Agora StableSwap Security Review

Pashov Audit Group

Conducted by: 0xbepresent, Shaka, 0xunforgiven, zark

June 5th 2025 - June 11th 2025

Contents

1. About Pashov Audit Group

Pashov Audit Group consists of multiple teams of some of the best smart contract security researchers in the space. Having a combined reported security vulnerabilities count of over 1000, the group strives to create the absolute very best audit journey possible - although 100% security can never be guaranteed, we do guarantee the best efforts of our experienced researchers for your blockchain protocol. Check our previous work here or reach out on Twitter @pashovkrum.

2. Disclaimer

A smart contract security review can never verify the complete absence of vulnerabilities. This is a time, resource and expertise bound effort where we try to find as many vulnerabilities as possible. We can not guarantee 100% security after the review or even if the review will find any problems with your smart contracts. Subsequent security reviews, bug bounty programs and on-chain monitoring are strongly recommended.

3. Introduction

A time-boxed security review of the **amphora-atlas/stable-swap-dev** repository was done by **Pashov Audit Group**, with a focus on the security aspects of the application's smart contracts implementation.

4. About Agora StableSwap

Agora StableSwap is a DEX for stablecoins and real-world assets (RWAs) that uses fixed pricing with interest rate adjustments instead of oracles, supporting assets across multiple EVM chains. It features KYC-restricted access, Uniswap router compatibility, and subgraph-based management.

5. Risk Classification

Severity	Impact: High	Impact: Medium	Impact: Low
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

5.1. Impact

- High - leads to a significant material loss of assets in the protocol or significantly harms a group of users.
- Medium - only a small amount of funds can be lost (such as leakage of value) or a core functionality of the protocol is affected.
- Low - can lead to any kind of unexpected behavior with some of the protocol's functionalities that's not so critical.

5.2. Likelihood

- High - attack path is possible with reasonable assumptions that mimic on-chain conditions, and the cost of the attack is relatively low compared to the amount of funds that can be stolen or lost.
- Medium - only a conditionally incentivized attack vector, but still relatively likely.
- Low - has too many or too unlikely assumptions or requires a significant stake by the attacker with little or no incentive.

5.3. Action required for severity levels

- Critical - Must fix as soon as possible (if already deployed)
- High - Must fix (before deployment if not already deployed)
- Medium - Should fix
- Low - Could fix

6. Security Assessment Summary

review commit hashes:

- 1dedf62430e2fcf164a807f95c80c12615bad135

fixes review commit hash:

- 0b424f359bc22a80f681a92440cf5746e5b7dcf8

Scope

The following smart contracts were in scope of the audit:

- `AgoraStableSwapAccessControl`
- `AgoraStableSwapFactory`
- `AgoraStableSwapPair`
- `AgoraStableSwapPairConfiguration`
- `AgoraStableSwapPairCore`

7. Executive Summary

Over the course of the security review, 0xbepresent, Shaka, 0xunforgiven, zark engaged with Agora to review Agora StableSwap. In this period of time a total of **16** issues were uncovered.

Protocol Summary

Protocol Name	Agora StableSwap
Repository	https://github.com/amphora-atlas/stable-swap-dev
Date	June 5th 2025 - June 11th 2025
Protocol Type	DEX

Findings Count

Severity	**Amount**
Medium	1
Low	15
Total Findings	**16**

Summary of Findings

ID	Title	Severity	Status
[M-01]	Wrong price calculation with inverse base and negative interest rates	Medium	Acknowledged
[L-01]	Frontrun sync() lets attacker steal tokens before reserve update	Low	Acknowledged
[L-02]	initialize may revoke admin if initialAdmin is msg.sender	Low	Resolved
[L-03]	Incorrect zero address check only validates token0 instead of token1	Low	Resolved
[L-04]	Pools are not compatible with Uniswap V2 routers	Low	Acknowledged
[L-05]	Unexpected behavior for fee on transfer tokens	Low	Acknowledged
[L-06]	setFeeBounds change not applied, causing outdated purchase fees	Low	Acknowledged
[L-07]	claimFor allows attackers to steal rewards via flash swap	Low	Acknowledged
[L-08]	calculatePrice linear model causes yield token pricing errors	Low	Acknowledged
[L-09]	configureOraclePrice can be exploited if price exceeds fee	Low	Acknowledged
[L-10]	configureOraclePrice() missing timestamp causes price errors	Low	Resolved
[L-11]	getAmountsIn fails to validate sufficient reserves for token out fees	Low	Resolved
[L-12]	Token address validation missing allows fee drain via alternate routes	Low	Acknowledged

[L-13]	Underflow in negative interest rate calculation leads to denial of service	Low	Acknowledged
[L-14]	Using fixed pair prices allows arbitrage	Low	Acknowledged
[L-15]	Stale fees cause reserve miscalculation, enabling asset drain	Low	Acknowledged

8. Findings

8.1. Medium Findings

[M-01] Wrong price calculation with inverse base and negative interest rates

Severity

Impact: High

Likelihood: Low

Description

To calculate `token0OverToken1` for tokens that accrue value, `swapStorage.perSecondInterestRate` is used to adjust the base price of the asset. Depending on the order of the tokens, this value can be positive or negative.

