// Smart Contract Security Assessment 10.22.2024 - 11.01.2024

BitFluxFi - Stable AMM

CoreDAO

HALBORN

BitFluxFi - Stable AMM - CoreDAO

Prepared by: HALBORN

Last Updated 11/20/2024

Date of Engagement by: October 22nd, 2024 - November 1st, 2024

Summary

100% OF ALL REPORTED FINDINGS HAVE BEEN ADDRESSED

ALL FINDINGS	CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL
7	0	0	1	4	2

TABLE OF CONTENTS

1. Introduction

CoreDAO engaged Halborn to conduct a security assessment on their smart contracts beginning on October 22nd and ending on November 1st, 2024. The security assessment was scoped to the smart contracts provided to the Halborn team.

Commit hashes and further details can be found in the Scope section of this report.

2. Assessment Summary

The team at Halborn assigned a full-time security engineer to assess the security of the smart contracts. The security engineer is a blockchain and smart-contract security expert with advanced penetration testing, smart-contract hacking, and deep knowledge of multiple blockchain protocols.

The purpose of this assessment is to:

- Ensure that smart contract functions operate as intended.
- Identify potential security issues with the smart contracts.

In summary, Halborn identified some improvements to reduce the likelihood and impact of risks, which were accepted and acknowledged by the BitFlux team. The main ones were the following:

- Replace all instances of transfer and transferFrom with safeTransfer and safeTransferFrom from OpenZeppelin's SafeERC20 library.
- Replace the use of approve with safeIncreaseAllowance to ensure consistent allowance handling.
- Prevent unnecessary overwriting and allow users to contribute liquidity according to their preferences.
- Review the differences between the origin implementation and the current one, and make sure that all deviations are design choices.
- Check for duplicate token addresses in the input arrays within liquidity operations.

3. Test Approach And Methodology

Halborn performed a combination of manual review of the code and automated security testing to balance efficiency, timeliness, practicality, and accuracy in regard to the scope of the smart contract assessment. While manual testing is recommended to uncover flaws in logic, process, and implementation; automated testing techniques help enhance coverage of smart contracts and can quickly identify items that do not follow security best practices. The following phases and associated tools were used throughout the term of the assessment:

- Research into the architecture, purpose, and use of the platform.
- Smart contract manual code review and walkthrough to identify any logic issue.
- Thorough assessment of safety and usage of critical Solidity variables and functions in scope that could lead to arithmetic related vulnerabilities.
- Manual testing by custom scripts.
- Graphing out functionality and contract logic/connectivity/functions (solgraph).
- Static Analysis of security for scoped contract, and imported functions. (Slither).
- Local or public testnet deployment (Foundry, Remix IDE).

4. RISK METHODOLOGY

Every vulnerability and issue observed by Halborn is ranked based on **two sets** of **Metrics** and a **Severity Coefficient**. This system is inspired by the industry standard Common Vulnerability Scoring System.

The two **Metric sets** are: **Exploitability** and **Impact**. **Exploitability** captures the ease and technical means by which vulnerabilities can be exploited and **Impact** describes the consequences of a successful exploit.

The **Severity Coefficients** is designed to further refine the accuracy of the ranking with two factors: **Reversibility** and **Scope**. These capture the impact of the vulnerability on the environment as well as the number of users and smart contracts affected.

The final score is a value between 0-10 rounded up to 1 decimal place and 10 corresponding to the highest security risk. This provides an objective and accurate rating of the severity of security vulnerabilities in smart contracts.

The system is designed to assist in identifying and prioritizing vulnerabilities based on their level of risk to address the most critical issues in a timely manner.

4.1 EXPLOITABILITY

ATTACK ORIGIN (AO):

Captures whether the attack requires compromising a specific account.

ATTACK COST (AC):

Captures the cost of exploiting the vulnerability incurred by the attacker relative to sending a single transaction on the relevant blockchain. Includes but is not limited to financial and computational cost.

ATTACK COMPLEXITY (AX):

Describes the conditions beyond the attacker's control that must exist in order to exploit the vulnerability. Includes but is not limited to macro situation, available third-party liquidity and regulatory challenges.

