STATE MIND

Curve Stablecoin

27-08-2023 - 30-10-2023

Table of contents

1. Project Brief

Title	Description
Client	Curve
Project name	Curve Stablecoin
Timeline	27-08-2023 - 30-10-2023
Initial commit	8dc86da7887eddd90d5132b441e8d378856e3e64
Final commit	037216578a6b90ba0a5c00db396df1d1910f2c8a

Short Overview

Curve Stablecoin is a over-collateralized USD stablecoin powered by a unique liquidating algorithm (LLAMMA), which progressively converts the put-up collateral token into crvUSD when the loan health decreases to certain thresholds. The design of the system has few concepts: LLAMMA, PegKeeper, Monetary Policy are the most important ones.

Stablecoin is a CDP where one borrows stablecoin against a volatile collateral (cryptocurrency, for example, against ETH). The collateral is loaded into LLAMMA in such a price range (such bands) that if price of collateral goes down relatively slowly, the ETH gets converted into enough stablecoin to cover closing the CDP. If prices rebound, crvUSD will be converted back into collateral, and the user's assets will be repurchased. This avoids permanent losses for the user in a volatile market.

Automatic monetary policy and PegKeeper mechanisms can help with peg-keeping. Policy is a dynamic component for adjusting lending rates, used to regulate interest rates based on market supply and demand, peg's state. PegKeepers are contracts that help stabilize the peg of crvUSD. Each Keeper is allocated a specific amount of crvUSD to secure the peg. they can single-sidedly deposit /withdraw into/from liquidity pool.

Project Scope

The audit covered the following files:

- AMM.vy
- Controller.vy
- ControllerFactory.vy
- Stablecoin.vy
- Stableswap.vy
- OwnerProxy.vy
- StableswapFactory.vy
- StableswapFactoryHandler.vy
- AggMonetaryPolicy.vy
- AggMonetaryPolicy2.vy
- AggregateStablePrice.vy
- AggregateStablePrice2.vy
- CryptoWithStablePrice.vy
- CryptoWithStablePriceAndChainlink.vy
- CryptoWithStablePriceAndChainlinkFrx
- CryptoWithStablePriceETH.vy
- CryptoWithStablePriceFrxethN.vy
- CryptoWithStablePriceTBTC.vy
- CryptoWithStablePriceWBTC.vy
- CryptoWithStablePriceWstethN.vy
- EmaPriceOracle.vy
- PegKeeper.vy
- HealthCalculatorZap.vy
- LeverageZap.vy
- LeverageZapSfrxETH.vy
- LeverageZapWstETH.vy

2. Finding Severity breakdown

All vulnerabilities discovered during the audit are classified based on their potential severity and have the following classification:

Severity	Description
Critical	Bugs leading to assets theft, fund access locking, or any other loss of funds to be transferred to any party.
High	Bugs that can trigger a contract failure. Further recovery is possible only by manual modification of the contract state or replacement.
Medium	Bugs that can break the intended contract logic or expose it to DoS attacks, but do not cause direct loss of funds.
Informational	Bugs that do not have a significant immediate impact and could be easily fixed.

Based on the feedback received from the Customer regarding the list of findings discovered by the Contractor, they are assigned the following statuses:

Status	Description
Fixed	Recommended fixes have been made to the project code and no longer affect its security.
Acknowledged	The Customer is aware of the finding. Recommendations for the finding are planned to be resolved in the future.

3. Summary of findings

Severity	# of Findings
Critical	0 (0 fixed, 0 acknowledged)
High	2 (2 fixed, 0 acknowledged)
Medium	4 (3 fixed, 1 acknowledged)
Informational	31 (17 fixed, 14 acknowledged)
Total	37 (22 fixed, 15 acknowledged)

4. Conclusion

During the audit of Curve crvUSD codebase, 37 issues were found in total:

- 2 high severity issues (2 fixed)
- 4 medium severity issues (3 fixed, 1 acknowledged)
- 31 informational severity issues (17 fixed, 14 acknowledged)

The final reviewed commit is 037216578a6b90ba0a5c00db396df1d1910f2c8a

5. Findings report

HIGH-01	Dynamic fee possible overflow	Fixed at c1a7b7

Description

In the function **AMM.limit_p_o**, the variable **ratio** represents a dynamic fee. As per the comments, the variable **ratio** is assured to be less than **1e18**. However, there's no check to enforce this assurance and it can be violated leading to critical consequences.

```
@internal
@view
def limit_p_o(p: uint256) -> uint256[2]:
    ...

    dt: uint256 = unsafe_sub(PREV_P_O_DELAY, min(PREV_P_O_DELAY, block.timestamp - self.prev_p_o_time))
    ratio: uint256 = 0
    ...

    # ratio = p_o_min / p_o_max

    ...

    # ratio is guaranteed to be less than 1e18
    # Also guaranteed to be limited, therefore can have all ops unsafe
    ratio = unsafe_div(
        unsafe_mul(
            unsafe_sub(unsafe_add(10**18, old_ratio), unsafe_div(pow_mod256(ratio, 3), 10**36)),  # (f' + (1 - r**3))
            dt),                                                    # * dt / T
        PREV_P_O_DELAY)
```

