CANTINA

Instadapp fluid

Competition

January 7, 2025

Contents

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
Critical	*Must* fix as soon as possible (if already deployed).
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.
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 table. Critical findings have a high likelihood of being exploited and must be addressed immediately. High findings are almost certain to occur, easy to perform, or not easy but highly incentivized thus must be fixed as soon as possible.

Medium findings are conditionally possible or incentivized but are still relatively likely to occur and should be addressed. Low findings a rare combination of circumstances to exploit, or offer little to no incentive to exploit but are recommended to be addressed.

Lastly, some findings might represent objective improvements that should be addressed but do not impact the project's overall security (Gas and Informational findings).

2 Security Review Summary

The Instadapp platform is a full feature platform for both users and developers to leverage the full potential of DeFi. The Instadapp protocol ('DSL') acts as the middleware that aggregates multiple DeFi protocols into one upgradable smart contract layer.

From Sep 11th to Oct 2nd Cantina hosted a competition based on instadapp-fluid. The participants identified a total of **13** issues in the following risk categories:

- Critical Risk: 0
- High Risk: 0
- Medium Risk: 2
- Low Risk: 7
- Gas Optimizations: 0
- Informational: 4

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

3 Findings

3.1 Medium Risk

3.1.1 The TWAP price will be incorrect due to the wrong update of the oracle

Submitted by etherSky, also found by Draiakoo and carlos404

Severity: Medium Risk

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

Description: Users can retrieve the `TWAP price` in the same way they do with other `DEXs` like `Uniswap` and rely on that value as a `source of truth`. Therefore, it's crucial to calculate the `TWAP price` accurately, as most users of `Fluid DEX` will depend on it. However, the `oracle data` is being updated incorrectly, leading to an extremely inaccurate `TWAP price`.

There is an `oracle map` to store `oracle data`, and its length is defined by `TOTAL_ORACLE_MAPPING`. Similar to `Uniswap`, once the entire `map` is occupied, it will be overwritten.

- helpers.sol#L1239

```
  function _updateOracle(
      uint newPrice_,
      uint centerPrice_,
      uint dexVariables_
  ) internal returns (uint) {
          temp_ = (dexVariables_ >> 154) & X22;
          uint nextOracleSlot_ = ((dexVariables_ >> 176) & X3);
          uint oracleMap_ = (dexVariables_ >> 179) & X16;

  1202:         if (temp_ > X9) {
              if (nextOracleSlot_ > 0) {
                  // if greater than 0 then current slot has 2 or more oracle slot empty
                  // First 9 bits are of time, so not using that
                  temp3_ = (temp3_ << 41) | (temp_ << 9);
  1207:               _oracle[oracleMap_] = _oracle[oracleMap_] | (temp3_ << (--nextOracleSlot_ * 32));
                  if (nextOracleSlot_ > 0) {
                      --nextOracleSlot_;
                  } else {
                      // if == 0 that means the oracle slots will get filled and shift to next oracle map
                      nextOracleSlot_ = 7;
                      oracleMap_ = (oracleMap_ + 1) % TOTAL_ORACLE_MAPPING;
                      _oracle[oracleMap_] = 0;
                  }
              } else {
                  // if == 0
                  // then seconds will be in last map
                  // precision will be in last map + 1
                  // Storing precision & sign slot in first precision & sign slot and leaving time slot
↪   empty
                  temp3_ = temp3_ << 9;
                  _oracle[oracleMap_] = _oracle[oracleMap_] | temp3_;
                  nextOracleSlot_ = 6; // storing 6 here as 7 is going to occupied right now
                  oracleMap_ = (oracleMap_ + 1) % TOTAL_ORACLE_MAPPING;
                  // Storing time in 2nd precision & sign and leaving time slot empty
                  _oracle[oracleMap_] = temp_ << ((7 * 32) + 9);
              }
          } else {
              temp3_ = (temp3_ << 9) | temp_;
              if (nextOracleSlot_ < 7) {
                  _oracle[oracleMap_] = _oracle[oracleMap_] | (temp3_ << (nextOracleSlot_ * 32));
              } else {
                  _oracle[oracleMap_] = temp3_ << ((7 * 32));
              }
              if (nextOracleSlot_ > 0) {
                  --nextOracleSlot_;
              } else {
                  nextOracleSlot_ = 7;
  1239:               oracleMap_ = (oracleMap_ + 1) % TOTAL_ORACLE_MAPPING;
              }
          }
      }
  }
```

When updating the `oracle data`, either one or two `slots` are written depending on whether the `time difference` is greater than 511 ($2^9 - 1$) or not (`line 1202`). The problem arises because we are not properly clearing the `old oracle map` when it is overwritten. Suppose the full `oracle map` has been written, and we loop back to the `first map`.