METRICS:

EXPLOITABILIY METRIC (M_E)	METRIC VALUE	NUMERICAL VALUE
Attack Origin (AO)	Arbitrary (AO:A) Specific (AO:S)	1 0.2

EXPLOITABILIY METRIC (M_E)	METRIC VALUE	NUMERICAL VALUE
Attack Cost (AC)	Low (AC:L) Medium (AC:M) High (AC:H)	1 0.67 0.33
Attack Complexity (AX)	Low (AX:L) Medium (AX:M) High (AX:H)	1 0.67 0.33

Exploitability E is calculated using the following formula:

$$E = \prod m_e$$

4.2 IMPACT

CONFIDENTIALITY (C):

Measures the impact to the confidentiality of the information resources managed by the contract due to a successfully exploited vulnerability. Confidentiality refers to limiting access to authorized users only.

INTEGRITY (I):

Measures the impact to integrity of a successfully exploited vulnerability. Integrity refers to the trustworthiness and veracity of data stored and/or processed on-chain. Integrity impact directly affecting Deposit or Yield records is excluded.

AVAILABILITY (A):

Measures the impact to the availability of the impacted component resulting from a successfully exploited vulnerability. This metric refers to smart contract features and functionality, not state. Availability impact directly affecting Deposit or Yield is excluded.

DEPOSIT (D):

Measures the impact to the deposits made to the contract by either users or owners.

YIELD (Y):

Measures the impact to the yield generated by the contract for either users or owners.

METRICS:

IMPACT METRIC (M_I)	METRIC VALUE	NUMERICAL VALUE
Confidentiality (C)	None (I:N) Low (I:L) Medium (I:M) High (I:H) Critical (I:C)	0 0.25 0.5 0.75 1
Integrity (I)	None (I:N) Low (I:L) Medium (I:M) High (I:H) Critical (I:C)	0 0.25 0.5 0.75 1
Availability (A)	None (A:N) Low (A:L) Medium (A:M) High (A:H) Critical (A:C)	0 0.25 0.5 0.75 1
Deposit (D)	None (D:N) Low (D:L) Medium (D:M) High (D:H) Critical (D:C)	0 0.25 0.5 0.75 1
Yield (Y)	None (Y:N) Low (Y:L) Medium (Y:M) High (Y:H) Critical (Y:C)	0 0.25 0.5 0.75 1

Impact I is calculated using the following formula:

$$I = max(m_I) + \frac{\sum m_I - max(m_I)}{4}$$

4.3 SEVERITY COEFFICIENT

REVERSIBILITY (R):

Describes the share of the exploited vulnerability effects that can be reversed. For upgradeable contracts, assume the contract private key is available.

SCOPE (S):

Captures whether a vulnerability in one vulnerable contract impacts resources in other contracts.

METRICS:

SEVERITY COEFFICIENT (C)	COEFFICIENT VALUE	NUMERICAL VALUE
Reversibility (r)	None (R:N) Partial (R:P) Full (R:F)	1 0.5 0.25
Scope (s)	Changed (S:C) Unchanged (S:U)	1.25 1

Severity Coefficient C is obtained by the following product:

$$C = rs$$

The Vulnerability Severity Score S is obtained by:

$$S = min(10, EIC * 10)$$

The score is rounded up to 1 decimal places.

SEVERITY	SCORE VALUE RANGE
Critical	9 - 10
High	7 - 8.9
Medium	4.5 - 6.9
Low	2 - 4.4

SEVERITY	SCORE VALUE RANGE
Informational	0 - 1.9

5. SCOPE

FILES AND REPOSITORY

(a) Repository: contracts

(b) Assessed Commit ID: 3e1cf3e

(c) Items in scope:

- stable-amm/AmplificationUtils.sol
- stable-amm/LPToken.sol
- stable-amm/MathUtils.sol
- stable-amm/OwnableUpgradeable.sol
- stable-amm/Router.sol
- stable-amm/Swap.sol
- stable-amm/SwapDeployer.sol
- stable-amm/SwapFlashLoan.sol
- stable-amm/SwapUtils.sol
- stable-amm/helpers/FlashLoanBorrowerExample.sol
- stable-amm/helpers/GenericERC20.sol
- stable-amm/interfaces/IERC20.sol
- stable-amm/interfaces/IERC20PermitUpgradeable.sol
- stable-amm/interfaces/IERC20Upgradeable.sol
- stable-amm/interfaces/IFlashLoanReceiver.sol
- stable-amm/interfaces/ISwap.sol
- stable-amm/interfaces/ISwapFlashLoan.sol
- stable-amm/libraries/Address.sol
- stable-amm/libraries/Clones.sol
- stable-amm/libraries/ContextUpgradeable.sol
- stable-amm/libraries/CountersUpgradeable.sol
- stable-amm/libraries/ECDSAUpgradeable.sol
- stable-amm/libraries/EIP712Upgradeable.sol
- stable-amm/libraries/ERC20BurnableUpgradeable.sol
- stable-amm/libraries/ERC20PermitUpgradeable.sol
- stable-amm/libraries/ERC20Upgradeable.sol
- stable-amm/libraries/OwnerPausableUpgradeable.sol
- stable-amm/libraries/PausableUpgradeable.sol
- stable-amm/libraries/ReentrancyGuardUpgradeable.sol
- stable-amm/libraries/SafeERC20.sol
- stable-amm/libraries/SafeMath.sol
- stable-amm/libraries/SafeMathUpgradeable.sol
- stable-amm/proxy/Initializable.sol
- stable-amm/test/TestSwapReturnValues.sol

- stable-amm/utils/AddressUpgradeable.sol

Out-of-Scope: Third party dependencies and economic attacks.

Out-of-Scope: New features/implementations after the remediation commit IDs.

6. ASSESSMENT SUMMARY & FINDINGS OVERVIEW

CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL
0	0	1	4	2

SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
USE OF STANDARD TRANSFER METHODS MAY LEAD TO UNDETECTED FAILURES	MEDIUM	RISK ACCEPTED - 11/20/2024
RACE CONDITION IN ALLOWANCE UPDATES WITH APPROVE	LOW	RISK ACCEPTED - 11/20/2024
POTENTIAL USER RESTRICTIONS IN LIQUIDITY CONTRIBUTIONS	LOW	RISK ACCEPTED - 11/20/2024
NON COMPLIANCE WITH CURVE IMPLEMENTATION	LOW	NOT APPLICABLE
POTENTIAL HANDLING ERRORS DUE TO DUPLICATE TOKEN ENTRIES IN LIQUIDITY FUNCTIONS	LOW	RISK ACCEPTED - 11/20/2024
MISSING PROTECTION AGAINST POTENTIAL REENTRANCY ATTACKS	INFORMATIONAL	ACKNOWLEDGED - 11/20/2024

SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
SUBOPTIMAL GAS USAGE DUE TO POST-INCREMENT IN LOOPS	INFORMATIONAL	ACKNOWLEDGED - 11/20/2024

7. FINDINGS & TECH DETAILS

7.1 USE OF STANDARD TRANSFER METHODS MAY LEAD TO UNDETECTED FAILURES

// MEDIUM

Description

The functions `convert`, `addLiquidity`, `removeLiquidity`, and `removeBaseLiquidityOneToken` in the **Router** contract use `transfer` and `transferFrom` methods for token transfers. Unlike `safeTransfer` and `safeTransferFrom` from OpenZeppelin's SafeERC20 library, the standard `transfer` and `transferFrom` methods do not handle cases where tokens do not return a value or return **false** upon failure. This could lead to undetected transaction failures and unexpected behavior if tokens do not adhere strictly to the ERC-20 standard, potentially resulting in failed transfers without reverts. The affected functions are the following:

- `Router.convert`
- `Router.addLiquidity`
- `Router.removeLiquidity`
- `Router.removeBaseLiquidityOneToken`

BVSS

AO:A/AC:L/AX:M/R:N/S:U/C:N/A:N/I:N/D:H/Y:N (5.0)

Recommendation

Replace all instances of `transfer` and `transferFrom` with `safeTransfer` and `safeTransferFrom` from OpenZeppelin's SafeERC20 library. This ensures that all token transfers are properly checked and revert in case of failure, improving the security and reliability of the contract.

