STATE MIND

# Curve lending

15-01-2024 - 02-02-2024

# Table of contents

# 1. Project Brief

| Title | Description |
| --- | --- |
| Client | Curve |
| Project name | Curve lending |
| Timeline | 15-01-2024 - 02-02-2024 |
| Initial commit | 5a939431084dcb1e7fb85c19c883048903a23beb |
| Final commit | d64d55877730e653ad1f0ea32c694a232c04ac53 |

## Short Overview

Curve lending allows the creation of permissionless lending/borrowing markets to borrow crvUSD against any token, or to borrow any token against crvUSD in an isolated mode, powered by LLAMMA for soft-liquidations. All markets are isolated from each other and do not intertwine.

The borrowable liquidity is provided by willing lenders through Vaults, which are ERC4626 contracts with some additional methods for convenience.

## Project Scope

The audit covered the following files:

- Vault.vy
- OneWayLendingFactory.vy
- SemilogMonetaryPolicy.vy
- CryptoFromPool.vy
- Controller.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 | 1 (1 fixed, 0 acknowledged) |
| High | 1 (1 fixed, 0 acknowledged) |
| Medium | 4 (4 fixed, 0 acknowledged) |
| Informational | 11 (8 fixed, 3 acknowledged) |
| Total | 17 (14 fixed, 3 acknowledged) |

# 4. Conclusion

During the audit of the codebase, 17 issues were found in total:

- 1 critical severity issues (1 fixed)
- 1 high severity issues (1 fixed)
- 4 medium severity issues (4 fixed)
- 11 informational severity issues (8 fixed, 3 acknowledged)

The final reviewed commit is d64d55877730e653ad1f0ea32c694a232c04ac53

# 5. Findings report

| CRITICAL-01 | Dilution of the depositor's share using callback functionality | Fixed at 7b823a |
|---|---|---|

### Description

In **Vault**, convertion to shares depends on **totalSupply** and **_total_assets**. **_total_assets** consists of balance of **Controller** contract and its **total_debt**. Ideally, when there is no interaction with **Vault**, if **balance** of **Controller** decreases, then **total_debt** should increase and vice versa. These actions should happen atomically. But, in function **create_loan_extended()** at Line 631 there is a transfer of **BORROWED_TOKEN**, and after **callback** actual loan is created and **total_debt** increases. These leads to underestimation of **Vault's shares** and inside call to **callbacker**, it is possible to buy shares at low price, thereby diluting shares of other depositors.

```
# Callbacker contract should have enough collateral to create position
# or may take flashloan and inside it call create_loan() + repay()

def create_loan_extended(...):
    self.transfer(BORROWED_TOKEN, callbacker, debt)
    # ^-- Here balance of Controller contract decrease, but total_debt stays the same,
    #     leading to lower _total_asset in Vault

    more_collateral: uint256 = self.execute_callback(
        callbacker, CALLBACK_DEPOSIT, msg.sender, 0, collateral, debt, callback_args).collateral
    # ^-- in callback we make deposit in Vault, using already transfered borrow tokens

    self._create_loan(collateral + more_collateral, debt, N, False)  # <-- here actual loan is created and total_debt value is
updated
    self.transferFrom(COLLATERAL_TOKEN, msg.sender, AMM.address, collateral)
    self.transferFrom(COLLATERAL_TOKEN, callbacker, AMM.address, more_collateral)
```

To repay debt and get collateral back attacker makea a withdraw from **Vault** and calls repay() function. After all these actions attacker gets back collateral and still have shares of **Vault**.
The PoC script was handed over to the customer.
Same issue:

- Via function **borrow_more_extended()** - same attack
- Via function **_liquidate()** - before callback **BORROWED_TOKEN** is transferred to Controller's balance, so price of share is increased. So, attacker, before making liquidation, can buy shares at low price, and them sell them during callback.

### Recommendation

It is recommended to keep operations atomic. Radical way of solving that is to disable **extended** functions to prevent external calls. Another way is to have invariant from **Vault** to check in **execute_callback()** function, but this breakward compatibility with classic **Controller**. E.g. **totalSupply** can be checked to remain the same.

```
band_x: uint256 = AMM.bands_x(data.active_band)
band_y: uint256 = AMM.bands_y(data.active_band)
vault_total_supply: uint256 = FACTORY.totalSupply()
# ^-- In lending FACTORY is actually VAULT, factory interfaces should be changed
#     or FACTORY should be replaced with separate VAULT variable

# Callback
response: Bytes[64] = raw_call(
    callbacker,
    concat(callback_sig, _abi_encode(user, stablecoins, collateral, debt, callback_args)),
    max_outsize=64
)
data.stablecoins = convert(slice(response, 0, 32), uint256)
data.collateral = convert(slice(response, 32, 32), uint256)

# Checks after callback
assert vault_total_supply == FACTORY.totalSupply()
assert data.active_band == AMM.active_band()
```