The formula can be written as $fee_{new} = (fee_{old} + (1 - r^3)) \cdot \frac{dt}{T}$, where $r = \max(\frac{p_{min}}{p_{max}}, \Delta_{max})$, $\Delta_{max} = 0.8$. Thus guaranteed $(1 - r^3) \leq (1 - 0.8^3) \approx 0.49 < 0.5$. However, as fee_{new} includes fee_{old}, the cumulative value may exceed 1. For example, if the oracle price changes by 51% and we update the dynamic fee with three calls of **limit_p_o** in three transactions in consecutive blocks (thus $\frac{dt}{T} = \frac{120-12}{120} = \frac{108}{120}$):

```
T = 120
dt = 108
rs = [0.8, 0.8, 0.89]
f = 0
for r in rs:
    f = (f + (1 - r**3)) * dt / T
print(f)

> 1.01655989999999998
```

This overflow allows all bands to be exchanged for 1 token (x or y) in case of **admin_fee** set to zero. Consider the **calc_swap_out** function in case of exchange x -> y.

```
@internal
@view
def calc_swap_out(pump: bool, in_amount: uint256, p_o: uint256[2], in_precision: uint256, out_precision: uint256) ->
DetailedTrade:
    ...

    antifee: uint256 = unsafe_div(
    (10**18)**2,
    unsafe_sub(10**18, max(self.fee, p_o[1]))
  ) # <-- will be zero due to cast of negative number
  admin_fee: uint256 = self.admin_fee # <-- set to zero
  ...
  for i in range(MAX_TICKS + MAX_SKIP_TICKS):
      if pump:
          ...
          x_dest: uint256 = (unsafe_div(lnv, g) - f) - x
      dx: uint256 = unsafe_div(x_dest * antifee, 10**18) # <-- will be 0
      if dx >= in_amount_left: # <-- will be false
          ...
              else:
                  dx = max(dx, 1) # <-- will be 1
                  x_dest = unsafe_div(unsafe_sub(dx, x_dest) * admin_fee, 10**18) # <-- will be zero
                  in_amount_left -= dx
                  out.ticks_in[j] = x + dx - x_dest
                  out.in_amount += dx # <-- provided 1
                  out.out_amount += y # <-- in exchange for the whole band
                  out.admin_fee = unsafe_add(out.admin_fee, x_dest)
```

Thus we get all the tokens y in exchange for 1 token x. The circle will go through all the bands and afterwards, there will be no checks that prevent this exchange.
In case of **admin_fee** set to non-zero, then there will probably be an overflow in calculation **x_dest** leading to revert, but in some cases, it can lead to incorrect calculation of **admin_fee**

```
x_dest = unsafe_div(unsafe_sub(dx, x_dest) * admin_fee, 10**18) # <-- will revert due to overflow in multiplication or an incorrect amount of fee.
in_amount_left -= dx
out.ticks_in[j] = x + dx - x_dest
out.in_amount += dx
out.out_amount += y
out.admin_fee = unsafe_add(out.admin_fee, x_dest) <-- will store large amount of fee
```

Recommendation

We recommend implementing an explicit assertion that ensures the **ratio** never exceeds **1e18**.

Client's comments

In current markets - admin_fee=1 in AMM fixes it. In new markets - see commits

HIGH-02	Method max_borrowable of a controller returns an incorrect amount	Fixed at a13629

Description

max_borrowable method of the controller is designed to determine the amount of debt the user can ask with given collateral. However, to create a loan different checks must be passed.
Method _create_loan used _calculate_debt_n1 to calculate the band for a loan

```
n1 = min(n1, 1024 - convert(N, int256)) + n0
if n1 <= n0:
assert AMM.can_skip_bands(n1 - 1), "Debt too high" # can't skip it in fact

# Let's not rely on active_band corresponding to price_oracle:
# this will be not correct if we are in the area of empty bands
assert AMM.p_oracle_up(n1) < AMM.price_oracle(), "Debt too high"
```