However, this approach is flawed, as applying a negative interest rate over the inverse price does not yield the correct result.

Consider the following example:

- XYZ is token0 (6 decimals).
- AUSD is token1 (6 decimals).
- XYZ price is $2 and its annual interest rate is 50%.
- token0OverToken1 is set to 0.5e18 (1/2 * 1e18) and annualizedInterestRate is set to -0.5e18 (negative 50% interest rate).
- After 1 year, XYZ price is $3, so token0OverToken1 should be ~0.33e18 (1/3 * 1e18).
- token0OverToken1 is calculated as 0.5e18 * (1 - 0.5) = 0.25e18, which is incorrect.

Recommendations

Use always the price of the non-AUSD token as the base price and avoid using the inverse price, as this breaks the linear interest calculations. This will require that AUSD is always token0 in the pair. Note that given that the AUSD address has 8 leading zeros, inverting the order of the tokens will result in AUSD being token0 in most cases, but this is not guaranteed. So it should be required to enforce it explicitly in the code.

8.2. Low Findings

[L-01] Frontrun `sync()` lets attacker steal tokens before reserve update

To increase reserves, the admin must first transfer tokens to the pair contract and then call `sync()` to update the reserves.

```
function _sync(
        uint256 _token0Balance,
        uint256 _token1Balance,
        uint256 _token0FeesAccumulated,
        uint256 _token1FeesAccumulated
    ) internal {
        uint112 token0Reserves =
          (_token0Balance - _token0FeesAccumulated).toUint112();
        uint112 token1Reserves =
          (_token1Balance - _token1FeesAccumulated).toUint112();

        _getPointerToStorage().swapStorage.reserve0 = token0Reserves;
        _getPointerToStorage().swapStorage.reserve1 = token1Reserves;
        _getPointerToStorage
          ().swapStorage.token0FeesAccumulated = _token0FeesAccumulated.toUint128();
        _getPointerToStorage
          ().swapStorage.token1FeesAccumulated = _token1FeesAccumulated.toUint128();

        // emit event
        emit Sync({ reserve0: token0Reserves, reserve1: token1Reserves });
    }
```

However, if these actions are done in separate transactions, an attacker can frontrun the `sync()` call by triggering a swap before the reserves are updated. This causes the contract to miscalculate available reserves, allowing the attacker to steal the deposited tokens without providing input tokens.

[L-02] `initialize` may revoke admin if `initialAdmin` is `msg.sender`

In both `Pair` and `PairFactory` contracts, the `initialize` function sets `_initialAdminAddress` as the admin of `ACCESS_CONTROL_MANAGER_ROLE`, then temporarily assigns `msg.sender` to `ACCESS_CONTROL_MANAGER_ROLE`, and finally after all init operations, revokes this role from `msg.sender`:

```
function initialize(InitializeParams memory _params) external initializer {
        // Set the admin role
@1>     _initializeAgoraAccessControl
  ({ _initialAdminAddress: _params.initialFactoryAccessControlManager });

        // `proxyAdminAddress` is the proxy admin for the proxy contracts
        // generated with this factory
        _getPointerToFactoryStorage
          ().proxyAdminAddress = _params.initialStableSwapProxyAdminAddress;

        // We temporarily set the manager role to the sender to set the
        // defaultValues on initialization
@2>     _assignRole(
  {_role:ACCESS_CONTROL_MANAGER_ROLE,
  _newAddress:msg.sender,
  _addRole:true}
);
       --Snipped--
@3>    _assignRole(
  {_role:ACCESS_CONTROL_MANAGER_ROLE,
  _newAddress:msg.sender,
  _addRole:false}
);
    }
```

If `_initialAdminAddress == msg.sender`, the role is first assigned and then revoked from the same address. This leads to a situation where no account has the `ACCESS_CONTROL_MANAGER_ROLE`.

NOTE: The same issue happens in `AgoraStableSwapPair.initialize()` too.

[L-03] Incorrect zero address check only validates `token0` instead of `token1`

In the `computePairDeploymentAddress` function, the zero address check is applied only to `_token0`, which is the larger of the two sorted token addresses. However, since `_token1` is guaranteed to be the lower address in the sorted pair, the check should instead validate that `_token1 != address(0)`. Failing to do so may allow `address(0)` to silently pass as one of the tokens, potentially causing incorrect behavior or deployment issues.

```
function computePairDeploymentAddress(
        address _tokenA,
        address _tokenB
    ) public view returns (address _pairDeploymentAddress) {
        // Checks: tokens in swap should be different
        if (_tokenA == _tokenB) revert IdenticalAddresses();
        // Sort the tokens
        (address _token0, address _token1) = sortTokens(_tokenA, _tokenB);

        // Checks: None of the tokens should be zero address
@>      if (_token0 == address(0)) revert ZeroAddress();

        bytes32 _salt = _generateSalt(_token0, _token1);
        // `this` is the factory proxy
        bytes32 _guardedSalt = keccak256(abi.encodePacked(bytes32(uint256
          (uint160(address(this)))), _salt));
        _pairDeploymentAddress = ICreateX
          (CREATEX_DEPLOYER).computeCreate3Address({ salt: _guardedSalt });
    }
```

