CANTINA

Aera v3

Competition

August 13, 2025

Contents

1 Introduction **2**
- 1.1 About Cantina . . . 2
- 1.2 Disclaimer . . . 2
- 1.3 Risk assessment . . . 2
  - 1.3.1 Severity Classification . . . 2

2 Security Review Summary **3**

3 Findings **4**
- 3.1 High Risk . . . 4
  - 3.1.1 Per-Vault Oracle Overrides Ignored in Slippage Hooks . . . 4
- 3.2 Medium Risk . . . 5
  - 3.2.1 Struct mismatch will result in swap failure . . . 5
  - 3.2.2 `UniswapV3DexHooks`: Incorrect Path Parsing for `exactOutput` Swaps Leads to Slippage/Accounting Bypass . . . 7
  - 3.2.3 `KyberSwapDexHooks` doesn't check `srcReceivers` . . . 9
- 3.3 Low Risk . . . 12
  - 3.3.1 Unbounded accrualLag overflow in paused branch . . . 12
  - 3.3.2 Incorrect Oracle Quote Calculation in Milkman Price Checker Due to Ignored Protocol Fee . . . 13
- 3.4 Informational . . . 15
  - 3.4.1 DoS When Depositing With Max Token Allowance in `SingleDepositorVault` . . . 15

1 Introduction

1.1 About Cantina

Cantina is a security services marketplace that connects top security researchers and solutions with clients. Learn more at cantina.xyz

1.2 Disclaimer

A competition provides a broad evaluation of the security posture of the code at a particular moment based on the information available at the time of the review. While competitions endeavor to identify and disclose all potential security issues, they cannot guarantee that every vulnerability will be detected or that the code will be entirely secure against all possible attacks. The assessment is conducted based on the specific commit and version of the code provided. Any subsequent modifications to the code may introduce new vulnerabilities, therefore, any changes made to the code would require an additional security review. Please be advised that competitions are not a replacement for continuous security measures such as penetration testing, vulnerability scanning, and regular code reviews.

1.3 Risk assessment

Severity	Description
High	*Must* fix as soon as possible (if already deployed) and can be triggered by any user without significant constraints, generating outsized returns to the exploiter. For example: loss of user funds (significant amount of funds being stolen or lost) or breaking core functionality (failure in fundamental protocol operations).
Medium	Global losses <10% or losses to only a subset of users, requiring significant constraints (capital, planning, other users...) to be exploited. For example: temporary disruption or denial of service (DoS), minor fund loss or exposure or breaking non-core functionality
Low	Losses will be annoying but easily recoverable, requiring unusual scenarios or admin actions to be exploited.
Gas Optimization	Suggestions around gas saving practices.
Informational	Suggestions around best practices or readability.

1.3.1 Severity Classification

The severity of security issues found during the security review is categorized based on the above matrix. High severity findings represent the most critical issues that must be addressed immediately, as they either have high impact and high likelihood of occurrence, or medium impact with high likelihood.

Medium severity findings represent issues that, while not immediately critical, still pose significant risks and should be addressed promptly. These typically involve scenarios with medium impact and medium likelihood, or high impact with low likelihood.

Low severity findings represent issues that, while not posing immediate threats, could potentially cause problems in specific scenarios. These typically involve medium impact with low likelihood, or low impact with medium likelihood.

Lastly, some findings might represent improvements that don't directly impact security but could enhance the codebase's quality, readability, or efficiency (Gas and Informational findings).

2 Security Review Summary

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

From Jun 18th to Jun 25th Cantina hosted a competition based on aera-v3. The participants identified a total of **7** issues in the following risk categories:

- High Risk: 1
- Medium Risk: 3
- Low Risk: 2
- Gas Optimizations: 0
- Informational: 1

The present report only outlines the **high** and **medium** risk issues.

3 Findings

3.1 High Risk

3.1.1 Per-Vault Oracle Overrides Ignored in Slippage Hooks

Submitted by 0xvangrim, also found by Ziusz, Lamsy, davidjohn and globalace

Severity: High Risk

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

Summary: Hooks that enforce slippage and daily loss limits do not respect per-vault oracle overrides, causing all risk checks to use the default oracle even when a vault has opted into a new or custom feed.

Finding Description: The protocol allows vaults to override the default oracle for specific asset pairs by calling `acceptPendingOracle`, which writes to `oracleOverrides[vault][base][quote]`. However, when slippage or daily loss checks are performed in hooks (e.g., `UniswapV3DexHooks`, `KyberSwapDexHooks`, `OdosV2DexHooks`, etc.), the code calls:

```
//src/periphery/hooks/slippage/BaseSlippageHooks.sol
IOracleRegistry oracleRegistry = _vaultStates[msg.sender].oracleRegistry;
return IOracle(oracleRegistry).getQuote(amount, token, _getNumeraire());
```

Inside the registry, the relevant function is:

```
//src/periphery/OracleRegistry.sol
function getQuote(uint256 baseAmount, address base, address quote) external view returns (uint256) {
    IOracle oracle = _getOracleForVault(msg.sender, base, quote);
    return oracle.getQuote(baseAmount, base, quote);
}
```

At this point, `msg.sender` is the hook contract, not the vault. So the registry looks up `oracleOverrides[hook][base][quote]`, which is almost always unset, and falls back to the default oracle. The override set for the vault is never used.

