public maxDailyExecutionLoss;
+ uint256 public immutable minDailyValue;
constructor(...) {
    ...
    // Requirements: check if max daily execution loss is bounded.
    if (maxDailyExecutionLoss_ > ONE) {
        revert Aera__MaxDailyExecutionLossIsGreaterThanOne();
    }
    ...
-   maxDailyExecutionLoss = maxDailyExecutionLoss_;
+   minDailyValue = ONE - maxDailyExecutionLoss_;
    ...
}
function afterSubmit(Operation[] calldata operations) ... {
    ...
    // Requirements: check that daily execution loss is within bounds.
-   if (newMultiplier < ONE - maxDailyExecutionLoss) {
+   if (newMultiplier < minDailyValue ) {
        revert Aera__ExceedsMaxDailyExecutionLoss();
    }
    ...
}
function afterFinalize() external override onlyCustody {
    ...
-    maxDailyExecutionLoss = 0;
    ...
}
```

Aera: Fixed in PR 217.

Spearbit: Verified.

5.4.11 Changing user-defined type `TargetSighash` to be based off `bytes32` is more suited and saves gas

Severity: Gas Optimization

Context: Types.sol#L10-L13, TargetSighashLib.sol#L4-L22

Description: The `TargetSighash` user-defined type is currently based off `uint256`, which is more appropriate for numerical values that may expect some accompanying arithmetic operations to be performed upon them. The intended purpose of `TargetSighash` is to store a fixed structure of 3 bytes sequences, which can be validated and converted from its unstructured components easily.

Recommendation: A fixed `bytes32` as the base type would seem more canon as it is sequences of bytes being dealt with, which provides a better native representation. In addition to this, some overhead pertained to working with `uint` types can be saved, and thereby gas savings achieved, with initial profiling indicating up to 40 gas saved per call with the following code candidate:

```
    function toTargetSighash(
        address target,
        bytes4 selector
    ) internal pure returns (TargetSighash targetSighash) {
-       // Upcast to uint256 is required to prevent truncation during left shift.
        targetSighash = TargetSighash.wrap(
-           (uint256(uint160(target)) << 32) | uint32(selector)
+           bytes20(target) | (bytes32(selector) >> (20 * 8))
...
```

Since fixed-size byte types are left aligned instead of right aligned like `uint`, an additional code and comment revision will be required under the Types.sol file to reflect this

```
 /// @notice Combination of contract address and sighash to be used in allowlist.
 /// @dev It's packed as follows:
- ///      [<empty> 64 bits] [target 160 bits] [selector 32 bits]
- type TargetSighash is uint256;
+ ///      [target 160 bits] [selector 32 bits] [<empty> 64 bits]
+ type TargetSighash is bytes32;
```

Aera: Fixed in PR 168.

Spearbit: Verified.

5.4.12 Saving a vault's `description` to storage can be expensive

Severity: Gas Optimization

Context: AeraVaultV2.sol#L43-L46

Description: It can be expensive writing the `description` to storage depending on the description length. It costs 20,000 gas per 32 bytes of description text, plus another 20,000 gas to store string length if more than 1 slot is needed. The description is not needed on-chain, so this storage write can be avoided.

Recommendation: Only emit the description in an event within the constructor of `AeraVaultV2` instead of saving it to storage. Alternatively, if vaults will only ever be deployed from the factory contract, consider just leveraging the event already emitted by the factory, and never passing the `description` to the vault constructor.

Aera: We will only emit the description event in the factory contract (since all vaults are only deployed by the factory). Fixed in PR 175.

Spearbit: Fixed.

5.4.13 Cache variables in `immutable` storage whenever possible

Severity: Gas Optimization

Context: AeraVaultAssetRegistry.sol#L350, AeraVaultV2.sol#L322-L323, AeraVaultV2.sol#L425-L426, AeraVaultV2.sol#L478, AeraVaultV2.sol#L508, AeraVaultV2.sol#L560, AeraVaultV2.sol#L738, AeraVaultV2.sol#L766

Description: The `numeraireDecimals` local variable in `AeraVaultAssetRegistry::spotPrices` can be just cached as an `immutable` value in the constructor since the numeraire asset can't change. Doing this will save gas for the callers of `spotPrices`.

Additionally, the `feeToken` is immutable. If all asset registries will have an immutable fee token, that can also be saved as an immutable in the `AeraVaultV2` constructor to remove all `assetRegistry.feeToken()` calls.

Recommendation: Cache the `numeraireDecimals` value in the constructor of `AeraVaultAssetRegistry`, and cache the `feeToken` value in the constructor of `AeraVaultV2` if all asset registries will have an immutable fee token.

Aera: Recommendation accepted. Fixed in PR 230.

Spearbit: Fixed.

5.4.14 Skip 0 transfers

Severity: Gas Optimization

Context: AeraVaultV2.sol#L342-L377

Description: The function `finalize()` might transfer `0` tokens if there are no tokens left for a specific asset. This costs gas and leaves an `event` on chain.

```
function finalize() ... {
    ...
    // Effects: transfer assets to owner.
    assetAmounts[i].asset.safeTransfer(owner(), assetAmounts[i].value); // value could be 0
    ...
}
```

Recommendation: Consider only transferring if `value > 0`.

```
 function finalize() ... {
    ...
    // Effects: transfer assets to owner.
+   if (assetAmounts[i].value > 0)
        assetAmounts[i].asset.safeTransfer(owner(), assetAmounts[i].value);
    ...
}
```

Aera: Recommendation accepted. Fixed in PR 221.

Spearbit: Fixed.

5.4.15 Use `unchecked` in for loop

Severity: Gas Optimization

Context: AeraVaultV2.sol#L227-L266

Description: On several locations in the code, the `i++` of a `for` loop is optimized by placing it inside the loop with an `unchecked` block. There are several other locations where this could be done. Additionally for unoptimized builds `++i` is slightly cheaper than `i++`.

```
function withdraw(AssetValue[] calldata amounts) ... {
    ...
    for (uint256 i = 0; i < numAmounts; i++) {
        assetValue = amounts[i];
        if (assetValue.value == 0) {
            continue;
        }
    }
    ...
}
```

Recommendation: Consider changing all `i++` statements in a `for` loop to an `unchecked` block. Be careful with `continue` statements because without the `i++` in the loop they might lead to an endless loop. Here is an example.