However, the debt amount that **max_borrowable** returns is too big to pass these assert statement cause it leads to small **n1** that can't be skipped here.

```
p_oracle: uint256 = AMM.price_oracle()
# Should be correct unless price changes suddenly by MAX_P_BASE_BANDS+ bands
n1: int256 = unsafe_div(self.log2(AMM.get_base_price() * 10**18 / p_oracle), LOG2_A_RATIO) + MAX_P_BASE_BANDS
p_base: uint256 = AMM.p_oracle_up(n1)
n_min: int256 = AMM.active_band_with_skip()
```

In the method max_p_base there is some adjust for **n1** variable. But it is done only for **n1** greater than active band. However if **n1** is small enough it should also be checked. For example, it can be too small due to division by p_oracle.
Thus, it is impossible to use the **max_borrowable** method to determine the debt amount under some market conditions, exactly, when the price of amm is sufficiently less than the price of the oracle. In the PoC it is about 4%.
The same **_max_p_base** code is used in the LeverageZap contracts, effectively blocking leverage calculations under the same circumstances.
The PoC script was handed over to the customer.

Recommendation

We recommend reconsidering the logic of the **max_borrowable** method.

MEDIUM-01	Missing default_return_value parameter	Fixed at c622ae

Description

In **Controller** contract **COLLATERAL_TOKEN** variable is instance of **ERC20** interface, where each method returns **bool** value. During collateral transfer at Line 1056 **default_return_value** is not indicated, so if collateral token doesn't return any value, this potentially could lead to execution revert.

Recommendation

It is recommended to provide **default_return_value** while calling **transferFrom()** method.

MEDIUM-02	Temporary DOS via get_D	Acknowledged

Description

The **get_D** function is used to calculate **D** via Newton root approximation and determines convergence via **abs(D - D_prev) <= 1** criteria in **255** iterations. If the method passes all **255** iterations without converging it will revert. There are a significant number of such inputs when **Dprev** and **D** will never be close during an execution.
Examples of reverting inputs: **xp = [200000000000000000000, 12515845246266009575], A = 500**
xp = [100000000000010000000000, 6148914691236510000], A = 500
xp = [14000013676000000010010000, 3367602000000000000051659], A = 100
xp = [20020000000000002000000000, 1060000000000000000001060], A = 50
We can set the storage to make **get_D** always revert via calling **exchange**, **remove_liquidity_one_coin**, or **remove_liquidity** as they do not recalculate **D** after getting new tokens. Although it's fairly easy to exit the **DOS** state (if anyone calls **remove_liquidity** with dust values) if the pool is trading near those 'revert' values an attack could be executed again.
Important to note, that the only **xp** values leading to revert are values far apart from each other (the minimum **x/y** we found is ~**500**). Due to this fact, we can attack only heavily imbalanced **stableswap** pools.
However, an attacker can completely **DOS** the contract if they are the only LP in the pool. (e.g. pool deployment).

Recommendation

We recommend using private mempools for **stableswap** deployment. Another way would be to change **raise** to **return D** as some of the 'problematic' inputs loops around the real **D** with low error, but that would mean some of the calculations can give wrong results.

Client's comments

> Known issue in stableswap. If this will be a problem - will need to create and seed the pool in the same tx for example (very possible)

MEDIUM-03	Unchecked default_return_value	Fixed at aee675

Description

- In the **repay_extended** the **COLLATERAL_TOKEN**s **transferFrom** invocation with the parameter **default_return_value=True** but without assertion. The callback is unknown (users are free to use their implementation). There is a scenario in which we assume:
  1. **COLLATERAL_TOKEN**'s **transferFrom** did not revert and did not transfer.
  2. **callbacker** has some tokens on balance and returned some **cb.collateral > 100 * N** (which also has to pass **self._calculate_debt_n1(cb.collateral, debt, size)**). In the partial repayment block, we create a new position (via **deposit_range**) with **cb.collateral** value, essentially reducing users' collateral. This scenario might occur when a user unknowingly passes a certain callback with a token balance (which could be some type of targeted griefing attack). Although, currently all included collaterals are reverting to invalid **transferFrom** it may be relevant in the future.
- In all the leverage zap constructors **approve** calls with **default_return_value=True**.

Recommendation

We recommend calling tokens with the **assert** keyword paired with the **default_return_value=True** parameter.

Client's comments

> Not a problem with existing collaterals though

MEDIUM-04	Possible to change the liquidation discount of any user	Fixed at 59374c

Description

In the Controller contract there is add_collateral method. As the name indicates, this method allows you to add collateral to any user.

```
if collateral == 0:
    return
self._add_collateral_borrow(collateral, 0, _for, False)
self._deposit_collateral(collateral, msg.value)
```

However in _add_collateral_borrow method the **liquidation_discounts[_for]** is changed without checking the **msg.sender**

```
AMM.deposit_range(_for, xy[1], n1, n2)
self.loan[_for] = Loan({initial_debt: debt, rate_mul: rate_mul})
liquidation_discount: uint256 = self.liquidation_discount
self.liquidation_discounts[_for] = liquidation_discount
```

Thus, it is possible to change liquidation_discount for any user whose debt is not in underwater mode. To do this you should just add small collateral to that user using add_collateral method. As mentioned here this is not intended behavior.