Now, the next available `slot` is the last `slot` of the `first map` (`slot number 0`). If the `time difference` is less than `511`, only `one slot` is needed.

In `line 1239`, this last `slot` is written, and we proceed to the `second map`. However, the `second map` is not cleared. During the next update, if the `time difference` is greater than `511` and two `slots` are needed, the process will reach `line 1207`. At this point, the second `map` is overwritten with old data, leading to incorrect stored `oracle data`.

Let me walk through a specific example in the `PoC` to illustrate this issue.

Impact: The impact of this issue is high because most users will rely on the `TWAP price` as they do in `Uniswap`. Typically, `TWAP` is considered more accurate and stable than the current spot price. In protocols integrated with `Uniswap`, `TWAP` is preferred over spot price since the latter can be easily manipulated. The same scenario applies to `Fluid DEX`.

Likelihood: The likelihood of this issue occurring is high, as it can easily happen, and once it does, the `oracle data` will become severely inaccurate.

Proof of Concept: Add below test to the `pool.t.sol`:

```
function updateOracleOneStep(DexParams memory dexPool_, DexType dexType_, uint256 skipTime) public {
    _testDepositPerfectColLiquidity(dexPool_, bob, 1e18, dexType_);
    skip(skipTime);
    _testSwapExactIn(dexPool_, alice, dexPool_.token0Wei / 100, true, dexType_, true, false, true);
}

function logOracleMapAndSlot(FluidDexT1 dex_) public {
    DexVariablesData memory d_ = _getDexVariablesData(dex_);
    console2.log('oracle map & oracle slot                    => ', d_.oracleMap, d_.oracleSlot);
}

function testOracleUpdate() public {
    /**
        1. For testing purposes, set TOTAL_ORACLE_MAPPING to 10
        */
    uint256 totalOracleMapping = 10;

    DexParams memory DAI_USDC_TEST = _deployPoolT1(address(DAI), address(USDC), "DAI_USDC_TEST",
↪   totalOracleMapping);
    DexAdminStrcuts.InitializeVariables memory i_;
    _setUpDexParams(DAI_USDC_TEST, 1e27, 1e4 * DAI_USDC_TEST.token0Wei, 1e4 * DAI_USDC_TEST.token0Wei, i_);

    DexType dexType_ = DexType.SmartColAndDebt;
    DexParams memory dexPool_ = DAI_USDC_TEST;
    FluidDexT1 dex_ = _getDexType(dexPool_, dexType_, false);

    /**
        2. Initially, there is no oracle data, so we need to write the first slot of the first oracle map.
        */
    logOracleMapAndSlot(dex_);

    /**
        3. Each time we update the oracle by executing a swap, the old oracle data is written to the oracle
↪   map.
        Since the skip time of 600 is greater than 511 (2^9 - 1),
            we write two slots during one update: one for the price difference and one for the time difference.
        Here, Please note that during the first update, the original time difference is 60 seconds, so only
↪   one slot is written.

        dexVariables = (i_.centerPrice << 1) |
            (i_.centerPrice << 41) |
            (i_.centerPrice << 81) |
            (block.timestamp << 121) |
            (60 << 154) | // just setting 60 seconds, no particular reason for it why "60"  // @audit, here
            (7 << 176);
        */
    for (uint256 i = 0; i < totalOracleMapping * 4; i ++) {
        updateOracleOneStep(dexPool_, dexType_, 600);
    }
```

```
    /**
        4. The oracle map and slot occupancy is as follows:
            First 4 updates: [0.7], [0.6, 0.5], [0.4, 0.3], [0.2, 0.1]
            Next 4 updates: [0.0, 1.7], [1.6, 1.5], [1.4, 1.3], [1.2, 1.1]
            ...
            Final 4 updates: [8.0, 9.7], [9.6, 9.5], [9.4, 9.3], [9.2, 9.1]
        Here, f represents the oracle map and s represents the slot in the format [f.s].

        We are now ready to write to the last slot of the last map.
        */
    logOracleMapAndSlot(dex_);

    /**
        5. Calculate the latest price difference.
        This can be retrieved from slot 2 of the last map.