Example call flow:

- Vault calls a hook (e.g., `UniswapV3DexHooks.exactInputSingle`).
- Hook calls `_convertToNumeraire`.
- `_convertToNumeraire` calls `oracleRegistry.getQuote`.
- Inside `getQuote`, `msg.sender` is the hook, not the vault.

As a result, the per-vault override is ignored and all calculations use the default oracle, breaking the guarantee that vaults can opt into new oracles.

Impact Explanation: Vaults may be exposed to stale or incorrect price feeds for risk checks, even after opting into a new oracle. This can result in trades being blocked unnecessarily or, worse, slippage/daily loss limits being calculated with outdated data, potentially allowing excessive losses.

Likelihood Explanation: Any vault that sets an override will be affected, and the issue is present in all slippage/daily loss hooks. No special conditions are required.

Proof of Concept: Add this helper in to `MockBaseSlippageHooks.sol`.

```
//test/periphery/mocks/hooks/slippage/MockBaseSlippageHooks.sol
// Expose internal functions for testing
function handleBeforeExactInputSingle(
    address tokenIn,
    address tokenOut,
    address recipient,
    uint256 amountIn,
    uint256 amountOutMinimum
) external returns (address, address, address) {
    return _handleBeforeExactInputSingle(tokenIn, tokenOut, recipient, amountIn, amountOutMinimum);
}

function convertToNumeraire(uint256 amount, address token) external view returns (uint256) {
    return _convertToNumeraire(amount, token);
}
```

Add this proof of concept to: `test/periphery/unit/OracleRegistry.t.sol`.

```
import { IBaseSlippageHooks } from "src/periphery/interfaces/hooks/slippage/IBaseSlippageHooks.sol";
import { MockBaseSlippageHooks } from "test/periphery/mocks/hooks/slippage/MockBaseSlippageHooks.sol";

// ...

    function test_hook_slippage_check_with_oracle_override_real_path() public {
    // Setup
    MockBaseSlippageHooks hooks = new MockBaseSlippageHooks(quote1);
    address vault = users.alice;

    vm.mockCall(vault, bytes4(keccak256("owner()")), abi.encode(address(this)));
    hooks.setOracleRegistry(vault, address(oracleRegistry));
    hooks.setMaxDailyLoss(vault, 1000e18);
    hooks.setMaxSlippagePerTrade(vault, 500); // 5% max slippage

    // Vault accepts new oracle (2x price)
    IOracle newOracle = _scheduleNewOracleUpdate(base1, quote1, BASE1_DECIMALS);
    vm.prank(vault);
    oracleRegistry.acceptPendingOracle(base1, quote1, vault, newOracle);

    // Demonstrate price discrepancy
    vm.prank(vault);
    uint256 priceVaultSees = oracleRegistry.getQuote(DEFAULT_BASE_AMOUNT, base1, quote1);
    assertEq(priceVaultSees, 2, "Vault sees override: 1 base1 = 2 quote1");

    vm.prank(vault);
    uint256 priceHookSees = hooks.convertToNumeraire(DEFAULT_BASE_AMOUNT, base1);
    assertEq(priceHookSees, 1, "Hook sees default: 1 base1 = 1 quote1");

    // Execute swap: 1e18 base1 for 1.4e18 quote1
    uint256 amountIn = DEFAULT_BASE_AMOUNT;
    uint256 amountOutMinimum = 1.4e18;

    // This should fail (30% slippage > 5% limit) but passes
    vm.prank(vault);
    hooks.handleBeforeExactInputSingle(
        base1,
        quote1,
        vault,
        amountIn,
        amountOutMinimum
    );

    // Hook thinks: 1 quote in, 1.4e18 quote out = profit!
    // Vault reality: 2 quote in, 1.4e18 quote out = 30% loss!
    assertTrue(priceHookSees < priceVaultSees, "Hook undervalues input, missing real slippage!");
}
```

Recommendation: Replace all calls to `oracleRegistry.getQuote(...)` in hooks with `oracleRegistry.getQuoteForUser(..., vault)`, passing the vault address explicitly. This ensures per-vault overrides are respected.

Example fix:

```
// Before
IOracleRegistry oracleRegistry = _vaultStates[msg.sender].oracleRegistry;
return IOracle(oracleRegistry).getQuote(amount, token, _getNumeraire());

// After
IOracleRegistry oracleRegistry = _vaultStates[vault].oracleRegistry;
return oracleRegistry.getQuoteForUser(amount, token, _getNumeraire(), vault);
```

This change should be applied wherever slippage/daily loss logic converts token amounts to the numeraire.