Recommendation

We recommend checking that **msg.sender** equals to **_for**, otherwise doesn't change **self.liquidation_discounts[_for]**.

INFORMATIONAL-01	The target_debt_fraction can be zero	Fixed at 2ffef6

Description

The **target_debt_fraction: uint256** storage variable has no minimum value, allowing it to be **0**. At the same time, the **calculate_rate** function uses the **target_debt_fraction** variable as a divisor in the second **power** calculation.

```
power -= convert(pk_debt * 10**18 / total_debt * 10**18 / target_debt_fraction, int256)
```

Recommendation

We recommend setting an explicit minimum value (**>0**) for the **target_debt_fraction** variable.

INFORMATIONAL-02	Mistakes and typos in comments	Fixed at e4bbbf

Description

1. In AMM.vy Line 373 there is a mistake. It should be **((A - 1) / A) ** n = exp(-n * ln(A / (A - 1))) = exp(-n * LOG_A_RATIO)**.
2. In Line 776 ds can't be zero because if we divide for same number we will get 1, but comment is correct for next line. **ds == user_shares[i]** in case of **frac == 10**18**.
3. In Line 788 **transfer** instead of **tranfer**.
4. In Controller.vy Line 502 **N * 1** can be reduced to **N**.
5. In the **CryptoWithStablePriceTBTC** line 79, there is a typo, the comment should be **# d_usd/d_tbtc**

Recommendation

We recommend fixing the comments.

INFORMATIONAL-03	Some asserts can be checked earlier	Fixed at c2fb44

Description

In some functions like deposit_range some asserts can be checked before computations e.g. checks after the cycle can be moved above the for.

```
...
    n_bands: uint256 = unsafe_add(convert(unsafe_sub(n2, n1), uint256), 1)
    assert n_bands <= MAX_TICKS_UINT

    y_per_band: uint256 = unsafe_div(amount * COLLATERAL_PRECISION, n_bands)
    assert y_per_band > 100, "Amount too low"

    assert self.user_shares[user].ticks[0] == 0  # dev: User must have no liquidity
    self.user_shares[user].ns = unsafe_add(n1, unsafe_mul(n2, 2**128))
...
```

Recommendation

We recommend performing all checks before computing anything else.

INFORMATIONAL-04	Incorrect interface for BaseRegistry	Acknowledged

Description

In **StableswapFactoryHandler** there is an interface declaration of **BaseRegistry**. Based on current logic and deployments **base_registry** will be a **StableswapFactory** instance. But the interface doesn't match with the actual **StableswapFactory**:

1. There is no function get_lp_token()
2. Function get_coin_indices() actually returns three variables - (int128, int128, bool).

Recommendation

It is recommended to make the interface match the actual contract.

Client's comments

Fixed in stableswap-ng

INFORMATIONAL-05	Redundant MSTORE in _raw_price	Fixed at 835ce2

Description

In the **CryptoWithStablePriceFrxethN**'s **_raw_price** function, the **self.use_chainlink** storage variable is saved into the memory just to be used in one expression.

Recommendation

We recommend removing redundant memory variables.

INFORMATIONAL-06	Potential irreversible DoS	Fixed at 3dd932

Description

In the **ControllerFactory** there is no possibility to remove the market (**AMM** and **Controller** pair). It can lead to irreversible DoS of all markets without the possibility to withdraw funds for users in case of broken or malicious implementation deployed. It will happen in case when controller **total_debt()** reverts.

```
@external
@view
def total_debt() -> uint256:
    """
    @notice Sum of all debts across controllers
    """
    total: uint256 = 0
    n_collaterals: uint256 = self.n_collaterals
    for i in range(MAX_CONTROLLERS):
        if i == n_collaterals:
            break
        total += Controller(self.controllers[i]).total_debt() # <-- revert in one cause DoS of the whole protocol
    return total
```

In this case, it won't be possible to calculate **total_debt()** which is used for **rate** calculation which is used in **Controller** functions: **create_loan**, **create_loan_extended**, **add_collateral**, **remove_collateral**, **repay**, **repay_extended**, **liquidate**, **liquidate_extended**, **collect_fees**.
Also, there may be a case when we want to remove some "bad" market and it is better to have that option.