        As we can see, it's quite small due to the swap amount is extremely small.
        */
        uint256 oracleMapData_0 = dex_.readFromStorage(
        _calculateMappingStorageSlotForUint256(6, 9)  // 9 is oracle map (the last map)
    );
    uint256 priceDiff_0 = ((oracleMapData_0 >> (2 * 32)) & X32) >> 10; // 2 is slot number
    console2.log('latest price diff                          => ', priceDiff_0);

    /**
        6. We update the oracle three times for testing purposes, which occupies the following slots: [9.0,
↪   0.7], [0.6, 0.5], [0.4, 0.3].
        Since all oracle maps have been written, we loop back to the first oracle map.
        */
    for (uint256 i = 0; i < 3; i ++) {
        updateOracleOneStep(dexPool_, dexType_, 600);
    }
    /**
        7. Now, the next available oracle map and slot is [0.2], and the time difference stored in
↪   dexVariables is 600.
        */
    logOracleMapAndSlot(dex_);

    /**
        8. We update the oracle once again, and the current time difference is 100, which will be stored in
↪   dexVariables.
        Since the previous time difference was 600, we write to two slots: [0.2, 0.1]
        */
    updateOracleOneStep(dexPool_, dexType_, 100);

    /**
        9. Now, the next available oracle map and slot is [0.0], and the time difference stored in
↪   dexVariables is 100.
        */
    logOracleMapAndSlot(dex_);

    /**
        10. We update the oracle once again, and the current time difference is 800, which will be stored in
↪   dexVariables.
        Since the previous time difference was 100, we write to one slot: [0.0]
        */
    updateOracleOneStep(dexPool_, dexType_, 800);

    /**
        11. Now, the next available oracle map and slot is [1.7], and the time difference stored in
↪   dexVariables is 800.
        */
    logOracleMapAndSlot(dex_);

    /**
        12. Now we are ready to write to oracle map 1, which should be empty for new updates.
        However, the old map data still remains
        */
    uint256 oracleMapData_1 = dex_.readFromStorage(
        _calculateMappingStorageSlotForUint256(6, 1)
    );
    console2.log('the first oracle map data                  => ', oracleMapData_1);

    /**
        13. In the next oracle update, since the time difference is 800, which is larger than 512, two slots
↪   [1.7, 1.6] will be written.
        However, the old data was written in the 6th slot still exists, and we can verify that value.
```

```
        */
    uint256 originalDataInSlot_6_OfOracleMap_1_ = ((oracleMapData_1 >> (6 * 32)) & X32) >> 9;
    console2.log('old data in the 6th slot of the first map  => ', originalDataInSlot_6_OfOracleMap_1_);

    /**
        14. We update the oracle once again.
        Since the previous time difference was 800, we write to two slots: [1.7, 1.6]
        */
    updateOracleOneStep(dexPool_, dexType_, 100);
    /**
        15. Now, the next available oracle map and slot is [1.5].
        */
    logOracleMapAndSlot(dex_);

    /**
        16. We retrieve the time difference from this slot [1.7, 1.6], which should be 801.
        Here, we account for the 1-second skip in the _testSwapExactIn function.
        However, the stored value is 809.
        We can deduce that 809 = 801 | 8, where 8 is the old value in that slot.
        This occurs because we did not clear the oracle map.
        */
    (OracleMapData memory data_1, ) = _getOracleMapData(dex_, 1, 7);
    console2.log('time difference                          => ', data_1.timeDiff);
    uint256 oracleMapData_2 = dex_.readFromStorage(
        _calculateMappingStorageSlotForUint256(6, 1)
    );

    /**
        17. Additionally, the stored price difference is larger than the correct value. (300 > 4)
        */
    uint256 priceDiff_2 = ((oracleMapData_2 >> (7 * 32)) & X32) >> 10;
    console2.log('stored price diff                       => ', priceDiff_2);
}
```