[L-04] Pools are not compatible with Uniswap V2 routers

While the interface of the `swap()` function matches the Uniswap V2 router, this router cannot be used with the protocol's pools. There are several reasons for this:

- Uniswap's router uses its own factory contract to find the pair.
- The pair address is calculated based on Uniswap's pool init code hash.
- For the calculation of amounts in and out the reserves in the pair are used.

[L-05] Unexpected behavior for fee on transfer tokens

The `swapExactTokensForTokens()` and `swapTokensForExactTokens()` functions will not work correctly with fee on transfer tokens, as in the calculations it is assumed that the amounts in and out are the same as the amounts sent and received, which is not the case with fee on transfer tokens.

When the fee on transfer token is the token in, the transaction will fail, and when the fee on transfer token is the token out, the user might receive fewer tokens than the amount they expected.

Consider adding specific functions to handle fees on transfer tokens if they are meant to be supported or documenting that the protocol does not support them.

[L-06] `setFeeBounds` change not applied, causing outdated purchase fees

The `setFeeBounds` function in **AgoraStableSwapPairConfiguration.sol** updates the allowable ranges (`minToken0PurchaseFee`, `maxToken0PurchaseFee`, etc.) for future fee settings but does **not** retroactively enforce these bounds on the **current** fee variables (`swapStorage.token0PurchaseFee`, `swapStorage.token1PurchaseFee`). As a result, if the admin initially configures a high fee (e.g., 2%), sets the current fee to this upper limit, and later tightens the bound to a lower value (e.g., 1%), the existing fee remains unchanged at 2%. This violates the intended invariant that `tokenPurchaseFee` should always lie within the configured bounds, potentially resulting in over-charging.

[L-07] `claimFor` allows attackers to steal rewards via flash swap

The protocol is designed to support interest-bearing tokens, some of which require explicit calls to claim rewards (e.g., via `claim()` or `harvest()`). However, the `Pair` contract does not include functionality to claim such rewards. Also when the reward platform has a `claimFor` functionality, an attacker can exploit it to steal the rewards: An attacker can initiate a `swap()` and within the `uniswapV2Call()` callback, they can call the token's `claimFor(address)` method to transfer reward tokens to the pair contract. Since the pair contract calculates token input amounts based on balance deltas, it incorrectly interprets the freshly claimed rewards as the attacker's input. The attacker then receives the output tokens without actually transferring any input tokens, effectively stealing the protocol's reward tokens.

```
function swap(
   uint256_amount0Out,
   uint256_amount1Out,
   address_to,
   bytesmemory_data
 ) public nonReentrant {
      --Snipped--

      if (_data.length > 0) {
@1>         IUniswapV2Callee(_to).uniswapV2Call({
                sender: msg.sender,
                amount0: _amount0Out,
                amount1: _amount1Out,
                data: _data
            });
        }
       --Snipped--

            uint256 _finalToken0Balance = IERC20(_swapStorage.token0).balanceOf
              ({ account: address(this) });
            uint256 _finalToken1Balance = IERC20(_swapStorage.token1).balanceOf
              ({ account: address(this) });


                         uint256 _previousToken0Balance = _swapStorage.reserve0 + _swa

                         uint256 _previousToken1Balance = _swapStorage.reserve1 + _swa

            // Calculate how many tokens were transferred
            _token0In = _finalToken0Balance >
              (_previousToken0Balance - _amount0Out)
 @2>            ? _finalToken0Balance - (_previousToken0Balance - _amount0Out)
                : 0;
            --Snipped--
  }
```

Recommendations:

- Integrate explicit reward claiming logic in the contract to handle reward tokens properly.
- Block external reward claims during sensitive operations such as `swap()` or restrict `claimFor` access to trusted roles or users.

[L-08] `calculatePrice` linear model causes yield token pricing errors

The `calculatePrice` function determines the current swap price using a linear interest model based on the base price, per-second interest rate, and time elapsed.

```
function calculatePrice(
        uint256 _priceLastUpdated,
        uint256 _timestamp,
        int256 _perSecondInterestRate,
        uint256 _basePrice
    ) public pure returns (uint256 _price) {
        // Calculate the time elapsed since the last price update
        uint256 timeElapsed = _timestamp - _priceLastUpdated;

        // Calculate the price
        _price = _perSecondInterestRate >= 0
            ? ((_basePrice * (PRICE_PRECISION + uint256
              (_perSecondInterestRate) * timeElapsed)) / PRICE_PRECISION)
            : ((_basePrice * (PRICE_PRECISION - (uint256
              (-_perSecondInterestRate) * timeElapsed))) / PRICE_PRECISION);
    }
```

However, this approach does not reflect the compounding nature of interest in yield-bearing tokens such as USTB. By dividing the annual interest rate by 365 and applying it linearly over time, the model overestimates the actual price growth, which should be exponential due to compounding effects.