Remediation

RISK ACCEPTED: The **BitFlux team** accepted this risk of this finding and stated the following:
Router contract already utilizes OpenZeppelin's SafeERC20 library in several parts of the contract. For example, functions such as swapFromBase, swapToBase, and removeLiquidity already use safeTransferFrom and safeTransfer to ensure secure token transfers. This is to make sure that transfers either succeed or revert, preventing issues with tokens that do not return a boolean on success.
There are specific cases where functions make direct calls to transfer and transferFrom. These are mainly used when interacting with well-known tokens, such as LPs, which strictly follows ERC-20 standards.

References

contracts/stable-amm/Router.sol#L31, L46, L102, L116, L154

7.2 RACE CONDITION IN ALLOWANCE UPDATES WITH APPROVE

// LOW

Description

The `removeBaseLiquidityOneToken` function in the **Router** contract uses the `approve` method to set allowances. Unlike other functions such as `removeLiquidity`, which use `safeIncreaseAllowance`, this inconsistency can introduce a potential approval race condition risk. The `approve` function can be exploited if a spender front-runs the transaction and uses the current allowance before it is updated. This could lead to unintended token transfers, posing a security risk if not handled properly.

Code Location

Use of the `approve` method to set allowances:

```
143     function removeBaseLiquidityOneToken(
144         ISwap pool,
145         ISwap basePool,
146         uint256 _token_amount,
147         uint8 i,
148         uint256 _min_amount,
149         uint256 deadline
150     ) external returns (uint256) {
151         IERC20 token = pool.getLpToken();
152         IERC20 baseToken = basePool.getLpToken();
153         uint8 baseTokenIndex = pool.getTokenIndex(address(baseToken));
154         token.transferFrom(msg.sender, address(this), _token_amount);
155         token.approve(address(pool), _token_amount);
156         pool.removeLiquidityOneToken(_token_amount, baseTokenIndex, 0, de
157         uint256 _base_amount = baseToken.balanceOf(address(this));
158         baseToken.approve(address(basePool), _base_amount);
159         basePool.removeLiquidityOneToken(_base_amount, i, _min_amount, de
160         IERC20 coin = basePool.getToken(i);
161         uint256 coin_amount = coin.balanceOf(address(this));
162         coin.safeTransfer(msg.sender, coin_amount);
163         return coin_amount;
164     }
```

BVSS

AO:A/AC:L/AX:M/R:N/S:U/C:N/A:N/I:M/D:M/Y:N (4.2)

Recommendation

Replace the use of `approve` with `safeIncreaseAllowance` to align with best practices and ensure consistent allowance handling.

Remediation

RISK ACCEPTED: The **BitFlux team** accepted this risk of this finding and stated the following:
We acknowledge that the approve method can introduce a race condition if a spender front-runs the transaction and uses the current allowance before it is updated. However, in our contract, this risk is mitigated by the specific context in which approve is used. The approve function is only called within tightly controlled internal logic, where the sequence of operations ensures that no external actor can exploit the allowance before it is consumed. Additionally, we use approve only with tokens that strictly follow the ERC-20 standard, which further reduces the likelihood of an issue.

7.3 POTENTIAL USER RESTRICTIONS IN LIQUIDITY CONTRIBUTIONS

// LOW

Description

The `addLiquidity` function in the **Router** contract has a potential issue where `meta_amounts[i]` is overwritten by `base_lp_received` when `coin` matches `base_lp`. This leads to two scenarios:

1. If the initial `meta_amounts[i]` is lower than `base_lp_received`, the user is compelled to use the higher `base_lp_received` value, potentially depositing more tokens than intended.
2. If the initial `meta_amounts[i]` is greater than `base_lp_received`, the user can only deposit up to `base_lp_received`, restricting the ability to contribute the desired amount.