```
 function withdraw(AssetValue[] calldata amounts) ... {
     ...
-    for (uint256 i = 0; i < numAmounts; i++) {
+    for (uint256 i = 0; i < numAmounts; ) {
         assetValue = amounts[i];
-        if (assetValue.value == 0) {
-            continue;
-        }
+        if (assetValue.value > 0)
             assetValue.asset.safeTransfer(owner(), assetValue.value);
+        unchecked {
+            ++i; // gas savings
+        }
     }
     ...
}
```

Changing `i++` to `++i` shouldn't be necessary for optimized builds.

Aera: Recommendation accepted. Fixed in PR 223.

Spearbit: Fixed.

5.4.16 More efficient duplicate check in `deposit()` and `withdraw()`

Severity: Gas Optimization

Context: AeraVaultV2.sol#L169-L224, AeraVaultV2.sol#L227-L266, AeraVaultV2.sol#L627-L668, AeraVaultAssetRegistry.sol#L145-L153

Description: The functions `deposit()` and `withdraw()` (via `_checkWithdrawRequest()`) check that the input array `amounts` doesn't contain duplicates via a nested for loop. If the input array is sorted, then this can be checked with a single for loop, which is more effecient. This pattern is also used in the `constructor` of `AeraVaultAssetRegistry`.

```
function deposit(AssetValue[] calldata amounts) ... {
    ...
    for (uint256 i = 0; i < numAmounts;) {
        ...
        for (uint256 j = 0; j < numAmounts;) {
            // Requirements: check that no duplicate assets are provided.
            if (i != j && assetValue.asset == amounts[j].asset) {
                revert Aera__AssetIsDuplicated(assetValue.asset);
            }
            ...
        }
        ...
    }
    ...
}
function withdraw(AssetValue[] calldata amounts) ... {
    ...
    // Requirements: check the withdraw request.
    _checkWithdrawRequest(assets, amounts);
    ...
}
function _checkWithdrawRequest(IAssetRegistry.AssetInformation[] memory assets, ... ) ... {
    ...
    for (uint256 i = 0; i < numAmounts;) {
        ...
        for (uint256 j = 0; j < numAmounts;) {
            if (i != j && assetValue.asset == amounts[j].asset) {
                revert Aera__AssetIsDuplicated(assetValue.asset);
            }
            ...
        }
        ...
    }
    ...
}
```

Recommendation: let the functions `deposit()` and `withdraw()` have sorted input parameters and update the check for duplicates logic.

Aera: Fixed in PR224 .

Spearbit: Verified.

5.4.17 Use `EnumerableSet` and `EnumerableMap` for `_assets`

Severity: Gas Optimization

Context: AeraVaultAssetRegistry.sol#L25, IAssetRegistry.sol#L14-L18

Description: The `_assets` registry requires a relative large amount of code to keep the array up to date. It is also possible to use standard libraries to do this, which is less errorprone and easier to maintain.

```
contract AeraVaultAssetRegistry is IAssetRegistry, ERC165, Ownable2Step {
    AssetInformation[] internal _assets;
}
interface IAssetRegistry {
    struct AssetInformation {
        IERC20 asset;
        bool isERC4626;
        AggregatorV2V3Interface oracle;
    }
}
```

Additionally, `assets()` and `decimals()` are called frequently, while they normally won't change. Retrieving these values once and storing them would save gas.

Sometimes it is relevant to determine if an ERC20 is used as an underlying asset for an ERC4626. That can also be stored in the data structures. See for example issue "Spot prices does not need to compute prices for assets that have 0 amounts".

Recommendation: Consider using the Openzeppelin EnumerableSet and Openzeppelin EnumerableMap.

As these libraries work with mappings to a maximum of `bytes32`, the entire `struct` for `AssetInformation` won't directly fit. However, the content could be split in information about `ERC20` and `ER4626` tokens, because they are often handled separately:

- For ERC20: Use an `AddressToUintMap` where the ERC20 maps to `oracle`, stored in an `uint256`.
- For ERC4626: Use an `EnumerableSet`. The bool isERC4626 isn't necessary anymore then because there are two different datastructures.

To save gas this can be combined with storing `assets()` and `decimals()` in the following way:

- For ERC20: Use an `AddressToUintMap` where the ERC20 maps to `(oracle, decimals)` , packed in an `uint256`.
- For ERC4626: Use an `AddressToUintMap` where the ERC4626 maps to `underlyingAsset`, stored in an `uint256`.

To combine this with an indication if an ERC20 is used as an underlying asset for an ERC4626 a counter can be maintained, which indicated for how many ERC4626s the ERC20 is the underlying asset. This should be updated when adding or removing tokens. Note: this also makes some checks in removeAsset easier.

- For ERC20: Use an `AddressToUintMap` where the ERC20 maps to `(address oracle, uint 8 decimals, uint8 counter)` , packed in an `uint256`.

Aera: We will keep the current structure due to the additional complexity of packing. There is a better solution for handling assets that uses caching of oracle values so we will have to rewrite this part in the future anyways.

Spearbit: Acknowledged.

5.5 Informational

5.5.1 Atomic deployments can simplify instance setup and reduce risk

Severity: Informational

Context: AeraVaultV2Factory.sol#L52-L84

Description: The process for creating new Aera instances is a multi-step process: The vault is deployed from the factory, the hooks and registry contracts are separately deployed, then setter methods are required to initialize the contracts so they are aware of each other's addresses. This process results in added complexity and issues such as *"Beware of CREATE2 dangers with replaceable contracts possible via arbitrary factory"* and *"Lack of assetRegistry.custody check upon being set can lead to invalid vault values and permanent loss of funds"*. It also means scripts that create instances must be capable of recovering from any transaction in the flow reverting.

Recommendation: Update the factory so it can support atomic deployment and initialization of all three contracts (the vault, the initial hooks contract, and the registry) required for an Aera instance. There are a few ways this can be done, and some of them share aspects.

First, consider removing the arbitrary `deploy` function from the factory and leveraging a deterministic deployer such as `deterministic-deployment-proxy`:

- This deployer is on main chains, and for chains it's not yet on, it can be deployed by anyone as long as that chain (1) does not require EIP-155 transactions, and (2) has the same gas metering as the EVM. Celo and Arbitrum are notable chains that each violate one of requirements, but the deployer was administratively placed on both of those chains.
- This deployer is automatically used in forge scripts when deploying a salt via e.g. `new MyContract{salt: mySalt}(args)`.
- Leveraging this in deploy scripts instead of relying on the the `AeraVaultV2Factory` factory achieves the same goals, but mitigates the risk of the owner deploying malicious vaults through the vault factory. The owner can still deploy malicious vaults through this deployer instead, but that makes it easier to discover nonstandard vaults since they did not use the expected factory.

The vaults can deploy the hook and registry contracts during construction, this way an instance's deploy and setup is fully atomic and there's no risk of a step failing. Given that the hook and registry contracts may change, there are two main ways to do this:

1. Pass in `bytes memory hookCreationCode` and `bytes memory registryCreationCode` as vault constructor arguments, and deploy them in the vault constructor with `CREATE`. The constructor would also execute any required setup calls such as calling `setCustody`. However, since the hook and registry both know the vault address is `msg.sender`, the separate setter transaction is unnecessary (and consequently the `setCustody` method can be removed). Additionally, all addresses can be set (as immutables where necessary) in constructors without requiring precomputation of addresses. The usage of `CREATE` instead of `CREATE2` is important for this, as it removes the the creation code from address calculation.
2. For a given hook or registry contract, also have a corresponding factory contract that deploys that contract. The vault now takes each factory address and two encoded constructor arguments (e.g. `bytes memory encodedHookConstructorArgs`) as inputs. The vault constructor calls the factory and passes along the constructor arguments. The constructor arguments are encoded as bytes to support differing constructor argument formats in a single vault (in case future hook or registry contracts need different parameters) so the hook/registry would be responsible for decoding these in their constructor. This has the same address-setting benefits as the prior option.

With either of these approaches the `ERC165Checker.supportsInterface()` could potentially be skipped, however they may be worth keeping to verify the correct contracts are used. If going with the hooks/registry factory approach, the `AeraVaultV2Factory` owner can maintain an allowlist of factories that are known to be safe and conform to the required interfaces.

Aera: Fixed by PR 178, PR 233, PR 246, PR 255.

Spearbit: Verified

5.5.2 The number of days isn't always 365

Severity: Informational

Context: File.sol#L123, AeraVaultV2.sol#L27-L30.

Description: The number of days isn't always 365 as shown in the comments, but slightly more as is shown by leap years. For details see wiki Year. The difference is very minor though.