3.2 Medium Risk

3.2.1 Struct mismatch will result in swap failure

Submitted by TheWeb3Mechanic

Severity: Medium Risk

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

Summary: `UniswapV3DexHooks::exactInputSingle` takes in the `ExactInputSingleParams` struct as an argument. The issue here is that imported struct used here has a deadline parameter which is absent in the router deployed on Base network.

```
struct ExactInputSingleParams {
    address tokenIn;
    address tokenOut;
    uint24 fee;
    address recipient;
    uint256 deadline; // <<<
    uint256 amountIn;
    uint256 amountOutMinimum;
    uint160 sqrtPriceLimitX96;
}
```

On other chains, uniswap has 2 routers deployed. `SwapRouter` and `SwapRouter02`. For example;

- Mainnet: SwapRouter has the deadline parameter.
- Mainnet: SwapRouter02 does not have the does not have the deadline parameter in it.

On base chain only `SwapRouter02` is deployed, hence the `ExactInputSingle` does not expect a `ExactInputSingleParams` struct with a deadline parameter in it (Link to all uniswap deployed contract on Base).

Finding Description:

```
function exactInputSingle(ISwapRouter.ExactInputSingleParams calldata params) external returns (bytes memory) {
    // Requirements, Effects: perform slippage and daily loss checks
    (address tokenIn, address tokenOut, address receiver) = _handleBeforeExactInputSingle(
        params.tokenIn, params.tokenOut, params.recipient, params.amountIn, params.amountOutMinimum
    );
    return abi.encode(tokenIn, tokenOut, receiver);
}
```

Impact Explanation: High - When guardians submit a swap transaction on `BaseVault::submit` using the correct payload on base chain(struct without a deadline), the call data will be passed to `UniswapV3DexHooks::exactInputSingle` which will revert due to abidecoding error.

Likelihood Explanation: High - Will always happen on base chain.

Proof of Concept:

```
// SPDX-License-Identifier: UNLICENSED
pragma solidity ^0.8.20;
import "forge-std/Test.sol";
import { ISwapRouter } from "src/dependencies/uniswap/v3/interfaces/ISwapRouter.sol";

contract MockUniswapV3DexHooks {
    function exactInputSingle(ISwapRouter.ExactInputSingleParams calldata params) external returns (bytes
    ↪  memory) {
        return abi.encode("");
    }
}

contract AaeraTes is Test{
    struct BaseExactInputSingleParams {
        address tokenIn;
        address tokenOut;
        uint24 fee;
        address recipient;
        uint256 amountIn;
        uint256 amountOutMinimum;
        uint160 sqrtPriceLimitX96;
    }

    MockUniswapV3DexHooks uniswaphook;

    function setUp() public {
        uniswaphook = new  MockUniswapV3DexHooks();
    }

    function testStructMismatch() public {

        BaseExactInputSingleParams memory baseParams = BaseExactInputSingleParams({
            tokenIn: address(0x1),
            tokenOut: address(0x2),
            fee: 3000,
            recipient: address(this),
            amountIn: 1e18,
            amountOutMinimum: 0,
            sqrtPriceLimitX96: 0
        });
        bytes memory data = abi.encodeWithSelector(
            MockUniswapV3DexHooks.exactInputSingle.selector,
            baseParams // incorrect layout
        );
        // Perform low-level call (should fail or return garbage)
        (bool success, bytes memory returnData) = address(uniswaphook).call(data);

        assertFalse(success, "Call should fail due to struct mismatch");
        emit log_bytes(returnData);
    }
}
```

Recommendation: Use `SwapRouter02` for both deployment chain, and import the struct without the deadline parameter. Almost all chains have `SwapRouter02`.

- Ethereum: https://docs.uniswap.org/contracts/v3/reference/deployments/ethereum-deployments.
- Arbitrum: https://docs.uniswap.org/contracts/v3/reference/deployments/arbitrum-deployments.
- Base: https://docs.uniswap.org/contracts/v3/reference/deployments/base-deployments.

3.2.2 `UniswapV3DexHooks`: Incorrect Path Parsing for `exactOutput` Swaps Leads to Slippage/Accounting Bypass

Submitted by 0xvangrim, also found by davidjohn

Severity: Medium Risk

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

Summary: The contract parses the Uniswap V3 swap path incorrectly for `exactOutput` swaps, reversing the roles of `tokenIn` and `tokenOut`. This causes all slippage and accounting checks to be performed on the wrong asset, breaking core security guarantees.

Finding Description: In `UniswapV3DexHooks`, both `exactInput` and `exactOutput` swaps extract `tokenIn` and `tokenOut` from the path using the same logic:

```
tokenIn = address(bytes20(params.path[:20]));
tokenOut = address(bytes20(params.path[pathLength - 20:]));
```

However, for `exactOutput` swaps, Uniswap V3 encodes the path in reverse order (see Uniswap Docs: Multihop Swaps). This means the first 20 bytes are actually the output token, and the last 20 bytes are the input token. The current implementation causes all downstream logic (slippage checks, loss accounting, oracle lookups) to operate on the wrong tokens. As a result, slippage protection and loss limits can be bypassed or may revert unexpectedly, allowing trades that should be blocked or blocking trades that should be allowed.