This behavior reduces flexibility and can result in unexpected outcomes for users who intend to provide a specific amount of liquidity.

Code Location

`meta_amounts[i]` is overwritten by `base_lp_received` when `coin` matches `base_lp`:

```
50      function addLiquidity(
51          ISwap pool,
52          ISwap basePool,
53          uint256[] memory meta_amounts,
54          uint256[] memory base_amounts,
55          uint256 minToMint,
56          uint256 deadline
57      ) external returns (uint256) {
58          IERC20 token = IERC20(pool.getLpToken());
59          IERC20 base_lp = IERC20(basePool.getLpToken());
60          require(base_amounts.length == basePool.getNumberOfTokens(), "inv
61          require(meta_amounts.length == pool.getNumberOfTokens(), "invalid
62          bool deposit_base = false;
63          for (uint8 i = 0; i < base_amounts.length; i++) {
64              uint256 amount = base_amounts[i];
65              if (amount > 0) {
66                  deposit_base = true;
67                  IERC20 coin = basePool.getToken(i);
68                  uint256 transferred = transferIn(coin, msg.sender, amount
69                  coin.safeIncreaseAllowance(address(basePool), transferred
70                  base_amounts[i] = transferred;
```

```
            }
        }

        uint256 base_lp_received;
        if (deposit_base) {
            base_lp_received = basePool.addLiquidity(base_amounts, 0, dea
        }

        for (uint8 i = 0; i < meta_amounts.length; i++) {
            IERC20 coin = pool.getToken(i);

            uint256 transferred;
            if (address(coin) == address(base_lp)) {
                transferred = base_lp_received;
            } else if (meta_amounts[i] > 0) {
                transferred = transferIn(coin, msg.sender, meta_amounts[i
            }

            meta_amounts[i] = transferred;
            if (transferred > 0) {
                coin.safeIncreaseAllowance(address(pool), transferred);
            }
        }

        uint256 base_lp_prior = base_lp.balanceOf(address(this));
        pool.addLiquidity(meta_amounts, minToMint, deadline);
        if (deposit_base) {
            require((base_lp.balanceOf(address(this)) + base_lp_received)
        }

        uint256 lpAmount = token.balanceOf(address(this));
        token.transfer(msg.sender, lpAmount);
        return lpAmount;
    }
```

BVSS

AO:A/AC:L/AX:L/R:P/S:U/C:N/A:L/I:N/D:M/Y:N (2.8)

Recommendation

Adjust the logic to ensure that the value of `meta_amounts[i]` aligns with user input while maintaining proper handling of `base_lp` tokens. This change should prevent unnecessary overwriting and allow users to contribute liquidity according to their preferences.

Remediation

RISK ACCEPTED: The **BitFlux team** accepted this risk of this finding and stated the following:
We acknowledge that there is potential for user-specified values in meta_amounts[i] to be overwritten by base_lp_received. However, this behavior is intentional and designed to ensure that liquidity contributions are handled efficiently. In cases where users provide liquidity with both base LP tokens and meta tokens, it is necessary to adjust meta_amounts[i] to reflect the actual amount received from the base pool. This is to make sure accurate accounting and prevents discrepancies between user expectations and actual liquidity added.

7.4 NON COMPLIANCE WITH CURVE IMPLEMENTATION

// LOW

Description

Context

The `stable-amm` is an implementation of Curve's stable swap, and the `Halborn` team verified the proper implementation by inspecting the differences with the `SwapTemplateBase.vy` contract from the `bObbf77` commit.

Inconsistencies were identified in:

- `calculateTokenAmount` from `SwapUtils.sol`
- `getAdminbalance` from `Swap.sol`
- `stopRampA` from `AmplificationUtils.sol`
- `killMe` feature

Calculate Token Amount

It was found that the `calculateTokenAmount` of the `SwapUtils.sol` library returns a positive amount when the total supply of the liquidity pool is zero, while the original Curve implementation simply returns a zero amount. The affected function is only for view purposes so it does not affect the pool state, but could cause unplanned behaviour in external contracts.