```
    /// @notice Largest possible fee earned proportion per one second.
    /// @dev 0.0000001% per second, i.e. 3.1536% per year.
    ///      0.0000001% * (365 * 24 * 60 * 60) = 3.1536%
    uint256 private constant _MAX_FEE = 10 ** 9;
```

Recommendation: Consider updating the comment.

Aera: Recommendation accepted. Fixed in PR 197.

Spearbit: Fixed.

5.5.3 Explicitly include inherited constructors for improved readability

Severity: Informational

Context: AeraVaultV2Factory.sol#L43, AeraVaultV2.sol#L117-L125, AeraVaultHooks.sol#L77-L82, AeraVaultAssetRegistry.sol#L95-L100

Description: The contracts intended for deployment utilize multi-level inheritance and among these is a dependency that has its own constructor. Namely, they tend to inherit `Ownable2Step` which in turn inherits `Ownable` and contains its own constructor that is fired off before that of the derived contracts. It initially sets the `Context._msgSender()` to owner until the derived contract's constructor overrides it with a `_transferOwnership(owner_)` call, in the current design.

Recommendation: Consider making the constructor effects of inherited contracts more obvious by explicitly including their invocation in the derived contract's constructor.

In the case of `AeraVaultAssetRegistry`

```
 constructor(
        address owner_,
        AssetInformation[] memory assets_,
        uint256 numeraireId_,
        IERC20 feeToken_
-   ) {
+   ) Ownable() {
```

which can similarly be applied to the other derived contracts noted under context. Note that the order here does not affect the invocation order of the constructors, the inheritance graph decides that.

Aera: Recommendation accepted. Fixed in PR 196.

Spearbit: Fixed.

5.5.4 Comments for `_checkUnderlyingAsset()` could be more detailed

Severity: Informational

Context: AeraVaultAssetRegistry.sol#L95-L226, AeraVaultAssetRegistry.sol#L439-L463

Description: The function `_checkUnderlyingAsset()` correctly doesn't show a match if the underlying asset is an ERC4626. This isn't expressed in the comments.

```
constructor(...) ... {
    ...
    if (assets_[i].isERC4626) {
        // Requirements: check that underlying asset exists.
        _checkUnderlyingAsset(assets_[i], assets_);
    }
    ...
}
function addAsset(AssetInformation calldata asset) ... {
    ...
    // Requirements: check that underlying asset is registered.
    if (asset.isERC4626) {
        _checkUnderlyingAsset(asset, _assets);
    }
    ...
}
/// @notice Check whether the underlying asset is listed or not.
...
function _checkUnderlyingAsset(...) ... {
     ...
     if (
          !assetsToCheck[underlyingIndex].isERC4626   // so if UnderlyingAsset is an ERC4626 it won't
          ↪ match
               && underlyingAsset == address(assetsToCheck[underlyingIndex].asset)
     ) {
          break;
     }
     ...
}
```

Recommendation: Expand the comments to indicate that underlying assets may not be an ERC4626.

Aera: Recommendation accepted. Fixed in PR 195.

Spearbit: Fixed.

5.5.5 `AeraVaultAssetRegistry` use different ways to track similar tokens

Severity: Informational

Context: AeraVaultAssetRegistry.sol#L95-L99

Description: The contract `AeraVaultAssetRegistry` uses two different ways to track the similar tokens `feeToken` and `numeraire`. `feeToken` is tracked by `address` and `numeraire` is tracked as an index of the `assets` array.

Using two different methods is more difficult to understand and maintain as well as more prone to errors.