Recommendation

We recommend adding a market removal function to prevent possible DoS.

Client's comments

MonetaryPolicies need to be redeployed

INFORMATIONAL-07	Uncertain coins amount in stableswap	Acknowledged

Description

StableSwap initialize method can take up to 4 coins to be used in the pool. However, it is not intended to use more than 2 coins. Also, it is possible to initialize the pool with only one coin cause of this statement. We just need to provide only one non-zero coin address and three zero addresses.

Recommendation

We recommend adding a check that the first two coins are non-zero and setting the max **_coins** length to 2 in the **initialize** method.

Client's comments

Correct - this factory is a quick change of the existing one. Made all more consistent in stableswap-ng

INFORMATIONAL-08	Redundant reentrant lock	Acknowledged

Description

HealthCalculatorZap contract has only one external function **health_calculator** with no possibility of flow hijacking.

Recommendation

We recommend removing the reentrant lock on the **health_calculator** function.

Client's comments

> Zap is only to be used in the frontend, only in a market where code requires "patching" via the read-only zap

INFORMATIONAL-09	Logarithm rounding	Fixed at 7edcf8

Description

In function max_p_base() when calculating **n1** there is no processing of case when **self.log2(AMM.get_base_price() * 10**18 / p_oracle)** gives negative value. At Lines 421 - 422 this case is handled using subtraction **LOG2_A_RATIO - 1**.

Recommendation

It is recommended to handle rounding issues case when log2() gives negative values.

INFORMATIONAL-10	Potential revert of PegKeeper	Acknowledged

Description

The **PegKeeper** contract update that was supposed to provide stablecoins to the pool may revert due to lack of available balance. Revert can occur either due to a large price change in the pool or price manipulation using sandwich swaps.

```
@internal
def _provide(_amount: uint256):
    if _amount == 0:
        return

    amounts: uint256[2] = empty(uint256[2])
    amounts[I] = _amount
    POOL.add_liquidity(amounts, 0) # <-- will revert in case of insufficient balance

    self.last_change = block.timestamp
    self.debt += _amount
    log Provide(_amount)
```

In the **_provide** function there is no check that **_amount** is lower or equal to the current stablecoin balance. So when **balance <= _amount = (balance_peg - balance_pegged) / 5** then there will be a revert.

Recommendation

We recommend checking the available balance of stablecoins in the **_provide** function and in case of exceeding the balance, provide the available amount.

Client's comments

> To be fixed in the new incarnation of PegKeeper (out of scope here)

INFORMATIONAL-11	Chainlink stale data	Acknowledged

Description

Some of the contracts do not check if feed data is fresh enough to use.

1. CryptoWithStablePriceAndChainlink - no staleness check
2. CryptoWithStablePriceAndChainlinkFrxeth - no staleness check
3. CryptoWithStablePriceWstethN - **CHAINLINK_AGGREGATOR_ETH** checked with **CHAINLINK_STALE_THRESHOLD = 86400 = 24 hours**, while **ETH/USD** feed has **Heartbeat = 3600 = 1 hour**.

Although chainlink price feeds are not actively used, some oracles have no **use_chainlink** switch.

Recommendation

We recommend checking chainlink price feed update timings.

Client's comments

Can only be fixed for future price oracles / we are not going to use Chainlink there anyway (leads to increased losses apparently)

INFORMATIONAL-12	Some divisions can be unsafe	Fixed at ad3f05

Description

As in Line 782 **s** can't be zero because of **DEAD_SHARES** never 0.

The same examples:

- Line 793 **COLLATERAL_PRECISION** is always not zero.
- Controller.vy Line 200 **Aminus1** is never zero because **A** minimum is 2.
- Line 201 **A_ratio** calculation can be unsafe. As **max(10**18 * A) = 10**22** and **min(A - 1) = 1**.
- Same for the **LOG2_A_RATIO** in the leverage zaps constructors. 1, 2, 3.
- Line 376 **max(collateral / N, DEAD_SHARES)** is minimum **DEAD_SHARES** which is not zero.
- Line 415 **(debt + 1) > 0**
- In **HealthCalculatorZap** function **get_y_effective** expression **max(collateral / N, DEAD_SHARES)** is always greater than zero.
- In the **AggMonetaryPolicy[2]**, the **power** calculation, unsafe division by **sigma** as its **MIN_SIGMA = 10**14**.
- In the **AggMonetaryPolicy[2]**, the second **power** calculation, unsafe division by **total_debt**.
- In the **AggMonetaryPolicy[2]**, the return value calculation, unsafe division by **10**18**

Recommendation

We recommend using unsafe_div here.