- The `Curve` implementation in `SwapTemplateBase.vy`:

```
267 def calc_token_amount(_amounts: uint256[N_COINS], _is_deposit: bool) -> u
268     """
269     @notice Calculate addition or reduction in token supply from a deposi
270     @dev This calculation accounts for slippage, but not fees.
271          Needed to prevent front-running, not for precise calculations!
272     @param _amounts Amount of each coin being deposited
273     @param _is_deposit set True for deposits, False for withdrawals
274     @return Expected amount of LP tokens received
275     """
276     amp: uint256 = self._A()
277     balances: uint256[N_COINS] = self.balances
278     D0: uint256 = self._get_D_mem(balances, amp)
279     for i in range(N_COINS):
280         if _is_deposit:
281
```

```
            balances[i] += _amounts[i]
        else:
            balances[i] -= _amounts[i]
    D1: uint256 = self._get_D_mem(balances, amp)
    token_amount: uint256 = CurveToken(self.lp_token).totalSupply()
    diff: uint256 = 0
    if _is_deposit:
        diff = D1 - D0
    else:
        diff = D0 - D1
    return diff * token_amount / D0
```

- The `Bitflux` implementation in `SwapUtils.sol` the new stable pool invariant `d1` is returned:

```
function calculateTokenAmount(
    Swap storage self,
    uint256[] calldata amounts,
    bool deposit
) external view returns (uint256) {
    uint256 a = _getAPrecise(self);
    uint256[] memory balances = self.balances;
    uint256[] memory multipliers = self.tokenPrecisionMultipliers;

    uint256 d0 = getD(_xp(balances, multipliers), a);
    for (uint256 i = 0; i < balances.length; i++) {
        if (deposit) {
            balances[i] = balances[i].add(amounts[i]);
        } else {
            balances[i] = balances[i].sub(
                amounts[i],
                "Cannot withdraw more than available"
            );
        }
    }

    uint256 d1 = getD(_xp(balances, multipliers), a);
    uint256 totalSupply = self.lpToken.totalSupply();

    if (totalSupply == 0) {
        return d1; // first depositor take it all
    }

```

```
627     if (deposit) {
628         return d1.sub(d0).mul(totalSupply).div(d0);
629     } else {
630         return d0.sub(d1).mul(totalSupply).div(d0);
631     }
632 }
```

Get Admin Balance

It was found that the `getAdminbalance` from the `Swap.sol` contract was not following the same logic than the admin_balances counterpart in the Curve implementation.

- The `Curve` implementation in `SwapTemplateBase.vy`:

```
849 def admin_balances(i: uint256) -> uint256:
850     return ERC20(self.coins[i]).balanceOf(self) - self.balances[i]
```

- The `Bitflux` implementation in `Swap.sol`:

```
640 function getAdminBalances() external view returns (uint256[] memory admin
641     uint256 length = swapStorage.pooledTokens.length;
642     adminBalances = new uint256[](length);
643     for (uint256 i = 0; i < length; i++) {
644         adminBalances[i] = swapStorage.getAdminBalance(i);
645     }
646 }
```

Stop Ramp A

It was found that the `stopRampA` of the `AmplificationUtils.sol` library included an additional check, in comparison to the original implementation. The `stopRampA` function stops the transition of the amplifier factor of the stable pool, during its transition. That additional check only makes sure that the function cannot be called outside a transition, which has essentially no impact.

- The `Curve` implementation in `SwapTemplateBase.vy`:

```
765 def stop_ramp_A():
766     assert msg.sender == self.owner  # dev: only owner
767
```

```
768
769         current_A: uint256 = self._A()
770         self.initial_A = current_A
771         self.future_A = current_A
772         self.initial_A_time = block.timestamp
773         self.future_A_time = block.timestamp
774         # now (block.timestamp < t1) is always False, so we return saved A
775
            log StopRampA(current_A, block.timestamp)
```

- The `Bitflux` implementation in `AmplificationUtils.sol` (note that the `onlyOwner` modifier is present upstream):

```
149 function stopRampA(SwapUtils.Swap storage self) external {
150     require(self.futureATime > block.timestamp, "Ramp is already stopped"
151
152     uint256 currentA = _getAPrecise(self);
153     self.initialA = currentA;
154     self.futureA = currentA;
155     self.initialATime = block.timestamp;
156     self.futureATime = block.timestamp;
157
158     emit StopRampA(currentA, block.timestamp);
159 }
```

Kill Feature

The original Curve contract has a `killMe` feature that allows an owner to shut down a contract, also able to revive it later. It was found that the current implementation does not support this feature. An external contract may assume this feature to be implemented an malfunction as it is not.