### Client's comments

There was a **check_lock()** method introduced in Controller. It reverts if reading it reenters the Controller. This method is invoked from Vault, so Vault is not working when Controller is currently running

| HIGH-01 | Incorrect implementation of Vault contract renders a set of crucial functions in Controller useless | Fixed at 1255bc |
|---|---|---|

### Description

The vulnerability arises from the **Controller** contract, deployed by the **Vault** contract, incorrectly assuming the presence of **admin** and **fee_receiver** variables in the **Vault** (set as its **FACTORY**). These variables, crucial for several functions in the **Controller**, are absent in the **Vault** contract. Consequently, the **Controller** attempts to access these variables on the **Vault** contract lead to transaction reversion.

The identified issue significantly impacts the **Controller** contract's functionality, particularly affecting critical functions dependent on the **admin** variable. Key functions such as

- **set_amm_fee()**
- **set_monetary_policy()**
- **set_borrowing_discounts()**

are rendered inoperative. These functions are essential for adapting to market dynamics, and their malfunction results in a considerable loss of the protocol's adaptability and responsiveness.

Furthermore, the **set_callback()** and **set_amm_admin_fee()** functions, though used less frequently, are also impaired. This loss of functionality, while not as frequent, still represents a notable deficiency in the contract's intended capabilities.

Lines:

Controller.vy#L1253

Controller.vy#L1260

Controller.vy#L1272

Controller.vy#L1291

Controller.vy#L1306

### Recommendation

If maintaining the deployment of the **Controller** through the **Vault** is a deliberate architectural choice, we recommend modifying the **Vault** contract to include the **admin** variable, along with any associated functionalities expected by the Controller. This adaptation should ensure that the **Vault** contract is fully equipped to support the **Controller**'s operations.

### Client's comments

Added factory and admin() method to the Vault. **fee_receiver** is not needed since lending is with zero admin fee by design, so it is supposed to revert

| MEDIUM-01 | All collected fees are stuck in AMM | Fixed at 671ac3 |
|---|---|---|

### Description

As it said in docs: "all the fees will go to the vault depositors."
But in fact, all collected fees are stored in the AMM contract, so there is no opportunity to withdraw them on the Controller contract and make it an incentive for liquidity providers. Currently, all collected fees go to borrowers.
Example from **calc_swap_out()**:

```
x_dest: uint256 = (unsafe_div(lnv, g) - f) - x
dx: uint256 = unsafe_div(x_dest * antifee, 10**18)
if dx >= in_amount_left:
    # This is the last band
    x_dest = unsafe_div(in_amount_left * 10**18, antifee)  # LESS than in_amount_left
    # ^-- (in_amount_left - x_dest) - all fees
    out.last_tick_j = min(lnv / (f + (x + x_dest)) - g + 1, y)  # Should be always >= 0
    x_dest = unsafe_div(unsafe_sub(in_amount_left, x_dest) * admin_fee, 10**18)  # abs admin fee now
    # ^-- admin fee. If admin_fee == 0, then x_dest == 0
    x += in_amount_left  # x is precise after this
    # Round down the output
    out.out_amount += y - out.last_tick_j
    out.ticks_in[j] = x - x_dest
    # ^-- fees excluding admin fees stay in the tick
    out.in_amount = in_amount
    out.admin_fee = unsafe_add(out.admin_fee, x_dest)
    # ^-- add admin fee
    break
```

After all, admin fees are collected in two tokens. But in Vault only one token is used to determine APR and pricePerShare.

### Recommendation

It is recommended to comment if fees should stay in AMM, then they would go to borrowers. In that case docs should be changed. If trading fees should go to depositors, then **ADMIN_FEE** should be changed to 100% for collecting them. To make it function **collect_fees()** may be used and in **Vault** contract **fee_receiver** should be presented and set to **Controller** address. Also during fee collection all tokens should be swap for **BORROWED_TOKEN**.

| MEDIUM-02 | Incorrect **NewVault** event parameter used in emit | Fixed at 2f2b94 |
|---|---|---|

### Description

Zero constant is used (instead of id) in the **NewVault** event emit. This will confuse the UI or offchain monitoring tools. It can be fixed by updating unused storage variables in the OneWayLendingFactory.
This issue also applies for TwoWayLendingFactory.