```
contract AeraVaultAssetRegistry is IAssetRegistry, ERC165, Ownable2Step {
    IERC20 public immutable feeToken;
    uint256 public numeraireId;

    constructor(... , AssetInformation[] memory assets_, uint256 numeraireId_, IERC20 feeToken_) {
        ...
        numeraireId = numeraireId_;
        feeToken = feeToken_;
        ...
    }
}
```

Recommendation: Consider using the same approach for both tokens, preferably by tracking via the `address` because that is less error prone. Also see issues:

- "Use `EnumerableSet` and `EnumerableMap` for `_assets`"
- "Cache variables in `immutable` storage whenever possible"

Aera: Numeraire is tracked by address. Fixed in PR 226 and PR 231.

Spearbit: Verified.

5.5.6 Detect failing oracles

Severity: Informational

Context: AeraVaultV2.sol#L545-L583, AeraVaultV2.sol#L505-L511

Description: If one of the oracles always `reverts` than `spotPrices()` always fails and then `lastFeeTokenPrice` will always stay `0` and no fees will acccrue. In that situation `value()` will also `revert`, which means `submit()` will also `revert`.

```
uint256 public lastFeeTokenPrice;

function _reserveFees() internal {
    ...
    try assetRegistry.spotPrices() returns (
        IAssetRegistry.AssetPriceReading[] memory erc20SpotPrices
    ) {
        //  assume it never arrives here because spotPrices always fails
    } catch {}
    if (lastFeeTokenPrice == 0) {  // then lastFeeTokenPrice always stays 0
        return;
    }
    ...
}
function value() external view override returns (uint256 vaultValue) {
    IAssetRegistry.AssetPriceReading[] memory erc20SpotPrices = assetRegistry.spotPrices();
    ...
}
```

Recommendation: Add offline monitoring to detect the sitation where oracles fails, by checking failed `submit()` transactions. The guardian will also complain soon if its not detected.

Aera: We will incorporate this in our monitoring.

Spearbit: Acknowledged.

5.5.7 Avoid name collision by renaming a local variable

Severity: Informational

Context: AeraVaultV2.sol#L740

Description: In `AeraVaultV2::_getHoldings` you have the `asset` local variable, but it is of type `IAssetRegistry.AssetInformation` which is a struct that itself has an `asset` variable in it. Consider renaming the `asset` local variable to `assetInfo` to avoid name collisions.

Recommendation: Change the code in `AeraVaultV2::_getHoldings` in the following way:

```
- IAssetRegistry.AssetInformation memory asset;
+ IAssetRegistry.AssetInformation memory assetInfo;
```

Aera: Recommendation accepted. Fixed in PR 192.

Spearbit: Fixed.

5.5.8 Vault should revert when there is 0 `availableFee` to claim

Severity: Informational

Context: AeraVaultV2.sol#L480-L490

Description: This code snippet is part of `AeraVaultV2::claim`:

```
uint256 availableFee =
    Math.min(feeToken.balanceOf(address(this)), reservedFee);
uint256 unavailableFee = reservedFee - availableFee;
feeTotal -= availableFee;
reservedFee -= availableFee;

// Effects: update leftover fee.
fees[msg.sender] = reservedFee;

// Interactions: transfer fee to caller.
feeToken.safeTransfer(msg.sender, availableFee);
```

The problem with this is if `availableFee == 0` then the code will still execute by doing a 0 value `ERC20::transfer` and also emitting a `Claimed` event. This shouldn't be the case, especially since you have a `Aera__NoAvailableFeeForCaller` custom error.

Recommendation: Check if `availableFee == 0` and revert with `Aera__NoAvailableFeeForCaller` in `AeraVaultV2::claim`.

Aera: Recommendation accepted. Fixed in PR 194.

Spearbit: Fixed.

5.5.9 Custom error name is misguiding

Severity: Informational

Context: AeraVaultV2.sol#L475

Description: This code snippet is from `AeraVaultV2::claim`:

```
uint256 reservedFee = fees[msg.sender];

// Requirements: check that there are fees to claim.
if (reservedFee == 0) {
    revert Aera__NoAvailableFeeForCaller(msg.sender);
}
```

The issue is that the custom error name `Aera__NoAvailableFeeForCaller` is misguiding - it is actually thrown when the caller has no fees to claim, not when there is no available fee balance (the intent of the error).

Recommendation: Use a new custom error for this check, for example `Aera__NoClaimableFeesForCaller(address caller)`.

Aera: Recommendation accepted. Fixed in PR 167.

Spearbit: Fixed.

5.5.10 Set index local variable to 0 for explicitness and readability

Severity: Informational

Context: AeraVaultAssetRegistry.sol#L446, AeraVaultAssetRegistry.sol#L356

Description: The `underlyingIndex` local variable in `AeraVaultAssetRegistry::_checkUnderlyingAsset` is used as a `for` loop index variable in the method. Same is the case with the `index` local variable in `AeraVaultAssetRegistry::spotPrices`. As a best practice, set them to 0 for explicitness and readability.

Recommendation: Do the following changes:

```
- uint256 underlyingIndex;
+ uint256 underlyingIndex = 0;
```

```
- uint256 index;
+ uint256 index = 0;
```

Aera: Recommendation accepted. Fixed in PR 191.

Spearbit: Fixed.

5.5.11 If statement in `_getHoldings()` can be made more readable.

Severity: Informational

Context: AeraVaultV2.sol#L731-L760

Description: An if statement in `_getHoldings()` can be made more readable.