And the log is:

```
oracle map & oracle slot                 =>  0 7
  oracle map & oracle slot                 =>  9 0
  latest price diff                        =>  4
  oracle map & oracle slot                 =>  0 2
  oracle map & oracle slot                 =>  0 0
  oracle map & oracle slot                 =>  1 7
  the first oracle map data                =>
↪  82958991088679686816544419826934165826892714737239636158569177629757317 12
  old data in the 6th slot of the first map  =>  8
  oracle map & oracle slot                 =>  1 5
  time difference                          =>  809
  stored price diff                        =>  300
```

Recommendation:

```
  function _updateOracle(
      uint newPrice_,
      uint centerPrice_,
      uint dexVariables_
  ) internal returns (uint) {
      temp_ = (dexVariables_ >> 154) & X22;
      uint nextOracleSlot_ = ((dexVariables_ >> 176) & X3);
      uint oracleMap_ = (dexVariables_ >> 179) & X16;
      if (temp_ > X9) {

      } else {
          temp3_ = (temp3_ << 9) | temp_;
          if (nextOracleSlot_ < 7) {
              _oracle[oracleMap_] = _oracle[oracleMap_] | (temp3_ << (nextOracleSlot_ * 32));
          } else {
              _oracle[oracleMap_] = temp3_ << ((7 * 32));
          }
          if (nextOracleSlot_ > 0) {
              --nextOracleSlot_;
          } else {
              nextOracleSlot_ = 7;
              oracleMap_ = (oracleMap_ + 1) % TOTAL_ORACLE_MAPPING;

+             _oracle[oracleMap_] = 0;
          }
      }
  }
```

Instadapp: The impact of this issue is high if external protocols are solely relying on Fluid's oracle. Fluid DEX itself doesn't have a high impact here.

3.1.2 The Fluid DEX may become locked due to the division by zero error

Submitted by etherSky

Severity: Medium Risk

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

Description: The `center price` can actively `shift` from the old value to a new one, and the `upper` and `lower percent` can gradually change to new values. The `26th` and `248th` bits of `dexVariables2` are used to indicate whether these `shifts` are `active`. When the `center price shift` is `active` (indicated by the `248th` bit being set to `1`), `_centerPriceShift` should not be `zero`, as there is a division by the `shift time` in the `_calcCenterPrice` function.

However, there is an error and dividing by zero can happen in the `_getPricesAndExchangePrices` function. If this division by zero occurs, the `DEX` will lock because all operations in the `Fluid DEX` call the `_getPricesAndExchangePrices` function.

In the `_getPricesAndExchangePrices` function, when the `center price shift` is `active`, the `_calcCenterPrice` function is called at `line 235`.

- helpers.sol#L235

```
  function _getPricesAndExchangePrices(
      uint dexVariables_,
      uint dexVariables2_
  ) internal returns (
      PricesAndExchangePrice memory pex_
  ) {
      uint centerPrice_;

      if (((dexVariables2_ >> 248) & 1) == 0) {

      } else {
  235:        centerPrice_ = _calcCenterPrice(dexVariables_, dexVariables2_);
      }
  }
```

In the `_calcCenterPrice` function, if the `price` has shifted sufficiently to reach the new `center price`, `_centerPriceShift` is set to `0` at lines `168` and `185`, and the `248th` bit of `dexVariables2` is cleared (set

to 0) at `lines 171` and 188.

- helpers.sol#L168

```
  function _calcCenterPrice(uint dexVariables_, uint dexVariables2_) internal returns (uint
↪  newCenterPrice_) {
      if (newCenterPrice_ > oldCenterPrice_) {
          oldCenterPrice_ += priceShift_;
          if (newCenterPrice_ > oldCenterPrice_) {
              newCenterPrice_ = oldCenterPrice_;
          } else {
  168:            delete _centerPriceShift;
  171:            dexVariables2 = dexVariables2 & ~uint(1 << 248);
          }
      } else {
          unchecked {
              oldCenterPrice_ = oldCenterPrice_ > priceShift_ ? oldCenterPrice_ - priceShift_ : 0;
          }
          if (newCenterPrice_ < oldCenterPrice_) {
              newCenterPrice_ = oldCenterPrice_;
          } else {
  185:            delete _centerPriceShift;
  188:            dexVariables2 = dexVariables2 & ~uint(1 << 248);
          }
      }
  }
```

In the `_getPricesAndExchangePrices` function, if the `percent change` is `active`, the `_calcRangeShifting` function is called at `line 245`.