Impact Explanation: High: This bug breaks core protocol security by disabling or misapplying slippage and loss controls, potentially leading to unbounded loss of user funds or protocol insolvency.

Likelihood Explanation: High: The bug will be triggered whenever the guardian executes any `exactOutput` swap through the vault. Since `exactOutput` is a standard Uniswap V3 operation, this is likely to occur during normal protocol operation.

Proof of Concept: This proof of concept can be run in:

```
test/periphery/unit/hooks/slippage/UniswapV3DexHooks.t.sol
```

```
function test_exactOutput_pathReversalBug() public {
    // Setup: Create a simple two-token path for exactOutput
    // Path encoding for exactOutput should be: tokenOut -> fee -> tokenIn
    // So for swapping tokenA for tokenB, path should be: tokenB -> fee -> tokenA
    bytes memory correctPath = abi.encodePacked(address(tokenB), FEE_30BPS, address(tokenA));

    // Mock oracle quotes to avoid reverts
    // The hook will incorrectly parse tokenB as tokenIn and tokenA as tokenOut
    // So we need to mock for these reversed addresses
    SwapUtils.mock_OracleRegistry_GetQuote({
        vm: vm,
        registry: address(oracleRegistry),
        baseAmount: TOKEN_A_AMOUNT,
        baseToken: address(tokenB), // Bug: hook thinks tokenB is tokenIn
        quoteToken: address(numeraire),
        quoteAmount: TOKENS_PRICE_IN_NUMERAIRE
    });

    SwapUtils.mock_OracleRegistry_GetQuote({
        vm: vm,
        registry: address(oracleRegistry),
        baseAmount: TOKEN_B_AMOUNT,
        baseToken: address(tokenA), // Bug: hook thinks tokenA is tokenOut
        quoteToken: address(numeraire),
        quoteAmount: TOKENS_PRICE_IN_NUMERAIRE
    });

    ISwapRouter.ExactOutputParams memory params = ISwapRouter.ExactOutputParams({
        path: correctPath,
        recipient: VAULT,
        deadline: vm.getBlockTimestamp() + 100,
        amountOut: TOKEN_B_AMOUNT,
        amountInMaximum: TOKEN_A_AMOUNT
    });

    // Call the hook and decode the return value
    vm.prank(VAULT);
    bytes memory returnData = hooks.exactOutput(params);
    (address parsedTokenIn, address parsedTokenOut,) = abi.decode(returnData, (address, address, address));

    // The bug: Hook incorrectly parses first 20 bytes as tokenIn, last 20 as tokenOut
    // This is wrong for exactOutput - it should be reversed
    assertEq(parsedTokenIn, address(tokenB), "BUG: tokenIn incorrectly parsed as tokenB (should be tokenA)");
    assertEq(parsedTokenOut, address(tokenA), "BUG: tokenOut incorrectly parsed as tokenA (should be tokenB)");

    // This means slippage checks are performed on the wrong token pair!
}
```

Recommendation: Update the `exactOutput` handler in `UniswapV3DexHooks`:

```
address tokenIn;
address tokenOut;
unchecked {
    tokenOut = address(bytes20(params.path[:20]));
    tokenIn  = address(bytes20(params.path[pathLength - 20:]));
}
```

3.2.3 `KyberSwapDexHooks` doesn't check `srcReceivers`

Submitted by davidjohn

Severity: Medium Risk

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

Summary: Admin sets `guardianRoots` for guardian to swap tokens and guard vault.. For this functionality, guardian uses `submit` function and this checks `root` using `MerkleProof` that operation is allowed. But guardian can exploit weakness of checking and steal tokens using `KyberSwapDex`.

Finding Description: When swapping with `KyberSwapDex`, `KyberSwapDexHooks` is used and `IMetaAggregationRouterV2.SwapDescriptionV2 calldata desc` is main parameter for swapping.

- KyberSwapDexHooks.sol#L19-L21.

- IMetaAggregationRouterV2.sol#L22-L28.
- IMetaAggregationRouterV2.sol#L8-L20.

```
function swap(SwapExecutionParams calldata execution)

struct SwapExecutionParams {
    address callTarget;
    address approveTarget;
    bytes targetData;
    SwapDescriptionV2 desc; // <<<
    bytes clientData;
}

struct SwapDescriptionV2 {
    IERC20 srcToken;
    IERC20 dstToken;
    address[] srcReceivers; // <<<
    uint256[] srcAmounts;
    address[] feeReceivers;
    uint256[] feeAmounts;
    address dstReceiver;
    uint256 amount;
    uint256 minReturnAmount;
    uint256 flags;
    bytes permit;
}
```

- MetaAggregationRouterV2.sol#L245.