This discrepancy results in incorrect pricing for tokens that rely on compound yield mechanics and needs for frequent price adjustment.

Recommendations: Update the pricing model to support compounding interest, for example by implementing an exponential growth formula using fixed-point math.

[L-09] `configureOraclePrice` can be exploited if price exceeds fee

The `configureOraclePrice` function is used to set a new base price and annualized interest rate for a trading pair:

```
function configureOraclePrice
     (uint256 _basePrice, int256 _annualizedInterestRate) public {
        --Snipped--
        _getPointerToStorage().swapStorage.priceLastUpdated =
          (block.timestamp).toUint40();
        // Effects: Convert yearly APR to per second APR
        _getPointerToStorage().swapStorage.perSecondInterestRate =
          (_annualizedInterestRate / 365 days).toInt72();
        // Effects: Set the price of the asset
        _getPointerToStorage().swapStorage.basePrice = _basePrice;

        emit ConfigureOraclePrice(_basePrice, _annualizedInterestRate);
    }
```

This base price is then used by the `calculatePrice` function to determine the effective swap price of tokens over time, factoring in interest rates and the elapsed time since the last update:

```
function calculatePrice(
        uint256 _priceLastUpdated,
        uint256 _timestamp,
        int256 _perSecondInterestRate,
        uint256 _basePrice
    ) public pure returns (uint256 _price) {
        // Calculate the time elapsed since the last price update
        uint256 timeElapsed = _timestamp - _priceLastUpdated;


        // Calculate the price
        _price = _perSecondInterestRate >= 0
            ? ((_basePrice * (PRICE_PRECISION + uint256
              (_perSecondInterestRate) * timeElapsed)) / PRICE_PRECISION)
            : ((_basePrice * (PRICE_PRECISION - (uint256
              (-_perSecondInterestRate) * timeElapsed))) / PRICE_PRECISION);
    }
```

An attacker can front-run the `configureOraclePrice` by buying the asset expected to increase in price, allowing the price update to take effect, and then back-run by selling at the new higher price. The exploit is profitable as long as the price change is greater than the total transaction and token swap fees. This vulnerability applies to both price increases and decreases.

Recommendations: To reduce the risk of exploitation, consider updating the base price gradually over multiple blocks instead of applying it immediately.

[L-10] `configureOraclePrice()` missing timestamp causes price errors

The `configureOraclePrice()` function in `AgoraStableSwapPairConfiguration` contract, sets the base price and annualized interest rate for a pair using `block.timestamp` as the time of update.

```
function configureOraclePrice
      (uint256 _basePrice, int256 _annualizedInterestRate) public {
        // Checks: Only the price setter can configure the price
        _requireSenderIsRole({ _role: PRICE_SETTER_ROLE });

        ConfigStorage memory _configStorage = _getPointerToStorage
          ().configStorage;

        // Checks: price is within bounds
        if
          (_basePrice < _configStorage.minBasePrice || _basePrice > _configStorage.max
            revert BasePriceOutOfBounds();
        }
        if (

                         _annualizedInterestRate < _configStorage.minAnnualizedInteres
            _annualizedInterestRate > _configStorage.maxAnnualizedInterestRate
        ) revert AnnualizedInterestRateOutOfBounds();

        // Effects: Set the time of the last price update
@>      _getPointerToStorage().swapStorage.priceLastUpdated =
  (block.timestamp).toUint40();
        // Effects: Convert yearly APR to per second APR
        _getPointerToStorage().swapStorage.perSecondInterestRate =
          (_annualizedInterestRate / 365 days).toInt72();
        // Effects: Set the price of the asset
        _getPointerToStorage().swapStorage.basePrice = _basePrice;

        // emit event
        emit ConfigureOraclePrice(_basePrice, _annualizedInterestRate);
    }
```

However, due to the non-deterministic nature of blockchain transaction processing, this approach introduces inaccuracies.

Transactions can be delayed significantly between signing and execution. As a result, the recorded timestamp may not reflect the intended time of configuration, which can lead to incorrect price computations.

Additionally, the function lacks a `deadline` mechanism. Without this, outdated transactions can be executed at a much later time, further compromising the accuracy and integrity of the pricing logic.

Recommendations: Add a `timestamp` parameter to allow the caller to specify the intended time for the base price update, along with a `deadline` parameter to ensure the transaction is only valid if mined before a specific time.