Client's comments

Scattered across various commits Left unchanged for contracts that have no bytespace constraints for readability

INFORMATIONAL-13	get_underlying_decimals can revert	Acknowledged

Description

get_underlying_decimals can revert in case if there is 8 token in pool, because of **decimals[i+1]** which will overflow if **i == 7**.
The same is happening at get_underlying_balances.

Recommendation

We recommend changing iterations or allocating **MAX_COINS + 1** for the **decimals** array as it is intended.

Client's comments

Fixed in stableswap-ng factory (elsewhere)

INFORMATIONAL-14	Too high limit for A	Fixed at 0c5659

Description

In ControllerFactory, there is a restriction that **A** can be up to 10000. However, some calculations in AMM.vy can't be done with a high **A**. **A ** 25 * 10 ** 18** needs to be less than 2^256 to not cause overflow. So for **x^25*10^18 < 2 ^256** x needs to be **x<230.434**.

```
MAX_ORACLE_DN_POW = unsafe_div(pow_mod256(unsafe_div(A**25 * 10**18, pow_mod256(Aminus1, 25)), 2), 10**18)
                                          # overflow here ---^
```

Recommendation

We recommend setting other limits for **A** in ControllerFactory.

Client's comments

Changed to a more readable cycle in init. Not gas-optimal (?) but bytecode-efficient

INFORMATIONAL-15	Can set incorrect rate in AMM	Fixed at fc34fd

Description

There is a callback mechanism in liquidate_extended method of Controller. In the beginning of the flow

```
debt: uint256 = 0
rate_mul: uint256 = 0
debt, rate_mul = self._debt(user)
```

Method _debt itself uses method _rate_mul_w that call **monetary_policy** to get **rate_mul**. To do this **monetary_policy** uses calculate_rate method. Also **rate** from **monetary_policy** is provided and saved in amm here
Let's look closer to the code at the end of calculate_rate method

```
power: int256 = (10**18 - p) * 10**18 / sigma  # high price -> negative pow -> low rate
if pk_debt > 0:
    total_debt: uint256 = CONTROLLER_FACTORY.total_debt() # get old total_debt
    if total_debt == 0:
        return 0
    else:
        power -= convert(pk_debt * 10**18 / total_debt * 10**18 / target_debt_fraction, int256)

return self.rate0 * min(self.exp(power), MAX_EXP) / 10**18
```

As we can see less **CONTROLLER_FACTORY.total_debt()** debt leads to a smaller **rate**, while more **CONTROLLER_FACTORY.total_debt()** leads to a larger **rate**. The problem is that **CONTROLLER_FACTORY.total_debt()** can be outdated and incorrect.
Let's return to the **liquidate_extended** method. As stated current **rate** is saved in amm. Then in a callback we can call **liquidate_extended** of another controller to liquidate another loan. In a new **liquidate_extended** callback we will also try to liquidate another user in another controller while we can do it.
Acting like this we will always update **rate** according to the state of the system before the first call. If we liquidate these users one by one **CONTROLLER_FACTORY.total_debt()** will decrease, so the **rate** in amm also will decrease. However the **rate** is saved before **total_debt** is changed so **monetary_policy** gets an incorrect amount of **CONTROLLER_FACTORY.total_debt()**.
So we calculate and save a greater **rate** than we should if we liquidate users one by one.
Note, that the same attack can be done in repay_extended method.
The same issue is valid also for AggMonetaryPolicy2.

Recommendation

We recommend changing **self._total_debt** before a callback.

Client's comments

MonetaryPolicy3 uses min TVL over 1 day, new implementations have rate updated at the end

INFORMATIONAL-16	Possible inconsistency between total_debt and circulating supply	Acknowledged

Description

Currently, we can get a number of **crvUSD** in circulation by subtracting Controller's **redeemed** value from **minted**. Actually, its value equals to the value of **total_debt** without accrued interests. **Controller** doesn't mint new tokens, this is done by **ControllerFactory**, and **Controller** just holds them and transfers them when a new loan is created. Due to the fact that **Controller** doesn't mint new tokens, there may be inconsistency between the actual **circulating supply** and the calculated one.

```
minted = 1000
redeemed = 800

actual_circulating_supply = 200
calculated_circulating_supply = minted - redeemed = 1000 - 800 = 200

# User creates loan and borrows 200 crvUSD

minted = 1200
redeemed = 800

actual_circulating_supply = 400
calculated_circulating_supply = minted - redeemed = 1200 - 800 = 400

# Then, the user transfers his tokens to Controller and another user creates a loan with these tokens

minted = 1400
redeemed = 800

actual_circulating_supply = 400  # because Controller used tokens that are not minted by ControllerFactory to create a new position
calculated_circulating_supply = minted - redeemed = 1400 - 800 = 600
```

Recommendation

It is recommended not to rely on the circulating supply calculated in **Controller** because it may differ from the actual one.