`MetaAggregationRouterV2.sol:245` transfers tokens from vault to `srcReceivers`.

```
function _transferFromOrApproveTarget(
  address from,
  address approveTarget,
  SwapDescriptionV2 memory desc,
  bool collected
) internal {
  // if token is collected
  require(desc.srcReceivers.length == desc.srcAmounts.length, 'invalid srcReceivers length');
  if (collected) {
    if (_flagsChecked(desc.flags, _APPROVE_FUND) && approveTarget != address(0)) {
      // approve to approveTarget since some systems use an allowance proxy contract
      desc.srcToken.safeIncreaseAllowance(approveTarget, desc.amount);
      return;
    }
  }
  uint256 total;
  for (uint256 i; i < desc.srcReceivers.length; ++i) {
    total += desc.srcAmounts[i];
    _doTransferERC20(desc.srcToken, collected ? address(this) : from, desc.srcReceivers[i],
    ↪  desc.srcAmounts[i]); // <<<
  }
  require(total <= desc.amount, 'Exceeded desc.amount');
}
```

But `KyberSwapDexHooks` doesn't check `srcReceivers` and this causes guardian can steal tokens from vault.

Impact Explanation: Guardian can steal tokens from vault by valid operations. So this is high impact.

Likelihood Explanation: Guardian can exploit this easily when `KyberSwapDex` is used for swapping. And competition doesn't show guardian is trusted user (`BaseVault` checks operations of guardian but he can steal tokens). So Likelihood is high.

Proof of Concept: Using `test/periphery/fork/hooks/slippage/KyberSwapDexHooks.fork.t.sol`:

```
struct SwapDescriptionV2 {
    address srcToken;
    address dstToken;
    address[] srcReceivers;
    uint256[] srcAmounts;
    address[] feeReceivers;
    uint256[] feeAmounts;
    address dstReceiver;
    uint256 amount;
    uint256 minReturnAmount;
    uint256 flags;
    bytes permit;
}

struct SwapExecutionParams {
    address callTarget;
    address approveTarget;
    bytes targetData;
    SwapDescriptionV2 desc;
    bytes clientData;
}

function changeData(bytes memory data) public pure returns (bytes memory) { // @audit using SWAP_DATA_WETH_USDC
↪  for swap
    bytes4 selector;
    assembly {
        selector := mload(add(data, 32))
    }
    bytes memory slicedData = new bytes(data.length - 4);
    for (uint256 i = 0; i < slicedData.length; i++) {
        slicedData[i] = data[i + 4];
    }
    SwapExecutionParams memory result = abi.decode(slicedData, (SwapExecutionParams));
    address[] memory newReceivers = new address[](2);
    uint256[] memory newAmounts = new uint256[](2);
    newReceivers[0] = result.desc.srcReceivers[0]; // @audit valid receiver
    newAmounts[0] = result.desc.srcAmounts[0] - 10000; // @audit deducts tokens to steal
    newReceivers[1] = address(0xabcdef); // @audit attacker's wallet address
    newAmounts[1] = 10000; // @audit the amount of tokens to steal
    result.desc.srcReceivers = newReceivers;
    result.desc.srcAmounts = newAmounts;
    bytes memory encodedParams = abi.encode(result);
    return bytes.concat(selector, encodedParams); // @audit changed swapdata
}

function test_fork_swap_success_steal_token() public { // @audit This is modified from test_fork_swap_success
    uint256 amountIn = INITIAL_WETH_BALANCE;
    uint256 amountOutMinimum = MIN_RETURN_AMOUNT_SWAP;

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

    // Approve KYBERSWAP_ROUTER to spend WETH
    Operation[] memory ops = new Operation[](2);
    ops[0] = Operation({
        target: WETH,
        data: abi.encodeWithSelector(IERC20.approve.selector, KYBERSWAP_ROUTER, amountIn),
        clipboards: new Clipboard[](0),
        isStaticCall: false,
        callbackData: Encoder.emptyCallbackData(),
        configurableHooksOffsets: Encoder.makeExtractOffsetsArray(0),
        proof: MerkleHelper.getProof(leaves, 2),
        hooks: address(0),
        value: 0
    });
    ops[1] = Operation({
        target: KYBERSWAP_ROUTER,
        data: changeData(SWAP_DATA_WETH_USDC), // @audit changeData from valid swapdata to malicious swapdata
        clipboards: new Clipboard[](0),
        isStaticCall: false,
        callbackData: Encoder.emptyCallbackData(),
        configurableHooksOffsets: new uint16[](0),
        proof: MerkleHelper.getProof(leaves, 0),
        hooks: address(hooks),
        value: 0
    });
```

```
    // ~~~~~~~~~~ Verify ~~~~~~~~~~
    assertEq(IERC20(USDC).balanceOf(address(vault)), 0, "USDC balance should be 0");
    assertEq(IERC20(WETH).balanceOf(address(vault)), amountIn, "WETH balance should be amountIn");
    uint256 lossBefore = hooks.vaultStates(address(vault)).cumulativeDailyLossInNumeraire;
    uint256 amountInNumeraire = _convertToNumeraire(address(oracleRegistry), amountIn, WETH);
    uint256 amountOutNumeraire = _convertToNumeraire(address(oracleRegistry), amountOutMinimum, USDC);
    uint256 lossDuringTrade = _calculateLoss(amountInNumeraire, amountOutNumeraire);

    vm.expectEmit(true, true, true, true);
    emit IBaseSlippageHooks.TradeSlippageChecked(
        address(vault), WETH, USDC, amountInNumeraire, amountOutNumeraire, uint128(lossDuringTrade)
    );
    assertEq(IERC20(WETH).balanceOf(address(0xabcdef)), 0, "WETH balance should be 0"); // @audit before
    ↪  swapping // <<<
    vm.prank(users.guardian);
    vault.submit(Encoder.encodeOperations(ops));

    assertGt(IERC20(USDC).balanceOf(address(vault)), 0, "USDC balance should be greater than expected output");
    assertEq(IERC20(WETH).balanceOf(address(vault)), 0, "WETH balance should be 0");

    uint256 lossAfter = hooks.vaultStates(address(vault)).cumulativeDailyLossInNumeraire;
    assertEq(lossAfter, lossBefore + lossDuringTrade, "Loss does not match the slippage from the swap");
    assertEq(IERC20(WETH).balanceOf(address(0xabcdef)), 10000, "USDC balance should be 0"); // @audit after
    ↪  swapping - stolen tokens // <<<
}
```