[L-11] `getAmountsIn` fails to validate sufficient reserves for token out fees

The `getAmountsIn` function only checks whether the pool reserves can satisfy the `_amountOut` value requested by the user. However, it does not account for the additional portion of the token that will be collected as a fee (`tokenFeesAccumulated`), which is also deducted from the reserves during a real swap.

```
function getAmountsIn(
      uint256_amountOut,
      address[]memory_path
    ) public view returns (uint256[] memory _amounts
        SwapStorage memory _swapStorage = _getPointerToStorage().swapStorage;
        uint256 _token0OverToken1Price = getPrice();

        // Checks: path length is 2 && path must contain token0 and token1 only
        requireValidPath(
          {_path:_path,
          _token0:_swapStorage.token0,
          _token1:_swapStorage.token1}
        );
@>        if
  (_path[1] == _swapStorage.token0 && _amountOut > _swapStorage.reserve0) revert Insuf
@>        if
  (_path[1] == _swapStorage.token1 && _amountOut > _swapStorage.reserve1) revert Insuf

        // Checks: amountOut is greater than 0
        if (_amountOut == 0) revert InsufficientOutputAmount();

        // instantiate return variables
        _amounts = new uint256[](2);
        // set the amountOut
        _amounts[1] = _amountOut;

        // ...
    }
```

This issue can result in wrong return values from `getAmountsIn`, and since this is a DEX, it is expected to cause unexpected reverts for swaps to integrators (like routers or aggregators).

Recommendations: Consider using the token amount fees that `AgoraStableSwapPairCore` functions return and adding them to the `amountOut`, in order to check if the reserves are sufficient.

[L-12] Token address validation missing allows fee drain via alternate routes

The `removeTokens` function never validates that the supplied `_tokenAddress` matches exactly `token0` or `token1`. In setups where a token can be interacted with through two addresses (for example, a proxy and a secondary entry-point contract), one of those addresses will not match the stored `token0`/`token1` values—even though they control the same underlying balances.

```
File: AgoraStableSwapPairConfiguration.sol
     function removeTokens
  (address _tokenAddress, uint256 _amount) external {
...
@>        if
  (_tokenAddress == _swapStorage.token0 && _amount > _token0Balance - _swapStorage.tok
              revert InsufficientTokens();
          }
@>        if
  (_tokenAddress == _swapStorage.token1 && _amount > _token1Balance - _swapStorage.tok
              revert InsufficientTokens();
          }
...
          // Interactions: transfer tokens from the pair to the token
// receiver
@>        IERC20(_tokenAddress).safeTransfer
  ({ to: _configStorage.tokenReceiverAddress, value: _amount });
```

Consequently, calls using the “alias” address bypass both fee-deduction checks and allow draining the full balance (including fees accumulated).

Recommendations: Enforce strict token-address equality:

```
if (_tokenAddress != _swapStorage.token0 &&
    _tokenAddress != _swapStorage.token1) {
    revert InvalidTokenAddress();
}
```

[L-13] Underflow in negative interest rate calculation leads to denial of service

The `calculatePrice` function attempts to apply a negative per-second interest rate by subtracting the accumulated rate adjustment from `PRICE_PRECISION`. However, when

```
File: AgoraStableSwapPairCore.sol
    function calculatePrice(
        uint256 _priceLastUpdated,
        uint256 _timestamp,
        int256 _perSecondInterestRate,
        uint256 _basePrice
    ) public pure returns (uint256 _price) {
        // Calculate the time elapsed since the last price update
        uint256 timeElapsed = _timestamp - _priceLastUpdated;

        // Calculate the price
        _price = _perSecondInterestRate >= 0
            ? ((_basePrice * (PRICE_PRECISION + uint256
  (_perSecondInterestRate) * timeElapsed)) / PRICE_PRECISION)
@>          : ((_basePrice * (PRICE_PRECISION - (uint256
  (-_perSecondInterestRate) * timeElapsed))) / PRICE_PRECISION);
    }
```

evaluates to a negative value, a uint256 underflow occurs and causes an immediate revert. This effectively becomes a denial-of-service vector: once `timeElapsed * |_perSecondInterestRate| > PRICE_PRECISION`, **every** call to `calculatePrice` will revert, blocking swaps, or any dependent logic. Consider the following scenario:

1. **Set parameters:**

   - `PRICE_PRECISION = 1e18`
   - `_perSecondInterestRate = -1e15` (–0.000001 per sec)
   - `_priceLastUpdated = 0`
   - `timeElapsed = 2000` seconds
   - `_basePrice = 1e18`

2. **Compute decay:**

   ```
   decay = uint256(-(-1e15)) * 2000 = 1e15 * 2000 = 2e18
   ```

3. **Attempt subtraction:**

   ```
   PRICE_PRECISION - decay = 1e18 - 2e18 → underflow → revert
   ```

4. **Result:** Any call to `calculatePrice()` reverts, breaking downstream logic.