Client's comments

> Unclear where it can cause issues, but noted

INFORMATIONAL-17	Wrong constant limit in exponent	Fixed at ff00a0

Description

In the exp function there is a check:

```
if power <= -42139678854452767551:
    return 0
```

This is not an accurate number, the correct limit is **-41446531673892821376**.

Recommendation

We recommend setting a correct limit to prevent extra calculations in some cases.

INFORMATIONAL-18	Unify the percentage of APY	Fixed at 9fe670

Description

In AggMonetaryPolicy2.vy, there is a comment

```
MAX_RATE: constant(uint256) = 43959106799  # 300% APY
```

But in other contracts :1, 2 - the same variable is a 400%.

Recommendation

We recommend unifying these constants.

INFORMATIONAL-19	Incorrect function name in raw_call	Acknowledged

Description

The **StableswapFactoryHandler** contract has the **get_gauges(_pool: address)** function to get gauges of the provided pool.
The internal **_get_gauge_type** function queries **GAUGE_CONTROLLER** to get the gauge type id via **gauge_type(address)** invocation, while **GAUGE_CONTROLLER** has only **gauge_types(_addr: address)** function.
Currently any call to the **get_gauges(_pool: address)** will end up reverting.
The **StableswapFactoryHandler** has the right interface **GaugeController** to call **GAUGE_CONTROLLER** contract, but never used.

Recommendation

We recommend using the **GaugeController** interface or fixing raw_call **method_id**.

Client's comments

Fixed in stableswap-ng

INFORMATIONAL-20	Lack of exponent tests with negative number	Fixed at 7d7daf

Description

In the exponent test only non-negative power is tested. It doesn't cover negative tests, which are used in oracles.

Recommendation

We recommend extending exponent tests for negative power.

INFORMATIONAL-21	ControllerFactory can mint to/burn from any address	Acknowledged

Description

ControllerFactory can mint to/burn from any address via set_debt_ceiling method. Although this method can only be called by the admin it is supposed to be used only for controller contracts.

Recommendation

We recommend adding checks that address belongs to a controller or adding a whitelist mechanism, to give crvUSD to non-controllers.

Client's comments

Correct -> This is to allow markets for other chains in the future. After can transfer ownership to an immutable proxy

INFORMATIONAL-22	Stableswap inconsistency between whitepaper and code	Acknowledged

Description

The Stableswap's invariant as specified in the whitepaper differs significantly from what's implemented in the code. The whitepaper presents the invariant as:

$$An^n \sum x_i + D = ADn^n + \frac{D^{n+1}}{n^n \prod x_i}$$

In contrast, the code implements the following invariant:

$$An \sum x_i + D = ADn + \frac{D^{n+1}}{n^n \prod x_i}$$

The discrepancy stems from the replacement of An^n in the whitepaper with An in the code (see 1, 2, 3, 4).

This relates to the leverage χ, which according to whitepaper is equal to $\chi = \frac{A \prod x_i}{(\frac{D}{n})^n} = \frac{An^n \prod x_i}{D^n}$ and in code it is $\chi = \frac{A \prod x_i}{\frac{D^n}{n}} = \frac{An \prod x_i}{D^n}$.

The change in formula misinterprets parameter A by a factor of n^{n-1}. Pool creators who base their parameter A choices on the whitepaper are misled, leading to inaccurate configurations.

Recommendation

We recommend aligning the code with the whitepaper's formula to avoid inconsistencies.

Client's comments

Correct - well-known inconsistency and smart contract's definition is more convenient

INFORMATIONAL-23	Rate manipulation	Fixed at fc34fd

Description

In **Controller** before any manipulation with loan, user debt and borrow rate are updated. The logic of update is implemented in function set_rate(). Firstly, **rate_mul** is updated with old **rate**, then new **rate** and **rate_time** are set. Based on that order, we can create a big loan to increase debt, then repay it for the new manipulated rate to take effect. If the loan is created and repaid in one transaction or one block, then the borrower loses nothing. For the new **rate** to be reflected on **rate_mul**, we need to place our transaction at the end of the block. Then in the new block **new_rate** will be accumulated and the positions of users could be liquidated.

Recommendation

It is recommended to implement rate updates in such a way that it will reflect the most recent changes in total debt, not the previous ones.