Malicious guardian steals tokens from vault bypass checking.

Recommendation: In `KyberSwapDexHooks.sol`, check `srcReceivers` are valid.

3.3 Low Risk

3.3.1 Unbounded accrualLag overflow in paused branch

Submitted by didy, also found by nis70000, tough and xero1337

Severity: Low Risk

Context: PriceAndFeeCalculator.sol#L162-L181

Summary: If a vault remains paused for more than ~194 days, the `accrualLag` variable overflows due to its `uint24` type, causing the protocol to under-accrue fees for the paused period.

Finding Description: The `accrualLag` field in `VaultPriceState` is used to track the total time a vault has been paused, so that fees can be properly accrued once the vault is unpaused. However, `accrualLag` is stored as a `uint24`, which can only represent up to `2^24 - 1` seconds (~194 days). When a vault is paused for longer than this, the following code in `setUnitPrice` will overflow:

```
else {
    // Effects: set the accrual lag
    unchecked {
        // Cant overflow because _validatePriceUpdate requires timestamp > vaultPriceState.timestamp
        vaultPriceState.accrualLag = uint24(timestamp - vaultPriceState.timestamp + vaultPriceState.accrualLag);
    }
}
```

If the vault is paused for more than `2^24` seconds, the sum will wrap around, and `accrualLag` will no longer reflect the true elapsed time. When the vault is unpaused, the protocol will under-accrue fees, breaking the guarantee that fees are always correctly calculated for the full paused duration.

Impact Explanation: Medium Impact: This bug causes the protocol and vault to under-accrue fees after a long pause, resulting in lost revenue for both. However, it does not put user funds at risk or break core protocol functionality. The impact is limited to protocol/vault fee collection.

Likelihood Explanation: Low Likelihood: The bug only occurs if a vault remains paused for more than ~194 days, which is an unusual and rare scenario. Most vaults are unlikely to be paused for such an extended period under normal operation.

Proof of Concept: Add this test to `./test/core/PriceAndFeeCalculator.t.sol`:

```
function test_accrualLag_overflow_when_paused_for_long_time() public {
    _setupFees();

    // Pause the vault by violating max price tolerance ratio
    uint128 price = uint128(INITIAL_UNIT_PRICE * MAX_PRICE_TOLERANCE_RATIO / ONE_IN_BPS + 1);
    skip(MIN_UPDATE_INTERVAL_MINUTES * ONE_MINUTE);

    vm.prank(users.owner);
    priceAndFeeCalculator.setUnitPrice(BASE_VAULT, price, uint32(vm.getBlockTimestamp()));

    (VaultPriceState memory vaultPrice,) = priceAndFeeCalculator.getVaultState(BASE_VAULT);
    assertTrue(vaultPrice.paused, "Vault should be paused");

    // Simulate a very long pause (> 2^24 seconds, e.g., 200 days)
    uint256 overflowSeconds = uint256(type(uint24).max) + 1000; // Just over the uint24 max
    skip(overflowSeconds);

    // Call setUnitPrice again while still paused
    vm.prank(users.owner);
    priceAndFeeCalculator.setUnitPrice(BASE_VAULT, price, uint32(vm.getBlockTimestamp()));

    (vaultPrice,) = priceAndFeeCalculator.getVaultState(BASE_VAULT);

    // The accrualLag should have wrapped around (overflowed)
    assertLt(
        vaultPrice.accrualLag,
        overflowSeconds,
        "accrualLag should overflow and be much less than the true elapsed time"
    );
}
```

Recommendation: Store `accrualLag` as a larger type (e.g., `uint32` or `uint64`) to prevent overflow, or add a check to prevent accumulation beyond the maximum representable value.