Test:

```solidity
// SPDX-License-Identifier: ISC
pragma solidity ^0.8.28;

import "@openzeppelin/contracts/token/ERC20/IERC20.sol";
import "src/test/BaseTest.sol";

/* solhint-disable func-name-mixedcase */
contract TestOverFlow is BaseTest {
    /// FEATURE: Advanced fee collection scenarios

    address payable public alice = labelAndDeal("alice");
    address payable public bob = labelAndDeal("bob");

    address token0Address;
    address token1Address;

    IERC20 token0;
    IERC20 token1;

    function setUp() public {
        /// BACKGROUND:
        _defaultFactorySetup();
        pairAddress = factory.createPair(
            AgoraStableSwapFactory.CreatePairArgs({
                token0: Constants.Mainnet.USDC_ERC20,
                token1: Constants.Mainnet.AUSD_ERC20,
                minToken0PurchaseFee: 0,
                maxToken0PurchaseFee: 1e18,
                minToken1PurchaseFee: 0,
                maxToken1PurchaseFee: 1e18,
                token0Decimals: 6,
                token1Decimals: 6,
                token0PurchaseFee: 5e14,
                token1PurchaseFee: 1e14,
                minBasePrice: 1e18,
                maxBasePrice: 1e18,
                basePrice: 1e18,
                minAnnualizedInterestRate: -50e18,
                maxAnnualizedInterestRate: 1e18,
                annualizedInterestRate: -31e18
            })
        );
        pair = AgoraStableSwapPair(pairAddress);
        token0Address = pair.token0();
        token1Address = pair.token1();
        token0 = IERC20(token0Address);
        token1 = IERC20(token1Address);
        _seedErc20(
          {_tokenAddress:token0Address,
          _to:pairAddress,
          _amount:1e6*1e6}
        );
        _seedErc20(
          {_tokenAddress:token1Address,
          _to:pairAddress,
          _amount:1e6*1e6}
        );
        pair.sync();
        _seedErc20({ _tokenAddress: pair.token0(), _to: alice, _amount: 100 *
          (10 ** pair.token0Decimals()) });
        _seedErc20({ _tokenAddress: pair.token1(), _to: alice, _amount: 100 *
          (10 ** pair.token1Decimals()) });
        _setApprovedSwapperAsWhitelister({ _pair: pair, _newSwapper: alice });
        assertTrue(pair.hasRole(TOKEN_REMOVER_ROLE, tokenRemoverAddress));
        hoax(feeSetterAddress);
        pair.setTokenPurchaseFees(1e16, 1e16); // 1% fee for both tokens
        vm.startPrank(alice);
        uint256 _amount0In = 10 * (10 ** pair.token0Decimals());
        (uint256 _amount1Out,) = pair.getAmount1Out(
            _amount0In,
            pair.getPrice(),
            pair.token1PurchaseFee()
        );
        uint256 _amount1In = _amount1Out;
        (uint256 _amount0Out,) = pair.getAmount0Out(
            _amount1In,
            pair.getPrice(),
            pair.token0PurchaseFee()
        );
        token0.approve(address(pair), type(uint256).max);
        token1.approve(address(pair), type(uint256).max);
        IERC20(pair.token0()).transfer(address(pair), _amount0In);
        pair.swap(0, _amount1Out, alice, "");
        IERC20(pair.token1()).transfer(address(pair), _amount1In);
        pair.swap(_amount0Out, 0, alice, "");
        vm.stopPrank();
        address[] memory toApprove = new address[](1);
        toApprove[0] = address(this);
        hoax(whitelisterAddress);
        pair.setApprovedSwappers(toApprove, true);
        _seedErc20({ _tokenAddress: token0Address, _to: address
          (this), _amount: 1e18 * 1e6 });
        _seedErc20({ _tokenAddress: token1Address, _to: address
          (this), _amount: 1e18 * 1e6 });
    }

    function test_OverFlowRevertNegative() public {
        PairStateSnapshot memory _initialPairSnapshot = pairStateSnapshot
          (pairAddress);
        uint256 initialToken1Balance = token1.balanceOf(address(pair));
        uint256 currentToken1FeesAccumulated = pair.token1FeesAccumulated();
        //
        // 1. calls calculatePrice() which will overflow
        vm.expectRevert();
        pair.getPrice(block.timestamp + 20 days);
    }
}
```

Recommendations: Negative-rate decay must not be allowed to underflow the precision constant, and appropriate bounds checks or caps should be introduced. Additionally, since a negative price is invalid, it should revert and emit a specific `error()`, especially because there are public functions like `getPrice` that depend on this behavior.

[L-14] Using fixed pair prices allows arbitrage

Pair prices are fixed and adjusted with linear interest in the case of interest-bearing assets. As AUSD is meant to be pegged to USD, the price of the pair might not necessarily represent the actual value of the asset. While stablecoins tend to be valued at 1 USD, market fluctuations can lead to temporary deviations from this value. Additionally, interest-bearing assets such as USTB and USYC might have discrete price changes (e.g. once per day), while their price in the pair is adjusted linearly. Also, the protocol might not be able to reflect the changes in the interest rate of the asset in real time, as they require a manual update of the pair price.