```
function _getHoldings(IAssetRegistry.AssetInformation[] memory assets) ... {
    ...
    if (assetAmounts[i].value > feeTotal) {
        assetAmounts[i].value -= feeTotal;
    } else {
        assetAmounts[i].value = 0;
    }
    ...
}
```

Recommendation: Consider changing the code to:

```
 function _getHoldings(IAssetRegistry.AssetInformation[] memory assets) ... {
    ...
-   if (assetAmounts[i].value > feeTotal) {
-       assetAmounts[i].value -= feeTotal;
-   } else {
-       assetAmounts[i].value = 0;
-   }
+   assetAmounts[i].value -= Math.min(feeTotal, assetAmounts[i].value);
    ...
}
```

Aera: Recommendation accepted. Fixed in PR 190.

Spearbit: Fixed.

5.5.12 Prefer ranged bound checks over exact for safety

Severity: Informational

Context: AeraVaultAssetRegistry.sol#L121, AeraVaultAssetRegistry.sol#L194, AeraVaultAssetRegistry.sol#L458

Description: Throughout the codebase, logical preconditions exist that test for an exact value to indicate an invalid state is reached, when the entire bound of values if ever reached would be invalid.

Recommendation: Even though these preconditions do seem logically sound in their current form, their current correctness depends on a number of different moving parts working correctly in tandem. Therefore it would be safer and more accurate to check the entire bound, and indeed consider those states explicitly invalid. It could save the team and end-users in case of any unforeseen or introduced off-by-one errors or the like.

AeraVaultAssetRegistry.sol#L121

```
- if (feeTokenIndex == numAssets) ...
+ if (feeTokenInfex >= numAssets) ...
```

AeraVaultAssetRegistry.sol#L194

```
- if (numAssets == MAX_ASSETS) ...
+ if (numAssets >= MAX_ASSETS) ...
```

AeraVaultAssetRegistry.sol#L458

```
- if (underlyingIndex == numAssets) ...
+ if (underlyingIndex >= numAssets) ...
```

Aera: Recommendation accepted. Fixed in PR 189.

Spearbit: Fixed.

5.5.13 Consider safer declarations of variables within loop

Severity: Informational

Context: AeraVaultAssetRegistry.sol#L353-L355

Description: 3 variables:

- `oracleDecimals`
- `price`
- `answer`

are used within a loop and declared above its scope. They are not intended to have any carry over between loop runs or outside of it.

Recommendation: It may be safer to have them declared each time within the loop to ensure no unexpected leaks between runs if the codebase is expected to see potential revisions in those portions of the code. This would come at a greater gas cost with a ceiling of ~350 gas for one full run through `spotPrices` when compiled optimized and assuming the worst case of 50 assets.

If the team is confident that this portion of the codebase is stable, they may choose to keep the code as is for efficiency, but to inspect future revisions for possible leaks.

Aera: Acknowledged. We expect this code path to be relatively stable.

Spearbit: Acknowledged.

5.5.14 Typos in code comments

Severity: Informational

Context: AeraVaultV2.sol#L35, AeraVaultV2.sol#L244

Description: One comment says `adress` instead of `address`, and another says `transfering` instead of `transferring`.

Recommendation: Correct the typos.

Aera: Solved in PR 166.

Spearbit: Verified.

5.5.15 Unnecessary initialization of `lastFeeCheckpoint`

Severity: Informational

Context: AeraVaultV2.sol#L73

Description: The initialization of `lastFeeCheckpoint = type(uint256).max` is unnecessary as `lastFeeCheckpoint` is set in the constructor with `lastFeeCheckpoint = block.timestamp;`

```
contract AeraVaultV2 is .. {
    ...
    uint256 public lastFeeCheckpoint = type(uint256).max;
    ...
    constructor(...) ... {
        ...
        lastFeeCheckpoint = block.timestamp;
        ...
    }
}
```

Recommendation: Remove the assignment at the variable declaration.

```
contract AeraVaultV2 is .. {
- uint256 public lastFeeCheckpoint = type(uint256).max;
+ uint256 public lastFeeCheckpoint;
```

Aera: Recommendation accepted. Fixed in PR 188.

Spearbit: Fixed.

5.5.16 Missing validation in `addTargetSighash` and `removeTargetSighash`

Severity: Informational

Context: AeraVaultHooks.sol#L126-L136, AeraVaultHooks.sol#L141-L152

Description: The `addTargetSighash` does not check that the target is a contract, which means EOAs may be added as targets. Additionally, the `addTargetSighash` method does not check that the given target/selector is currently `false`, and the `removeTargetSighash` does not check that the given target/selector is not already set to `true`.

Recommendation: If only contracts are intended to be added, add a check to `addTargetSighash` to verify the target has code. If redundant event emissions are not desirable, add a check that the target/selector is in the expected state before continuing with insertion or removal.

Aera: Recommendation accepted. Fixed in PR 236.

Spearbit: Fixed.

5.5.17 Inconsistent interfaces for specifying target addresses and selectors

Severity: Informational

Context: AeraVaultHooks.sol#L39, AeraVaultHooks.sol#L77-L82, AeraVaultHooks.sol#L126-L129, AeraVaultHooks.sol#L141-L144

Description: The constructor requires target addresses and selectors to be pre-formatted into the `TargetSighash` type, but `addTargetSighash` and `removeTargetSighash` take the target and selector as explicit arguments, leading to inconsistent interfaces for callers.

This difference also shows up in the `mapping(TargetSighash => bool) public targetSighashAllowed` variable, which matches the constructor's interface.

Recommendation: Use an array of structs in the constructor instead to match the interfaces of the `addTargetSighash` and `removeTargetSighash` methods, such as:

```
struct TargetSighashData {
  address target;
  bytes4 selector;
}

constructor(
    address owner_,
    address custody_,
    uint256 maxDailyExecutionLoss_,
    TargetSighashData[] memory targetSighashAllowlist
) {
```

Similarly, make the `targetSighashAllowed` mapping `internal` and add a getter method that exposes its data by taking explicit target and selector inputs.

This way the user that sets allowed targets and selectors never has to worry about the internal packed representation.

Aera: Fixed in PR 174.

Spearbit: Verified.

5.5.18 Local variable names shadow state variable names

Severity: Informational

Context: AeraVaultV2Factory.sol#L52-L72, AeraVaultV2Factory.sol#L87-L107

Description: In the `create` and `computeVaultAddress` methods of `AeraVaultV2Factory`, the local `owner` function argument shadows the `owner()` method from the `Ownable` contract.

Recommendation: In each method, rename the function argument to `vaultOwner` to disambiguate from the factory owner, or append an underscore such as `owner_`.

Aera: Fixed in PR 151 and PR 172.

Spearbit: Verified.

5.5.19 Beware of ERC4626 inflation attack

Severity: Informational

Context: Aera

Description: ERC4626 contracts have a know issue with inflation attacks, which occurs when the contract is still empty. For further information see:

- a-novel-defense-against-erc4626-inflation-attacks
- OpenZeppeling ERC4626.sol

Also see issue "Malicious Tokens, Oracles and erc4626 could undo `ExecutionLoss` checks".

Recommendation: When selecting ERC4626 contracts, be careful not to use empty ones because of risk of inflation attack.

Aera: Will add empty contract check to our ERC4626 whitelist criteria.

Spearbit: Acknowledged.

5.5.20 Result of function `value()` not 100% accurate

Severity: Informational

Context: AeraVaultV2.sol#L590-L622

Description: The function `value()`, which uses `_value()` uses `convertToAssets()` to retrieve the amount of underlying tokens in erc4626 contracts. According to the specification of eip-4626.

> The methods `totalAssets`, `convertToShares` and `convertToAssets` are estimates useful for display purposes, and do *not* have to confer the *exact* amount of underlying assets their context suggests.

Thus the result of function `value()` is not 100% accurate. The differences are probably neglectable, but it's good to be aware of this.

```
function _value(...) ... {
    ...
    balance = IERC4626(address(assets[i].asset)).convertToAssets(assetAmounts[i].value);
    ...
 }
```

Recommendation: Be aware of the fact that the result of function `value()` is not 100% accurate.

Aera: Acknowledged.

Spearbit: Acknowledged.

5.5.21 Emit events in all state-changing methods

Severity: Informational

Context: AeraVaultV2Factory.sol#L113, AeraVaultHooks.sol#L106-L111

Description: The `deploy` method in `AeraVaultV2Factory` deploys a contract, but does not emit an event for this. Consider emitting one so that off-chain monitoring can be done for the method. Another place where emitting an event would be good is when adding target sighashes in the constructor of `AeraVaultHooks`.

Recommendation: Add a `ContractDeployed` event in the `deploy` method of `AeraVaultV2Factory`. Emit a `TargetSighashAdded` event in the constructor of `AeraVaultHooks` when setting a target sighash as allowed.

Aera: Recommendation accepted. Fixed in PR 176.

Spearbit: Fixed.

5.5.22 Emit all sensible state-changing data in events

Severity: Informational

Context: AeraVaultV2Factory.sol#L75-L83, AeraVaultV2.sol#L545-L583, AeraVaultV2.sol#L440, AeraVaultV2.sol#L463, AeraVaultV2.sol#L223, AeraVaultV2.sol#L265

Description: The `VaultCreated` event in `AeraVaultV2Factory::create` emits all constructor arguments of `AreaVaultV2` apart from the `owner` value, which should also be emitted.

The `_reserveFees` method in `AeraVaultV2` performs 5 storage writes which are not emitted in an event: `feeTotal`, `fees[feeRecipient]`, `lastFeeCheckpoint`, `lastValue`, `lastFeeTokenPrice`.

Unlike the `Aera__SubmissionFailed` error, the `Aera__SubmitTargetIsHooksAddress` error does not include the operation's index. This makes it harder to identify the invalid operation when a `submit` call fails.

The `Submitted` event emits the owner address, but that method is called by the guardian. If guardian's change, this can make it difficult to identify the guardian that submitted the transaction.

For the `Deposit` and `Withdraw` events, all amounts are batched into a single event emission, which makes it harder to filter deposit and withdraw events for a given token.

Lastly, the `Claimed` event does not include the new `feeTotal`.

A best practice for event emissions is to ensure that current state can be reconstructed from all events.

Recommendation: Consider emitting the `owner` value as part of the `VaultCreated` event in `AeraVaultV2Factory::create`. Additionally, add a new event to log the storage changes from `_reserveFees()`.

Include the operation's index in the `Aera__SubmitTargetIsHooksAddress` signature.

Additionally, replace the owner's address with the guardian's address when emitting the `Submitted` event. Alternatively, consider emitting both the guardian and owner address if relevant.

```
emit Submitted(guardian, operations);
```

Change the `Deposit` and `Withdraw` events to be emitted within each loop iteration, where each event contains data for a single asset.

```
event Deposit(address indexed owner, IERC20 indexed asset, uint256 value);
event Withdraw(address indexed owner, IERC20 indexed asset, uint256 value);
```

Lastly, include the updated `feeTotal` in the `Claimed` event.

```
event Claimed(
    address indexed feeRecipient, uint256 feeTotal, uint256 claimedFee, uint256 unclaimedFee
);
```

Aera: Recommendation accepted. We will emit it as an indexed value. Fixed in PR 179.

Spearbit: Fixed.

5.5.23 Inherent limits of `submit()`

Severity: Informational

Context: AeraVaultV2.sol#L411-L454, AeraVaultHooks.sol#L215-L283, AeraVaultHooks.sol#L183-L212

Description: The expression space for `submit()` is limited:

- the result of one operation (e.g. amount of received tokens) can't be use for the next action;
- the number of received tokens can't be verified to limit slippage or to check the received erc4626 tokens (unless this is built in the protocol that is called, or its implemented in a surrounding check in a guardian contract). Also see eip-4626.md slippage limits.

Additionally some of the more advanced protocols can't be interacted with, due to checks on `allowances` in `afterSubmit()`. For example:

- Permit2
- UniswapX
- Cowswap TWAP

Also there is no support for eip-1271

- Cowswap

```
function submit(Operation[] calldata operations)
    ...
    for (uint256 i = 0; i < numOperations;) {
        ...
        (success, result) = operation.target.call{value: operation.value}(operation.data);
        ...
    }
}
```

`submit()` also can't send native tokens (ETH) directly to the `receive()` function of a smart contract because the allow list requires the calldata to be at least 4 bytes. Related to this is the additional limitation that there is no way to call a `fallback()` function for the same reason. This could be relevant for other EVM programming languages that don't use selectors. The owner would have to know the expected calldata and allow the first 4 bytes as a selector.