Example Fix:

```
// Change accrualLag to uint32 in VaultPriceState
struct VaultPriceState {
    // ...existing fields...
    uint32 accrualLag; // was uint24
    // ...existing fields...
}

// And update all relevant casts and assignments accordingly:
vaultPriceState.accrualLag = uint32(timestamp - vaultPriceState.timestamp + vaultPriceState.accrualLag);
```

Alternatively, add a cap or revert if the lag would overflow.

3.3.2 Incorrect Oracle Quote Calculation in Milkman Price Checker Due to Ignored Protocol Fee

Submitted by 0xvangrim

Severity: Low Risk

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

Summary: The `MilkmanHooks.checkPrice` function calculates the expected output using the full `amountIn` without accounting for the protocol fee that will be deducted before the swap. This causes the price checker to expect more output than is actually possible, leading to valid trades being rejected.

Finding Description: The core issue is that `MilkmanHooks.checkPrice` asks the oracle for a quote using the full input amount, but the CoW Protocol settlement only swaps the input minus protocol fees. This mismatch causes the price checker to expect more output than is actually possible.

In the following step-by-step walkthrough I'll try to show how this issue manifests. For simplicity's sake we'll explain using 100 tokens as `amount_in` and 1 token as `fee_amount`:

- Step 1: Vault requests to swap 100 tokens. The vault calls MilkmanRouter's `requestSell()` function with 100 tokens. These tokens are transferred to a Milkman order contract and a `SwapRequested` event is emitted as can be seen in Milkman.sol#L62-L87.

- Step 2: Milkman bot creates the CoW order. The bot deducts the protocol fee and calculates the minimum output:

```
// https://github.com/charlesndalton/milkman-bot/blob/58904360e75416712b9858bff84ec42499f4a67c/src/main.rs#L176
// main.rs line 176-179

let sell_amount_after_fees = requested_swap.amount_in - quote.fee_amount;
// If amount_in = 100 and fee = 1, then sell_amount = 99

let buy_amount_after_fees_and_slippage =
    quote.buy_amount_after_fee * (10000 - config.slippage_tolerance_bps) / 10000;
```

  From the code snippet we can see that only 99 tokens will be swapped (100 - 1 fee). And in addition to that we get a quote from the CoW API for swapping 99 tokens together with our slippage tolerance (default 50 bps = 0.5%). This quote will be used in the `minOut` parameter of our `checkPrice` function.

- Step 3: Milkman validates the order. During settlement, Milkman will call our price checker with the `minOut`, which was based on swapping 99 tokens:

```
// https://github.com/charlesndalton/milkman/blob/f7719b59c349eb60eadaed734270a440c77345f8/src/Milkman.sol#L142-L152
// Milkman.sol line 133-142

IPriceChecker(_priceChecker).checkPrice(
    _order.sellAmount.add(_order.feeAmount),  // 99 + 1 = 100 (full amount)
    address(_order.sellToken),
    address(_order.buyToken),
    _order.feeAmount,                         // 1 (the fee)
    _order.buyAmount,                         // minOut based on swapping 99
    _priceCheckerData
)
```

- Step 4: Our `checkPrice` makes the wrong calculation. However, the current implementation in `MilkmanHooks.sol` asks the oracle about the `expectedOut` with the entirety of `amountIn`, which wrongly includes the fee:

```
// MilkmanHooks.sol lines 54-55

uint256 expectedOut = state.oracleRegistry.getQuoteForUser(
    amountIn,  // This is 100 (includes fee) - WRONG!
    fromToken,
    toToken,
    vault
);
```

In summary, the mismatch occurs because:

CoW Protocol/Milkman bot:

- Deducts protocol fee first: 100 → 99 tokens.
- Gets quote for 99 tokens: ~99 output.
- Applies slippage on the reduced amount: 99 × 0.995 = 98.505 tokens output.

Our `checkPrice`:

- Ignores protocol fee, uses full 100 tokens.
- Gets quote for 100 tokens: 100 output.
- Applies slippage on the full amount: 100 × 0.99 = 99 tokens output.

Result: `checkPrice` expects 99 minimum, but solver offers 98.505 (which is correct for swapping 99 tokens). Trade gets rejected even though it should be valid due to not deducting protocol fee (98.505 > 99 → Rejected).

Impact Explanation: Valid trades at the slippage boundary will be rejected, causing denial of service for vaults. While trades with more favorable pricing will still execute, this prevents vaults from trading when market conditions are at their acceptable limits.

Likelihood Explanation: This will occur whenever a solver quotes exactly at the slippage boundary, which is common behavior as solvers optimize to provide the minimum acceptable output.