- helpers.sol#L245

```
  function _getPricesAndExchangePrices(
      uint dexVariables_,
      uint dexVariables2_
  ) internal returns (
      PricesAndExchangePrice memory pex_
  ) {
      uint centerPrice_;

      if (((dexVariables2_ >> 248) & 1) == 0) {

      } else {
  235:        centerPrice_ = _calcCenterPrice(dexVariables_, dexVariables2_);
      }

      if (((dexVariables2_ >> 26) & 1) == 1) {
  245:        (upperRange_, lowerRange_, dexVariables2_) = _calcRangeShifting(upperRange_, lowerRange_,
↪  dexVariables2_);
      }
  }
```

This function is called with `dexVariables2_`, which is an old copy of `dexVariables2`. Remember that `dexVariables2` was already updated in the `_calcCenterPrice` function.

In the `_calcRangeShifting` function, if the `shifting period` has ended, `_rangeShift` is cleared to `0`, and the `26th` bit of `dexVariables2_` is cleared at `line 91`.

- helpers.sol#L91

```
  function _calcRangeShifting(uint upperRange_, uint lowerRange_, uint dexVariables2_) internal returns
↪  (uint, uint, uint) {
      if ((startTimeStamp_ + shiftDuration_) < block.timestamp) {
          delete _rangeShift;
  91:        dexVariables2_ = dexVariables2_ & ~uint(1 << 26);
  92:        dexVariables2 = dexVariables2_;
          return (upperRange_, lowerRange_, dexVariables2_);
      }
  }
```

The issue arises because the updated `dexVariables2_` is then copied back to `dexVariables2` at `line 92`. At this point, the `248th` bit of `dexVariables2_` is still `1`, causing `dexVariables2` to incorrectly recover the `248th` bit to `1`. However, `_centerPriceShift` is already `0`.

As a result, in the next call to `_getPricesAndExchangePrices`, a division by zero occurs in the `_calcCenterPrice` function at `line 159`.

- helpers.sol#L159

```
  function _calcCenterPrice(uint dexVariables_, uint dexVariables2_) internal returns (uint
↪  newCenterPrice_) {
      uint centerPriceShift_ = _centerPriceShift;
      uint startTimeStamp_ = centerPriceShift_ & X33;
      uint percent_ = (centerPriceShift_ >> 33) & X20;
      uint time_ = (centerPriceShift_ >> 53) & X20;

      uint fromTimeStamp_ = (dexVariables_ >> 121) & X33;
      fromTimeStamp_ = fromTimeStamp_ > startTimeStamp_ ? fromTimeStamp_ : startTimeStamp_;

      newCenterPrice_ = ICenterPrice(AddressCalcs.addressCalc(DEPLOYER_CONTRACT, ((dexVariables2_ >> 112)
↪  & X30))).centerPrice();
  159:    uint priceShift_ = (oldCenterPrice_ * percent_ * (block.timestamp - fromTimeStamp_)) / (time_ *
↪  SIX_DECIMALS);
  }
```

Impact: Since all operations in the `Fluid DEX` call `_getPricesAndExchangePrices`, this leads to a high-impact issue. Although the issue is severe, the `DEX admin` can resolve it by updating the `center price address`, so I have marked this as medium in impact.

Likelihood: The likelihood is at least medium.