Client's comments

MonetaryPolicy3 uses min TVL over 1 day, new implementations have rate updated at the end

INFORMATIONAL-24	Restrictions for sigma can be int256	Fixed at 46c800

Description

MAX_SIGMA and **MIN_SIGMA** in AggMonetaryPolicy2.vy and AggMonetaryPolicy.vy are uint256, but **self.sigma** is int256. These constants can be changed to int256, for code clarity and saving some convert operations.

Recommendation

We recommend changing these constants and setter functions.

INFORMATIONAL-25	Unchecked parameters during **stableswap** deployment	Acknowledged

Description

Neither **stableswap**s **initialize** nor **factory**'s **deploy_plain_pool** functions have maximum value checks for **_A** and **_fee** input variables. Hence, the deployer can make a pool with **fee > MAX_FEE = 5 * 10 ** 9** and/or **A > MAX_A = 10**6**. Also, **deploy_plain_pool** has check **_fee > 0**, while **stableswap** allows setting **_fee = 0**.

Recommendation

We recommend checking input parameters during initialization.

Client's comments

Fixed in stableswap-ng

INFORMATIONAL-26	Inconsistent 2-step factory ownership transfer	Acknowledged

Description

In the **OwnerProxy** contract there are two admin methods to commit and accept ownership transfer in a particular factory. In commit_transfer_ownership **msg.sender** is checked. But function accept_transfer_ownership() is open and anyone can call it and accept pending admin in **factory**. This makes a meaningless 2-step transfer and doesn't prevent from accepting incorrect admin.

Recommendation

It is recommended to add **msg.sender** check in **accept_transfer_ownership()**. Either check if it is called by **OwnerProxy**'s admin or by the pending admin in the target factory.

Client's comments

Fixed in stableswap-ng

INFORMATIONAL-27	Unnecessary variables update	Fixed at 7b7010

Description

In deposit_range() function at Lines 737-738 values of variables **rate_mul** and **rate_time** are updated, but they have no effect. **deposit_range()** is only called from **Controller** and in every function that calls it there is an update of rate and a call to set_rate() in **AMM**. So **rate_time** always equals **block.timestamp** at the moment of update and based on calculations in the **_rate_mul()** function there will be no effect on the value of **rate_mul**.

Recommendation

It is recommended to remove unnecessary update.

INFORMATIONAL-28	HealthCalculatorZap functionality optimization	Acknowledged

Description

HealthCalculatorZap contract can be optimized by implementing the **_calculate_debt_n1** function from the controller. The **_calculate_debt_n1** function uses already ported function **get_y_effective** and also makes the same **AMM.p_oracle_up(n1)** call as the one after **calculate_debt_n1** invocation.

Recommendation

We recommend porting necessary functionality to the **health_calculator** function if lower gas cost is a requirement.

Client's comments

> Zap is only to be used in frontend, only in the market where code requires "patching" via the read-only zap

INFORMATIONAL-29	Strange naming	Fixed at ba987c

Description

At **CryptoWithStablePriceETH.vy** some variables are named with BTC. 1, 2.

Recommendation

We recommend changing variable names in this contract.

INFORMATIONAL-30	State mutability	Fixed at 906be5

Description

Several function state mutabilities can be restricted to pure. These functions do not read from storage.

- **Stableswap.vy**[1 2 3 4 5]
- **StableswapFactoryHandler**[1]
- **AMM**[1]
- **Controller**[1 2 3 4 5]
- **ControllerFactory** [1]
- **AggregateStablePrice** [1 2 3]
- **AggregateStablePrice2** [1 2 3]
- **HealthCalculatorZap** [1]
- **CryptoWithStablePrice**[1 2 3 4 5 6 7]
- **CryptoWithStablePriceAndChainlink**[1 2 3 4 5 6 7]
- **CryptoWithStablePriceAndChainlinkFrxeth**[1 2 3 4 5 6 7 8]
- **CryptoWithStablePriceFrxethN**[1]
- **CryptoWithStablePriceETH**[1]
- **CryptoWithStablePriceWBTC**[1]
- **CryptoWithStablePriceWstethN**[1]
- **EmaPriceOracle**[1 2 3]

Recommendation

We recommend changing mutability from view to pure.

INFORMATIONAL-31	Unsafe PegKeeper addition	Acknowledged

Description

In the current state, PegKeepers are deployed and provided with stablecoins independently, then added to monetary policy by its admin. This architecture grants significant power to Monetary Policy admins. A malicious PegKeeper has the potential to manipulate the **rate** or to initiate a DoS attack against the **calculate_rate** function.

Recommendation

We recommend implementing a factory contract explicitly for the creation of PegKeepers. This factory should also handle the minting of stablecoins to them, ensuring a more controlled process.

Client's comments

Cannot change factory but can make an immutable proxy that sets **MonetaryPolicy** once it stops receiving improvements

STATE
MIND