Proof of Concept: The proof of concept can be run inside:

```
test/periphery/unit/hooks/slippage/MilkmanHooks.t.sol
```

```
function test_checkPrice_protocolFeeIgnored() public {
    // Scenario: Vault swaps 100 DAI with 1 DAI protocol fee
    uint256 amountIn = 100e18;
    uint256 protocolFee = 1e18;
    uint256 actualSwapAmount = amountIn - protocolFee; // Only 99 DAI swapped

    // Mock oracle response for FULL amount (bug: should use actualSwapAmount)
    SwapUtils.mock_OracleRegistry_GetQuoteForVault({
        vm: vm,
        registry: address(oracleRegistry),
        baseAmount: amountIn, // 100 DAI - asks about wrong amount
        baseToken: address(tokenA),
        quoteToken: address(tokenB),
        quoteAmount: 100e18, // 1:1 rate
        vault: VAULT
    });

    // checkPrice expects: 100 USDC * 0.99 (1% slippage) = 99 USDC minimum
    // But solver knows only 99 DAI are swapped, so offers: 99 * 0.995 = 98.505 USDC
    uint256 solverMinOut = 98505e15;

    bool result = hooks.checkPrice(
        amountIn,
        address(tokenA),
        address(tokenB),
        protocolFee, // Ignored by checkPrice
        solverMinOut,
        abi.encode(VAULT)
    );

    assertFalse(result, "Valid trade rejected: checkPrice expects 99 USDC but solver offers 98.505");

    // Verify the trade would pass if we used the correct amount
    SwapUtils.mock_OracleRegistry_GetQuoteForVault({
        vm: vm,
        registry: address(oracleRegistry),
        baseAmount: actualSwapAmount, // 99 DAI - correct amount
        baseToken: address(tokenA),
        quoteToken: address(tokenB),
        quoteAmount: 99e18,
        vault: VAULT
    });

    // With correct calculation: expects 99 * 0.99 = 98.01 USDC minimum
    // Solver offers 98.505, so it passes
    assertTrue(solverMinOut > (99e18 * 9900 / 10000), "Trade would pass with correct fee accounting");
}
```

Recommendation: Subtract the fee amount before querying the oracle.

3.4 Informational

3.4.1 DoS When Depositing With Max Token Allowance in `SingleDepositorVault`

Submitted by mohitisimmortal

Severity: Informational

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

Summary: Deposit of tokens will fail in SingleDepositor vault during user gives max allowance of ERC20's.

Finding Description: It's a comman practice to give max `allowance(type(uint256).max)` to contracts whom user trusts, now when a user gives max allowance to Single depositor vault, and try to deposit tokens, it will revert because of this check:

```
// In src/core/SingleDepositorVault.sol:
27    function deposit(TokenAmount[] calldata tokenAmounts) external requiresAuth {
      // ...

35            // Requirements: allowance must be zero after deposit
36            require(
37>>               tokenAmount.token.allowance(msg.sender, address(this)) == 0,
38                Aera__UnexpectedTokenAllowance(tokenAmount.token.allowance(msg.sender, address(this)))
39            );
```

- It tries to check that all allowance must be used up, but during max allowance- erc20 contract not reduce used up allowance-.

```
// from ERC20.sol:
function _spendAllowance(address owner, address spender, uint256 value) internal virtual {
    uint256 currentAllowance = allowance(owner, spender);
    if (currentAllowance < type(uint256).max) { // <<< not reducing allowance if max allowance is there
        if (currentAllowance < value) {
            revert ERC20InsufficientAllowance(spender, currentAllowance, value);
        }
        unchecked {
            _approve(owner, spender, currentAllowance - value, false);
        }
    }
}
```

Impact Explanation: DoS of deposit into single depositor vault.

Likelihood Explanation: Common practice that user give max allowance to trusted contracts.

Proof of Concept:

1. Put the test below into: `test/core/unit/SingleDepositorVaultTest.sol`.

```
function test_deposit_fail_when_allowance_max() public {
    uint256 amountA = 100e18;

    tokenA.mint(users.owner, amountA);

    vm.startPrank(users.owner);
    tokenA.approve(address(singleDepositorVault), type(uint256).max);

    TokenAmount[] memory tokenAmounts = new TokenAmount[](1);
    tokenAmounts[0] = TokenAmount({ token: IERC20(address(tokenA)), amount: amountA });
    vm.expectRevert(abi.encodeWithSelector(ISingleDepositorVault.Aera__UnexpectedTokenAllowance.selecto
    ↪  r,type(uint256).max ));

    singleDepositorVault.deposit(tokenAmounts);
}
```

2. Run with `forge test --mt test_deposit_fail_when_allowance_max -vvvv`.
3. Output:

```
[Revert] Aera__UnexpectedTokenAllowance(115792089237316195423570985008687907853269984665640564039457584
↪  007913129639935 [1.157e77])
```

Recommendation: Make a conditional that if max allowance then ignore that require check, else do.