Proof of Concept: Add below test to `pool.t.sol`:

```
function testCenterPriceShift() public {
    DexParams memory dexPool_ = DAI_USDC;
    DexType dexType_ = DexType.SmartCol;
    FluidDexT1 dex_ = _getDexType(dexPool_, dexType_, false);

    uint256 upperPercent_ = 11 * 1e4;
    uint256 lowerPercent_ = 11 * 1e4;
    uint256 shiftTime_ = 600;

    /**
        1. Currently, the upper and lower percents are set to 10%.
        We want to adjust these to 11% over the next 600 seconds.
        */
    vm.prank(address(admin));
    FluidDexT1Admin(address(dex_)).updateRangePercents(
        upperPercent_,
        lowerPercent_,
        shiftTime_
    );
    vm.stopPrank();

    /**
        2. Deploy the mock contract for the center price.
        */
    vm.prank(bob);
    contractDeployerFactory.deployContract(type(MockDexCenterPrice).creationCode);
    vm.stopPrank();

    /**
        3. centerPriceAddress_ serves as the nonce used to calculate the address of the deployed mock contract.
        */
    uint256 centerPriceAddress_ = contractDeployerFactory.totalContracts();
    uint256 percent_ = 10 * 1e4;
    uint256 time_ = 600;

    /**
        4. We want to shift the center price by 10% per 600 seconds.
        */
    vm.prank(address(admin));
    FluidDexT1Admin(address(dex_)).updateCenterPriceAddress(
        centerPriceAddress_,
        percent_,
        time_
    );
    vm.stopPrank();
```

```
    MockDexCenterPrice mockDexCenterPrice_ =
↪   MockDexCenterPrice(contractDeployerFactory.getContractAddress(centerPriceAddress_));

    /**
        5. The current center price is initially set to 1e27.
        The new center price will be set to 110% of the original value,
        and the center price will gradually shift to this new value over the 600 seconds.
        */
    uint256 newCenterPrice_ = 1e27 + 1e26;
    mockDexCenterPrice_.setPrice(newCenterPrice_);
    assertEq(mockDexCenterPrice_.centerPrice(), newCenterPrice_);

    /**
        6. The percent change and center price shift are then activated in dexVariables2
            is percent change active     =>  true
            is center price shift active =>  true
        */
    DexVariables2Data memory dexVariables2_ = _getDexVariables2Data(dex_);
    console2.log('is percent change active     => ', dexVariables2_.isPercentChangeActive);
    console2.log('is center price shift active => ', dexVariables2_.isCenterPriceShiftActive);

    /**
        7. After advancing the time by 1200 seconds, the percent change and price shift are completed
        */
    skip(time_ * 2);

    /**
        8. Perform any operation that triggers the _getPricesAndExchangePrices function.
        */
    _testDepositPerfectColLiquidity(dexPool_, bob, 1e18, dexType_);

    /**
        9. The percent change is no longer active, but the center price shift remains active.
            is percent change active     =>  false
            is center price shift active =>  true
        */
    DexVariables2Data memory dexVariables2_1 = _getDexVariables2Data(dex_);
    console2.log('is percent change active     => ', dexVariables2_1.isPercentChangeActive);
    console2.log('is center price shift active => ', dexVariables2_1.isCenterPriceShiftActive);

    /**
        10. Although the center price shift is still active, _centerPriceShift becomes 0, leading to a
↪   division by zero error.
                _centerPriceShift            =>  0 0 0
        */
    DexCenterPriceShiftData memory cps_ = _getDexCenterPriceShiftData(dex_);
    console2.log('_centerPriceShift            => ', cps_.timestampOfShiftStart, cps_.percentShift,
↪   cps_.timeToShiftPercent);

    /**
        11. All operations will be reverted due to the division by zero error.
        */
    _testDepositPerfectColLiquidity(dexPool_, bob, 1e18, dexType_);
}
```

The revert reason is:

```
 DAI_USDC:::Col::::depositPerfect(1000000000000000000 [1e18],
↪   115792089237316195423570985008687907853269984665640564039457584007913129639935 [1.157e77],
↪   115792089237316195423570985008687907853269984665640564039457584007913129639935 [1.157e77], false)
        [266] MockDexCenterPrice::centerPrice()
           ← [Return] 110000000000000000000000000 [1.1e27]
        ← [Revert] panic: division or modulo by zero (0x12)
     ← [Revert] panic: division or modulo by zero (0x12)
```

Recommendation:

```
  function _getPricesAndExchangePrices(
      uint dexVariables_,
      uint dexVariables2_
  ) internal returns (
      PricesAndExchangePrice memory pex_
  ) {
      uint upperRange_ = ((dexVariables2_ >> 27) & X20);
      uint lowerRange_ = ((dexVariables2_ >> 47) & X20);
      if (((dexVariables2_ >> 26) & 1) == 1) {
-         (upperRange_, lowerRange_, dexVariables2_) = _calcRangeShifting(upperRange_, lowerRange_,
↪  dexVariables2_);
+         (upperRange_, lowerRange_, dexVariables2_) = _calcRangeShifting(upperRange_, lowerRange_,
↪  dexVariables2);
      }
  }
```

Instadapp: We feel the likelihood to be low but will still consider this a medium severity. Why likelihood is low?

- Governance need to update center price & range in the same proposal with same shifting time.
- Center price can end before the shifting time if price is reached but range will only once entire shifting time is reached.

If this situation happens then yes, pool can get locked until governance intervention which can take 4 days to unlock the pool by updating the configs.