### Recommendation

We recommend replacing zero constant with n_vaults.

```
+self.vaults[self.n_vaults] == vault
+log NewVault(self.n_vaults, collateral_token, borrowed_token, vault.address, controller, amm, price_oracle, monetary_policy)
+self.n_vaults += 1
-log NewVault(0, collateral_token, borrowed_token, vault.address, controller, amm, price_oracle, monetary_policy)
```

| MEDIUM-03 | Incorrect pricePerShare() calculation | Fixed at 132cf6 |
|---|---|---|

### Description

The **Vault** contract an has external **pricePerShare()** function in which **totalSupply** is counted without **DEAD_SHARES**.
Related **_convert_to_shares()** and **_convert_to_assets()** functions are calculated with **DEAD_SHARES**.
This may lead to an incorrect calculation of **pricePerShare()** for a small value of **totalSupply**(~ 10^3).
When using **pricePerShare()** along with the oracle, the price will be calculated incorrectly. This can lead to further incorrect calculations of functions inside **AMM** and **Controller**.
In addition, since related functions take into account rounding when buying/selling shares, this function also needs such functionality.

### Recommendation

We recommend changing **pricePerShare()** calculation with DEAD_SHARES and add the possibility of rounding up and down.

| MEDIUM-04 | Rate updating when interacting with Vault | Fixed at 6067f5 |
|---|---|---|

### Description

Interactions with **Vault**, such as deposits and withdrawals, affect protocol's utilization. Rate is calculated in **SemilogMonetaryPolicy** and used in **AMM**. But function **_update_rates()** only updates internal state of **Monetary Policy** and doesn't trigger changes in **AMM**'s rate.
This leads to a discrepancy between the necessary rate, based on actual utilization, and rate in **AMM**.
E.g. when withdrawing from **Vault**, utilization increases, so rate should also increase. But in **AMM**, rate remains unchanged and borrowers are charged less interest, until there is no interaction with **Controller**.

### Recommendation

It is recommended to update rate in **AMM** after every action in **Vault** which leads to change in utilization (deposit, withdraw).

| INFORMATIONAL-01 | Excessive conditions | Fixed at 740a92 |
|---|---|---|

### Description

The **OneWayLendingFactory** contract has internal **_create** function which contains an assert to check the correctness of **max_rate** and **min_rate**. To check whether two sorted values belong within the interval, 3 conditions are sufficient instead of 5.
Also, the **set_default_rates** function / **SemilogMonetaryPolicy's constructor** have the same problem.

### Recommendation

We recommend using assert conditions like this:

```
min_rate <= max_rate and min_rate >= MIN_RATE and max_rate <= MAX_RATE
```

| INFORMATIONAL-02 | Redundant check for N | Fixed at ba450e |
|---|---|---|

### Description

In OneWayLendingFactory.vy#L235 there is a check that **N > 1** but if N is equal to 1, then one of **collateral_ix** and **borrowed_ix** is equal to 100. So it will be reverted earlier.

### Recommendation

We recommend removing this check.

| INFORMATIONAL-03 | Inconsistency in withdraw()/redeem() | Fixed at f32329 |
|---|---|---|

### Description

In the **redeem()** function we have a sequence as described in OZ implementation. First, we do burn, second - transfer. But in **withdraw()** token transfer is performed first.

### Recommendation

We recommend making related functions similar.

| INFORMATIONAL-04 | Log2 implementation has bad accuracy | Fixed at 711651 |
|---|---|---|

### Description

The **Controller.vy** contract has **log2** implementation. It is assumed that the calculation is accurate to 10 decimals. At the same time, 18 decimals are needed to accurately calculate **LOG2_A_RATIO**:

```
@external
def __init__(
        collateral_token: address,
        monetary_policy: address,
        loan_discount: uint256,
        liquidation_discount: uint256,
        amm: address):

    # ...

    AMM = LLAMMA(amm)
    _A: uint256 = LLAMMA(amm).A()
    A = _A
    Aminus1 = unsafe_sub(_A, 1)
    LOG2_A_RATIO = self.log2(unsafe_div(_A * 10**18, unsafe_sub(_A, 1))) # <-- 18 decimals

    # ...
```

The **log2** function calculates a logarithm smaller in magnitude than it actually is (The difference is on average 9 decimal places. For example **14499569660983974** vs. **14499569695115170** for Vault accuracy, with **A = 100**).

### Recommendation

We recommend increasing the accuracy of calculations, as in Vault implementation.

| INFORMATIONAL-05 | Importing a non-existent function | Fixed at 60f556 |
|---|---|---|

### Description

The **Controller** contract imports the **LLAMMA** interface. The imported **set_price_oracle** function does not exist in the **AMM** contract and is not used anywhere in the **Controller** contract.

### Recommendation

We recommend removing this function from the interface import.

| INFORMATIONAL-06 | Sanity checks for pricePerShare | Fixed at c2382b |
|---|---|---|

### Description

**pricePerShare** is calculated based on **_total_assets** and **totalSupply**. Attacker can manipulate **_total_assets** by simply transferring tokens on **Controller** contract or by making more complex actions. Contracts, that uses this function should have sanity checks, because **pricePerShare** is vulnerable to manipulations and situations when **10******18** * **self.precision** * **self._total_assets()** is less than **totalSupply** are possible and could lead to incorrect price of share.