Given that there is no slippage in the swap and there is no limit on the amount of tokens that can be swapped, users can potentially arbitrage the price difference between the pair and the actual market price of the asset, withdrawing all the liquidity of the overvalued asset and leaving the pool with a single token, causing a loss of funds for the protocol and potentially degrading the value of the stablecoin.

As the arbitration strategy is only profitable when the price divergence is greater than the swap fee, increasing the swap fee can mitigate the risk of arbitrage attacks. However, this has some drawbacks as 1) This will only reduce the chance of arbitrage attacks, but not eliminate them, and 2) It will deter users from using the protocol, as they will have to pay a higher fee for the swap.

Recommendations: Use external price oracles to determine the actual market price of the assets in the pair.

[L-15] Stale fees cause reserve miscalculation, enabling asset drain

The contract updates both the fee accumulators and reserves in the **memory** struct `_swapStorage` (code line 334-337), even though a malicious callback can mutate the on-chain fee accumulated state via `collectFees()` (code line 280). Because `_swapStorage` was loaded *before* the callback (code line 265), it remains stale, and line 334 blindly adds the new purchase fee onto an outdated value. Consequently, line 336 calculates `_swapStorage.reserve1` using a wrong fee offset, breaking the invariant and enabling theft.

```
File: AgoraStableSwapPairCore.sol
255:     /// @param _data The data to send to the callback
256:     function swap(
  uint256_amount0Out,
  uint256_amount1Out,
  address_to,
  bytesmemory_data
) public nonReentrant {
...
264:         // Cache information about the pair for gas savings
265:@>       SwapStorage memory _swapStorage = _getPointerToStorage
  ().swapStorage;
...
278:         // Execute the callback (if relevant)
279:         if (_data.length > 0) {
280:@>           IUniswapV2Callee(_to).uniswapV2Call({
281:                 sender: msg.sender,
282:                 amount0: _amount0Out,
283:                 amount1: _amount1Out,
284:                 data: _data
285:             });
286:         }
287:
288:         uint256 _token0In;
289:         uint256 _token1In;
290:         {
291:             // Create local scope
292:             // Take snapshot of balances
293:             uint256 _finalToken0Balance = IERC20
  (_swapStorage.token0).balanceOf({ account: address(this) });
294:             uint256 _finalToken1Balance = IERC20
  (_swapStorage.token1).balanceOf({ account: address(this) });
295:
296:@>
             uint256 _previousToken0Balance = _swapStorage.reserve0 + _swapStorage.tok
297:@>
             uint256 _previousToken1Balance = _swapStorage.reserve1 + _swapStorage.tok
...
333:             // Calculate new fees + reserves in memory struct


336:@>           _swapStorage.reserve0 =
  (_finalToken0Balance - _swapStorage.token0FeesAccumulated).toUint112();
337:@>           _swapStorage.reserve1 =
  (_finalToken1Balance - _swapStorage.token1FeesAccumulated).toUint112();
338:         }
```