```
function beforeSubmit(Operation[] calldata operations) ... {
    ...
    for (uint256 i = 0; i < numOperations;) {
        selector = bytes4(operations[i].data[0:4]); // requires data to be at least 4 bytes
}
```

Recommendation: Doublecheck the inherent limits of `submit()`. Several solutions might help to reduce MEV extraction and frontruning. Perhaps its useful to be able to check the received tokens. Document the limitations of `submit()` in the documenation for the guardian.

Aera: Yes, aware. Current protocol functionality is sufficient for guardians. We will likely expand it in the future.

Spearbit: Acknowledged.

5.5.24 Index event parameters to simplify off-chain log queries

Severity: Informational

Context: AeraVaultV2Factory.sol#L25

Description: The `VaultCreated` event in `AeraVaultV2Factory` can index up to two more parameters, which might be worth it for easier off-chain querying of state changes.

Recommendation: Index two more parameters of the `VaultCreated` event.

Aera: Fixed in PR 165.

Spearbit: Verified.

5.5.25 On different chains `weth` is not the wrapped native token name

Severity: Informational

Context: AeraVaultV2Factory.sol#L13, AeraVaultV2.sol#L38-L39

Description: The contract `AeraVaultV2Factory` and `AeraVaultV2` have the `weth` storage variable, which should hold the address for the wrapped native token on the Ethereum blockchain. The issue is that the protocol will be deployed on multiple chains, and in most of which the native asset is not ETH, which makes the name `weth` incorrect.

```
contract AeraVaultV2Factory is IAeraVaultV2Factory, Ownable2Step {
    /// @notice The address of WETH.
    address public immutable weth;
    ...
}
contract AeraVaultV2 is
    /// @notice The address of WETH.
    address public immutable weth;
    ...
}
```

Recommendation: Consider renaming `weth` to `wrappedNativeToken` in `AeraVaultV2Factory` and `AeraVaultV2`.

Aera: Recommendation accepted. Fixed in PR 171.

Spearbit: Fixed.

5.5.26 Unused code can be removed and interfaces can be simplified

Severity: Informational

Context: IAeraVaultV2Factory.sol#L4-L6, IAssetRegistry.sol#L41-L56, IHooks.sol#L9, AeraVaultAssetRegistry.sol#L11, AeraVaultAssetRegistry.sol#L78

Description: Some smart contracts in the codebase contain imports of other contracts that are not used and can be deleted. The full list is:

- `import "../AeraVaultHooks.sol";` in `IAeraVaultV2Factory.sol`
- `import {TargetSighash} from "../Types.sol";` in `IAeraVaultV2Factory.sol`
- `import {ONE} from "./Constants.sol";` in `AeraVaultAssetRegistry.sol`

The import of `AeraVaultV2` in `IAeraVaultV2Factory` is not needed in that interface, but is needed in `AeraVaultV2Factory.sol`.

There is also an unused error: `Aera__ValueLengthIsNotSame` is defined in `AeraVaultAssetRegistry` but never used.

Additionally, interfaces for hook and registry contracts can simplified. Currently, the `IHooks` and `IAssetRegistry` contain many methods and events that are required by the current implementors of those interfaces, but may not be required for future hook or registry contracts. Specifically:

- `IHooks` inherits from `IHooksEvents`, but future hook contracts may use different events, and these events are not required to be part of the hook interface.
- `IAssetRegistry` contains methods that are never used by the vault or hooks contracts, and therefore do not need to part of the interface for future registry implementations. The full list of functions not required to be part of the `IAssetRegistry` interface are: `numeraireId()`, `addAsset()`, and `removeAsset()`.

Recommendation: Remove the unused imports and errors, and move the `AeraVaultV2` import from `IAeraVaultV2Factory` to `AeraVaultV2Factory`.

Additionally, the `AeraVaultHooks` contract can directly inherit `IHooksEvents` instead of `IHooks` inheriting from it. Consider renaming `IHooksEvents` to make it clear that this interface is for a specific hook implementation, not all hook implementations.

Remove the `numeraireId()`, `addAsset()`, and `removeAsset()` methods from the `IAssetRegistry` interface.

Aera: Recommendation accepted. Fixed in PR 214.

Spearbit: Fixed.

5.5.27 Comment in `finalize()` could be more detailed

Severity: Informational

Context: AeraVaultV2.sol#L365

Description: The comment in `finalize()` about "transfer assets to owner" could be more detailed.

```
function finalize() ... {
    ...
    // Effects: transfer assets to owner.
    assetAmounts[i].asset.safeTransfer(owner(), assetAmounts[i].value);
    ...
}
```

Recommendation: Consider changing the comment to something like:

```
- // Effects: transfer assets to owner.
+ // Effects: transfer registered assets to owner. Excluding reserved feetokens and native token (ETH).
```

Aera: Fixed in PR 164.

Spearbit: Verified.

5.5.28 Ordering of `_checkReservedFees()`

Severity: Informational

Context: AeraVaultV2.sol#L411-L464

Description: It seems slightly more logical to do the `prevFeeTokenBalance / _checkReservedFees()` in function `submit()` around the `hooks`. This might be relevant if future versions of hooks could somehow influence fees (this is not possible in the current code).

```
function submit(Operation[] calldata operations) ... {
    _reserveFees();

    // Hooks: before executing operations.
    hooks.beforeSubmit(operations);
    uint256 prevFeeTokenBalance = assetRegistry.feeToken().balanceOf(address(this));

    ... // submit actions

    // Invariants: check that insolvency of fee token was not introduced or increased.
    _checkReservedFees(prevFeeTokenBalance);
    // Hooks: after executing operations.
    hooks.afterSubmit(operations);
    ...
 }
```

Recommendation: Consider making a `modifier` for these checks. As `_reserveFees()` needs to be done before checking the current balance, this should be combined with making a modifier for `_reserveFees()`. See issue: "Ordering of `_reserveFees()`". Such a `modifier` could also be used at other functions as an extra safety precaution

Alternatively consider changing the code as shown below.