- The `kill_me` definition in `SwapTemplateBase.vy`:

```
117 is_killed: bool
118 kill_deadline: uint256
119 KILL_DEADLINE_DT: constant(uint256) = 2 * 30 * 86400
```

- The `kill_me` toggles in `SwapTemplateBase.vy`:

```
@external
def kill_me():
    assert msg.sender == self.owner  # dev: only owner
    assert self.kill_deadline > block.timestamp  # dev: deadline has pass
    self.is_killed = True


@external
def unkill_me():
    assert msg.sender == self.owner  # dev: only owner
    self.is_killed = False
```

- Example usage of the `kill_me` feature in the `add_liquidity` function of `SwapTemplateBase.vy`:

```
@external
@nonreentrant('lock')
def add_liquidity(_amounts: uint256[N_COINS], _min_mint_amount: uint256)
    """
    @notice Deposit coins into the pool
    @param _amounts List of amounts of coins to deposit
    @param _min_mint_amount Minimum amount of LP tokens to mint from the
    @return Amount of LP tokens received by depositing
    """
    assert not self.is_killed  # dev: is killed

    amp: uint256 = self._A()
    old_balances: uint256[N_COINS] = self.balances
```

BVSS

AO:A/AC:L/AX:M/R:N/S:C/C:N/A:N/I:N/D:L/Y:N (2.1)

Recommendation

It is recommended to review the differences between the origin implementation and the current one, and make sure that all deviations are design choices.

Remediation

NOT APPLICABLE: The **BitFlux team** highlighted that the issue is not applicable and mentioned the following: *We recognize that our implementation differs from Curve's original design in certain areas, such as calculateTokenAmount, getAdminBalance, and stopRampA. These differences were intentional design*

choices made to optimize performance and improve usability for our specific use case. For example, returning a positive amount when the total supply is zero in calculateTokenAmount simplifies initial liquidity provision without affecting pool state or security. Similarly, our additional check in stopRampA ensures that transitions are handled more safely without introducing any negative side effects.

7.5 POTENTIAL HANDLING ERRORS DUE TO DUPLICATE TOKEN ENTRIES IN LIQUIDITY FUNCTIONS

// LOW

Description

In the `addLiquidity` and `removeLiquidity` functions of the **Router** contract, user-passed arrays (`meta_amounts`, `base_amounts`, etc.) are used to perform token transfers and allowances. If these arrays contain duplicate token addresses, the functions may perform redundant operations on the same token, leading to potential mismanagement of allowances or incorrect token balances. This could cause inconsistencies in the expected behavior of these operations, making it crucial to ensure that each token is processed only once.

BVSS

AO:A/AC:L/AX:M/R:N/S:U/C:N/A:L/I:L/D:N/Y:N (2.1)

Recommendation

Implement a validation mechanism to check for duplicate token addresses in the input arrays within `addLiquidity` and `removeLiquidity` functions. This will ensure that operations are executed accurately, with each token being handled only once, maintaining consistency and preventing any potential mismanagement of token balances.

Remediation

RISK ACCEPTED: The **BitFlux team** accepted this risk of this finding and stated the following:
We acknowledge that allowing duplicate token entries in liquidity functions could theoretically lead to handling errors. However, our contract's internal logic ensures that duplicate entries are processed correctly without causing unexpected behavior. This is a design decision to handle each token individually, even if duplicates exist in the input array, we make sure that all tokens are accounted for properly during liquidity operations.