### Recommendation

It is recommended to add sanity check for the case when **pricePerShare** equals zero (e.g. reverting an execution) and additional checks in contracts that uses this function to mitigate manipulations.

### Client's comments

> This is actually intended behavior: pricePerShare is measured in tokens normalized to 18 decimals. This behavior is used in two-way vaults internally. Added tests around pricePerShare()

| INFORMATIONAL-07 | State Manipulation Vulnerability in Controller Contract Functions | Acknowledged |
|---|---|---|

### Description

An analysis of the **Controller** contract's execution flow, particularly during **_create_loan_extended** and similar functions, reveals a vulnerability wherein external calls can manipulate the contract's state. This manipulation leads to a discrepancy between the **Controller**'s recorded **total_debt** and the actual **borrow_token.balanceOf(controller.address)**, due to token balance modifications preceding these calls without a concurrent **total_debt** update. This issue directly affects several view functions across the **Vault**, **Controller**, and **SemilogMonetaryPolicy** contracts, which rely on the **Controller**'s state for accurate information.

Vault:

**_total_assets()**, which impacts

- **lend_apr()**,
- **totalAssets()**,
- **pricePerShare()**,
- **convertToShares()**,
- **convertToAssets()**

and those reliant on **borrowed_token.balanceOf(self.controller.address)**, such as

- **maxWithdraw()**,
- **previewWithdraw()**,
- **maxRedeem()**, and
- **previewRedeem()**.

Controller:

- **total_debt()** – external
- **max_borrowable()**

SemilogMonetaryPolicy:

- **calculate_rate()** which affects:
  - **rate_write()**
  - **rate()**

These functions are susceptible to returning inaccurate values, as they can be influenced during the callback phase of external calls.

This vulnerability poses a risk by allowing key view functions to be manipulated, leading to incorrect outputs. The issue extends to third-party protocols that might potentially rely on these functions, amplifying the vulnerability across the ecosystem.

### Recommendation

1. State Synchronization: Ensure synchronization between the token balance and **total_debt** in the Controller before executing external calls. This could involve updating the **total_debt** concurrently with the token transfer, thus maintaining consistent state information.
2. Exercise Caution in Critical Operations: Protocols should exercise caution when incorporating the affected functions into critical parts of the system. It is recommended to implement thorough sanity checks before and after interacting with these functions to ensure the data integrity and consistency of the inputs and outputs.

### Client's comments

What is important is to check that this is not coming from the callback where total_debt is in the middle of the update. For that, check_lock() method is implemented in Controller. If someone donates to the controller – it is a legitimate donation which does pump the share price.

| INFORMATIONAL-08 | Gas saving on set_rates() | Fixed at 82eef6 |
|---|---|---|

### Description

The **SemilogMonetaryPolicy** contract has an external **set_rates()** function which sets new maximum and minimum thresholds for rates. If one of the boundaries does not change, the logarithm will be calculated again, which is gas-expensive.

### Recommendation

We recommend adding a check for the equality of new rates with old ones.

| INFORMATIONAL-09 | Minor gas optimisation | Acknowledged |
|---|---|---|

### Description

Line CryptoFromPool.vy#41 is redundant, since this temporary variable **p** is used only one time

### Recommendation

We recommend removing it like this

```
if COLLATERAL_IX > 0:
    p_collateral = POOL.price_oracle()
else:
    p_borrowed = POOL.price_oracle()
```

| INFORMATIONAL-10 | Comments typos | Fixed at 886b08 |
|---|---|---|

### Description

Typo in **set_admin()** description on this line.
Typo in **ln_int()** comment on this line. The argument of **math.log2** should be 10**18.

### Recommendation

We recommend fixing these typos

| INFORMATIONAL-11 | Precision loss in single loan repayment | Acknowledged |
|---|---|---|

### Description

In the **Controller._debt** function, a mechanism is implemented to mitigate precision loss by rounding up the debt calculation. However, this mechanism incorrectly assumes that rounding is only necessary when there are multiple loans, leading to potential precision loss when there is only one loan.

```
debt: uint256 = loan.initial_debt * rate_mul
if debt % loan.rate_mul > 0:
    if self.n_loans > 1:
        debt += loan.rate_mul
debt /= loan.rate_mul
```

### Recommendation

It is recommended to adjust the conditional logic to ensure that the debt calculation always rounds up.

### Client's comments

This is by design. If we do not do it - it will be impossible to repay all the minted debt in case of a stablecoin. So let's call it a design decision: if user is the only one left with the debt - his debt is rounded down

STATE
MIND