```
function submit(Operation[] calldata operations) ... {
    _reserveFees();
+   uint256 prevFeeTokenBalance = assetRegistry.feeToken().balanceOf(address(this));
    hooks.beforeSubmit(operations);
-   uint256 prevFeeTokenBalance = assetRegistry.feeToken().balanceOf(address(this));

    ... // submit actions

-   _checkReservedFees(prevFeeTokenBalance);
    hooks.afterSubmit(operations);
+   _checkReservedFees(prevFeeTokenBalance);
    ...
 }
```

Aera: We will add a `checkReservedFees` modifier. Fixed in PR 143.

Spearbit: Fixed. If going modifier route, ensure it's a post-run modifier, that runs following the body, just to be clear.

5.5.29 Comments for `_transferOwnership` not accurate

Severity: Informational

Context: AeraVaultV2.sol#L157-L158, AeraVaultAssetRegistry.sol#L178-L179, AeraVaultHooks.sol#L119-L120, Ownable2Step.sol#L44-L47, Ownable.sol#L78-L82

Description: The comment at the `contructors` of `AeraVaultV2`, `AeraVaultAssetRegistry` and `AeraVaultHooks` seem to suggest the `_transferOwnership()` results in a two step process. However `_transferOwnership()` is an internal function that directly transfers the ownership.

```
contract AeraVaultV2 is ... {
    constructor(...) {
        // Effects: initiate ownership transfer and pause vault.
        _transferOwnership(owner_);
    }
}
contract AeraVaultAssetRegistry is IAssetRegistry, ERC165, Ownable2Step {
    constructor(...) {
        // Effects: initiate ownership transfer to initial owner.
        _transferOwnership(owner_);
    }
}
contract AeraVaultHooks is IHooks, ERC165, Ownable2Step {
    constructor(...) {
        // Effects: create a pending ownership transfer.
        _transferOwnership(owner_);
    }
}
```

```
abstract contract Ownable2Step is Ownable {
    function _transferOwnership(address newOwner) internal virtual override {
        delete _pendingOwner;
        super._transferOwnership(newOwner);
    }
}
abstract contract Ownable is Context {
    function _transferOwnership(address newOwner) internal virtual {
        address oldOwner = _owner;
        _owner = newOwner;
        emit OwnershipTransferred(oldOwner, newOwner);
    }
}
```

Recommendation: Update the comments to show that the `owner` is changed immediately, for example in the following way:

```
- // Effects: create a pending ownership transfer.
+ // Effects: set new owner
```

Aera: Fixed in PR 163.

Spearbit: Verfied.

5.5.30 Separation of roles

Severity: Informational

Context: AeraVaultV2.sol#L777-L798

Description: Within the Aera protocol there are several roles defined:

1) owners, for AeraVaultV2Factory, AeraVaultV2, AeraVaultHooks and AeraVaultAssetRegistry
2) guardian
3) feeRecipient (and previous feeRecipient)
4) custody for AeraVaultHooks and AeraVaultAssetRegistry, which is the AeraVaultV2 contract

Some checks are done to make sure these addresses are not the same:

```
function _checkGuardianAddress(address newGuardian) internal view {
    ...
    if (newGuardian == owner()) {
        revert Aera__GuardianIsOwner();
    }
}
 function _checkFeeRecipientAddress(address newFeeRecipient) internal view {
    if (newFeeRecipient == owner()) {
        revert Aera__FeeRecipientIsOwner();
    }
 }
```

However there are lot of combinations that are not checked. Several of them are unwanted, for example:

- `guardian != feeRecipient` . According to `How-AeraV2-Works.pdf: it's important that there is no possible collusion with the guardian/fee recipient.`
- `owner of AeraVaultHooks != guardian`. If they are equal, the `guardian` could do `addTargetSighash()`
- `AeraVaultV2 != owner of any of the contracts`. If that would be the case, the `guardian` could perhaps use `submit()` to do owner actions, depending on the allow list configuration.

With `transferOwnership()` the different `owners` can be changed. There is currently no check to verify they are separate from other addresses.

Recommendation: Consider the relevance of checking addresses because one entity could easily use two addresses. If you do want to enforce: Determine which combinations of addresses should have different addresses. Implement checks to enforce this, also at locations where one of these addresses changes. This includes:

- `transferOwnership()`
- `setHooks()`
- `setGuardianAndFeeRecipient()`

Aera: The following checks will be enforced:

`guardian` != `owner` (vault) `guardian` != `owner` (hooks) `guardian` != `owner` (asset registry) `feeRecipient` != `owner` (vault) `vault` != `owner` (hooks) `vault` != `owner` (asset registry)

In:

AeraVaultV2

- `constructor` check `guardian` != `owner` (vault)
- `constructor` check `guardian` != `owner` (asset registry)
- `constructor` check `feeRecipient` != `owner` (vault)
- `constructor` check `vault` != `owner` (asset registry)
- `transferOwnership` check `guardian` != `owner` (vault)
- `transferOwnership` check `feeRecipient` != `owner` (vault)
- `setGuardianAndFeeRecipient` check `guardian` != `owner` (hooks)
- `setGuardianAndFeeRecipient` check `guardian` != `owner` (asset registry)
- `setHooks` check `guardian` != `owner` (hooks)
- `setHooks` check `vault` != `owner` (hooks)

AeraVaultHooks

- `transferOwnership` check `guardian` != `owner` (hooks)

AeraVaultAssetRegistry

- `transferOwnership` check `guardian` != `owner` (asset registry)

Fixed in PR 235. As many checks as possible are implemented, including in `transferOwnership()`. Excluding situations where the vault would assume that hooks / asset registry are ownable.

Spearbit: Verified.

5.5.31 Use one term for `Vault`

Severity: Informational

Context: ICustody.sol#L12, ICustodyEvents.sol#L7, AeraVaultV2.sol#L18-L24

Description: The contract `AeraVaultV2` inherits from `ICustody`. `Vault` and `Custody` are used to refer to the same object, which not is intuitive. This makes the code more difficult to read.

```
contract AeraVaultV2 is ICustody, ... {
    ...
}
interface ICustody is ICustodyEvents {
    ...
}
```

Recommendation: Consider renaming all uses of `Custody` to `Vault` (or the other way around).

Aera: Fixed in PR 170 and PR 177.

Spearbit: Verified.