7.6 MISSING PROTECTION AGAINST POTENTIAL REENTRANCY ATTACKS

// INFORMATIONAL

Description

The **Router** contract includes functions such as `convert`, `addLiquidity`, `removeLiquidity`, and `removeBaseLiquidityOneToken` that handle ERC20 token transfers. Although the use of `transfer` and `transferFrom` functions is common, it is important to consider the risk of reentrancy when these functions make multiple external calls. Even though standard ERC20 tokens are assumed to be secure, there could still be potential vulnerabilities if reentrancy protection is not implemented. This observation is informational, as no immediate issues are present given that only valid ERC20 tokens are used. However, it highlights the importance of adopting protective measures as a best practice.

BVSS

AO:A/AC:M/AX:H/R:N/S:U/C:N/A:N/I:N/D:M/Y:N (1.1)

Recommendation

It is recommended to implement **ReentrancyGuard** from OpenZeppelin's library by adding the `nonReentrant` modifier to the mentioned function to prevent recursive calls.

Remediation

ACKNOWLEDGED: The **BitFlux team** acknowledged this issue and stated the following:
Reentrancy protection is indeed crucial for preventing malicious recursive calls. We have already implemented reentrancy protection using OpenZeppelin's ReentrancyGuard in critical areas where external calls are made. However, for other functions where reentrancy risks are minimal or non-existent (e.g., pure view functions or internal operations), we have opted not to apply reentrancy protection to avoid unnecessary gas costs.

7.7 SUBOPTIMAL GAS USAGE DUE TO POST-INCREMENT IN LOOPS

// INFORMATIONAL

Description

In multiple functions across the codebase, the `for` loops use `i++` (post-increment) for incrementing the loop counter. In Solidity, post-increment (`i++`) is slightly less efficient than pre-increment (`++i`) because `i++` requires storing the original value of `i` in a temporary variable before incrementing, which consumes more gas. Although the gas difference is minimal, especially in recent Solidity versions, it becomes noticeable in larger loops or frequently executed functions, leading to inefficiencies in contract execution. The affected functions are the following:

- `Router.convert`
- `Router.addLiquidity`
- `Router.removeLiquidity`
- `Router.calculateConvert`
- `Swap.initialize`
- `Swap.getAdminBalances`
- `SwapUtils.calculateWithdrawOneTokenDY`
- `SwapUtils.getYD`
- `SwapUtils.getD`
- `SwapUtils._xp`
- `SwapUtils.getY`
- `SwapUtils._calculateRemoveLiquidity`
- `SwapUtils.calculateTokenAmount`
- `SwapUtils.addLiquidity`
- `SwapUtils.removeLiquidity`
- `SwapUtils.removeLiquidityImbalance`
- `SwapUtils.withdrawAdminFees`

BVSS

AO:A/AC:L/AX:H/R:N/S:U/C:N/A:N/I:N/D:L/Y:N (0.8)

Recommendation

To optimize gas usage, especially when iterating over large arrays or loops, it is recommended to replace `i++` with `++i`. Pre-increment (`++i`) does not require storing the old value of `i`, making it slightly more efficient in terms of gas consumption.

Remediation

ACKNOWLEDGED: The **BitFlux team** acknowledged this issue and stated the following:

We acknowledge that using post-increment (i++) instead of pre-increment (++i) can lead to marginally higher gas usage. However, this difference is negligible in practice and does not significantly impact overall gas efficiency. Given that post-increment is more widely used and understood by developers, we have chosen to prioritize readability and consistency across our codebase over micro-optimizations.

References

contracts/stable-amm/Router.sol#L27, L37, L63, L79, L125, L134, L229
contracts/stable-amm/Swap.sol#L130, L307
contracts/stable-amm/SwapUtils.sol#L224, L280, L294, L319, L330, L332, L385, L456, L476, L576, L609, L767, L803, L870, L978, L986, L1006, L1028

Halborn strongly recommends conducting a follow-up assessment of the project either within six months or immediately following any material changes to the codebase, whichever comes first. This approach is crucial for maintaining the project's integrity and addressing potential vulnerabilities introduced by code modifications.