The following test demonstrates how an attacker can leave a manipulated `reserve1` (with fewer tokens than should be). This happens because during the callback, the malicious user calls `collectFees()`, but that change is not reflected within the `swap()` function at `AgoraStableSwapPairCore#L334-337`, allowing the attacker to manipulate the reserve value. At the end the malicious fee collector is able to drain the reserves:

```
// SPDX-License-Identifier: ISC
pragma solidity ^0.8.28;

import "@openzeppelin/contracts/token/ERC20/IERC20.sol";
import "src/test/BaseTest.sol";

/* solhint-disable func-name-mixedcase */
contract TestSwapTokensMaliciousCallback is BaseTest {
    /// FEATURE: Advanced fee collection scenarios

    address payable public alice = labelAndDeal("alice");
    address payable public bob = labelAndDeal("bob");

    address token0Address;
    address token1Address;

    IERC20 token0;
    IERC20 token1;

    MaliciousFeeCollectorCallback maliciousContract;

    function setUp() public {
        /// BACKGROUND:
        _defaultFactorySetup();
        pairAddress = factory.createPair(
            AgoraStableSwapFactory.CreatePairArgs({
                token0: Constants.Mainnet.USDC_ERC20,
                token1: Constants.Mainnet.AUSD_ERC20,
                minToken0PurchaseFee: 0,
                maxToken0PurchaseFee: 1e18,
                minToken1PurchaseFee: 0,
                maxToken1PurchaseFee: 1e18,
                token0Decimals: 6,
                token1Decimals: 6,
                token0PurchaseFee: 5e14,
                token1PurchaseFee: 1e14,
                minBasePrice: 1e18,
                maxBasePrice: 1e18,
                basePrice: 1e18,
                minAnnualizedInterestRate: 0,
                maxAnnualizedInterestRate: 1e18,
                annualizedInterestRate: 0
            })
        );
        pair = AgoraStableSwapPair(pairAddress);
        token0Address = pair.token0();
        token1Address = pair.token1();
        token0 = IERC20(token0Address);
        token1 = IERC20(token1Address);
        _seedErc20(
          {_tokenAddress:token0Address,
          _to:pairAddress,
          _amount:1e6*1e6}
        );
        _seedErc20(
          {_tokenAddress:token1Address,
          _to:pairAddress,
          _amount:1e6*1e6}
        );
        pair.sync();
        //
        // 1. Generate some swaps to accumulate fees
        _seedErc20({ _tokenAddress: pair.token0(), _to: alice, _amount: 100 *
          (10 ** pair.token0Decimals()) });
        _seedErc20({ _tokenAddress: pair.token1(), _to: alice, _amount: 100 *
          (10 ** pair.token1Decimals()) });
        _setApprovedSwapperAsWhitelister({ _pair: pair, _newSwapper: alice });
        assertTrue(pair.hasRole(TOKEN_REMOVER_ROLE, tokenRemoverAddress));
        hoax(feeSetterAddress);
        pair.setTokenPurchaseFees(1e16, 1e16); // 1% fee for both tokens

        vm.startPrank(alice);
        uint256 _amount0In = 10 * (10 ** pair.token0Decimals());
        (uint256 _amount1Out,) = pair.getAmount1Out(
            _amount0In,
            pair.getPrice(),
            pair.token1PurchaseFee()
        );
        uint256 _amount1In = _amount1Out;
        (uint256 _amount0Out,) = pair.getAmount0Out(
            _amount1In,
            pair.getPrice(),
            pair.token0PurchaseFee()
        );
        token0.approve(address(pair), type(uint256).max);
        token1.approve(address(pair), type(uint256).max);
        IERC20(pair.token0()).transfer(address(pair), _amount0In);
        pair.swap(0, _amount1Out, alice, "");
        IERC20(pair.token1()).transfer(address(pair), _amount1In);
        pair.swap(_amount0Out, 0, alice, "");
        vm.stopPrank();
        //
        // 2. Create a malicious contract that will receive the swap
        //(it is used by malicious fees collector)
        // The malicious collector has permission to call collectFees()
        maliciousContract = new MaliciousFeeCollectorCallback(address(pair));
        hoax(adminAddress);
        pair.assignRole(TOKEN_REMOVER_ROLE, address(maliciousContract), true);
        //
        // 3. Set up this contract as approved swapper
        address[] memory toApprove = new address[](1);
        toApprove[0] = address(this);
        hoax(whitelisterAddress);
        pair.setApprovedSwappers(toApprove, true);
        _seedErc20({ _tokenAddress: token0Address, _to: address
          (this), _amount: 1e18 * 1e6 });
        _seedErc20({ _tokenAddress: token1Address, _to: address
          (this), _amount: 1e18 * 1e6 });
    }

    function test_MaliciousCallbackCanManipulateSwap() public {
        PairStateSnapshot memory _initialPairSnapshot = pairStateSnapshot
          (pairAddress);
        uint256 initialToken1Balance = token1.balanceOf(address(pair));
        uint256 currentToken1FeesAccumulated = pair.token1FeesAccumulated();
        //
        // 5. Perform a swap that will trigger the malicious callback
        //(malicious fee collector)
        uint256 _amount1Out = 10 * (10 ** pair.token1Decimals());
        //
        (
          uint256_amount0In,
          uint256_token1PurchaseFeeAmount
        ) = pair.getAmount0In(
            _amount1Out,
            pair.getPrice(),
            pair.token1PurchaseFee()
        );
        IERC20(pair.token0()).transfer(address(pair), _amount0In);
        bytes memory data = abi.encode("malicious");
        pair.swap(0, _amount1Out, address(maliciousContract), data);
        //
        // 6. The reserve1 is manipulated by the malicious callback. 100000 are
        // removed from reserve1 which is incorrect
        assertEq(pair.reserve1(
            pair.reserve1

          ), initialToken1Balance - _amount1Out - currentToken1FeesAccumulated - _token1
          //

          vm.startPrank(address(maliciousContract));
          pair.collectFees(pair.token1(), _token1PurchaseFeeAmount + 100000);
      }
  }


  contract MaliciousFeeCollectorCallback {
      address public pair;

      constructor(address _pair) {
          pair = _pair;
      }

      // This function will be called during the swap
      function uniswapV2Call(
          address sender,
          uint256 amount0,
          uint256 amount1,
          bytes calldata data
      ) external {
          // Try to manipulate the state by calling removeTokens or collectFees
          // This will affect the swap calculations
          // AgoraStableSwapPair(pair).removeTokens(AgoraStableSwapPair
          //(pair).token1(), 1337);
          AgoraStableSwapPair(pair).collectFees(AgoraStableSwapPair(pair).token1
            (), 100000);
      }
  }
```

Recommendations: Reload `_swapStorage` in `swap()` if `uniswapV2Call` is executed.