MakerDAO: Spark ALM Controller

Security Review

Cantina Managed review by:
Chris Smith, Lead Security Researcher
Xmxanuel, Lead Security Researcher

April 8, 2025

Contents

1 Introduction

1.1 About Cantina

Cantina is a security services marketplace that connects top security researchers and solutions with clients. Learn more at cantina.xyz

1.2 Disclaimer

Cantina Managed provides a detailed evaluation of the security posture of the code at a particular moment based on the information available at the time of the review. While Cantina Managed endeavors to identify and disclose all potential security issues, it cannot guarantee that every vulnerability will be detected or that the code will be entirely secure against all possible attacks. The assessment is conducted based on the specific commit and version of the code provided. Any subsequent modifications to the code may introduce new vulnerabilities that were absent during the initial review. Therefore, any changes made to the code require a new security review to ensure that the code remains secure. Please be advised that the Cantina Managed security review is not a replacement for continuous security measures such as penetration testing, vulnerability scanning, and regular code reviews.

1.3 Risk assessment

Severity	Description
Critical	*Must* fix as soon as possible (if already deployed).
High	Leads to a loss of a significant portion (>10%) of assets in the protocol, or significant harm to a majority of users.
Medium	Global losses <10% or losses to only a subset of users, but still unacceptable.
Low	Losses will be annoying but bearable. Applies to things like griefing attacks that can be easily repaired or even gas inefficiencies.
Gas Optimization	Suggestions around gas saving practices.
Informational	Suggestions around best practices or readability.

1.3.1 Severity Classification

The severity of security issues found during the security review is categorized based on the above table. Critical findings have a high likelihood of being exploited and must be addressed immediately. High findings are almost certain to occur, easy to perform, or not easy but highly incentivized thus must be fixed as soon as possible.

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

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

2 Security Review Summary

The Maker Protocol, also known as the Multi-Collateral Dai (MCD) system, allows users to generate Dai (a decentralized, unbiased, collateral-backed cryptocurrency soft-pegged to the US Dollar) by leveraging collateral assets approved by the Maker Governance, which is the community organized and operated process of managing the various aspects of the Maker Protocol.

From Mar 10th to Mar 21st the Cantina team conducted a review of spark-alm-controller on commit hash 3dd606c2. The team identified a total of **9** issues in the following risk categories:

- Critical Risk: 0
- High Risk: 0
- Medium Risk: 3
- Low Risk: 2
- Gas Optimizations: 1
- Informational: 3

The Cantina Managed team reviewed Maker's `spark-alm-controller` holistically on commit hash b618c3fb (corresponding to the tag `v1.4.0`) and concluded that all issues were addressed and no new vulnerabilities were identified.

3 Findings

3.1 Medium Risk

3.1.1 `addLiquidityCurve` `rateLimit` miscalculation for Curve pool `N > 2` enables partial limit bypass

Severity: Medium Risk

Context: MainnetController.sol#L685

Description: The fix review version included a new `rateLimit` decrease in the `addLiquidityCurve` function where this finding has occurred. If the `depositAmounts` for an `add_liquidity` Curve call are not equally distributed, internally in Curve a swap happens to equalize the `depositAmounts`.

For example, if in a balanced pool with enough liquidity the `depositAmounts` are `depositAmounts=[100, 10]`. The total value added would be `110`, and `45` units of token1 would be sold for `45` units of token2 to achieve an equal balance of `55` for both.

In the `MainnetController.addLiquidityCurve`, the `rateLimit` for `addLiquidity` is decreased after a call which limits the volume a relayer can use. In the new version, an additional `rateLimit` decrease for `swapCurve` has been added to `addLiquidityCurve` to adjust the `rateLimit` if the unequal `depositAmounts` result effectively in a swap.

This is achieved by the following logic:

```
// Compute the average swap value by taking the difference of the current underlying
// asset values from minted shares vs the deposited funds, and decrease the swap
// rate limit by this amount.
uint256 totalSwapped;
for (uint256 i; i < depositAmounts.length; i++) {
    totalSwapped += _absSubtraction(
        curvePool.balances(i) * rates[i] * shares / curvePool.totalSupply(),
        depositAmounts[i] * rates[i]
    );
}
uint256 averageSwap = totalSwapped / depositAmounts.length / 1e18;

rateLimits.triggerRateLimitDecrease(
    RateLimitHelpers.makeAssetKey(LIMIT_CURVE_SWAP, pool),
    averageSwap
);
```

The `totalSwapped` is basically the absolute difference between the initial `depositAmounts` provided by the relayer and the actual new `depositAmounts` in the Curve pool after the internal swap.

The `update` for the `rateLimit` should be the same amount as if the tokens would have been sold before by calling `swapCurve` to achieve an equal distribution in `depositAmounts`.

Dividing the `totalSwapped` by the `depositAmounts.length` is incorrect if there are more than two tokens in the pool. The `totalSwapped`, based on the absolute difference, is basically twice the sum of sold tokens. Therefore, it is only required to divide the `totalSwapped` by 2.

- Example with three tokens:

```
Input
depositAmounts = [150, 30, 60]
totalValue = $240

Curve
equally balanced at = [80, 80, 80]
the effective swap would be = [-70, +50, +20]

MainnetController
totalSwapped = 150 - 80 + 80 - 30 + 80 - 60;
totalSwapped = 70 + 50 + 20;
totalSwapped = 140;
averageSwap = 46.66;
```

However, the `averageSwap = 46.66` is not correct. The reasoning for this suggested change is that the update of the `rateLimit - LIMIT_CURVE_SWAP` should be the same as if the relayer were to first call `swapCurve`

themselves and then call addLiquidity with an even distribution. For the example above `depositAmounts = [150, 30, 60]`, this would be as if two sells of `tokenIdx=0` would happen:

- swapCurve:
  - inputIndex: 0.
  - outputIndex: 1.
  - amountIn: 50.
- swapCurve:
  - inputIndex: 0.
  - outputIndex: 2.
  - amountIn: 20.

Recommendation: The correct calculation for the approximated update should be:

```
- uint256 averageSwap = totalSwapped / depositAmounts.length / 1e18;
+ uint256 effectiveSwap = totalSwapped / 2 / 1e18;
```

Maker: Fixed in commit 7762501.

Cantina Managed: Fixed.

3.1.2 `maxSlippage` is insufficient swap protection for unbalanced Curve Pools

Severity: Medium Risk

Context: MainnetController.sol#L572-L583

Description: Currently the Curve `swap` function utilizes a `maxSlippage` protection. This value is set by the `DEFAULT_ADMIN_ROLE` to a percentage (i.e. `0.98e18` or `1.02e18`). It is then used as a check against the `RELAYER` during the swap to ensure the `RELAYER` specifies an acceptable `minAmount` parameter when doing the swap. It always specifies that the `minAmount` parameter must be `amountIn * rates[inputIndex] / rates[outputIndex] * maxSlippage` so with 1:1 priced tokens and max slippage set to 98% the `minAmount` would have to be 98 or greater when `amountIn = 100`. If all the same conditions existed, but max slippage was 102%, the `minAmount` would have to be 102.

Essentially `maxSlippage` provides a floor to ensure that the `RELAYER` is not making trades that incur more slippage than is acceptable.

Curve uses slippage to punish/reward making the Pool less/more healthy. So a trade that significantly moves the pool away from equilibrium will incur more slippage than one that does not substantively affect the equilibrium. If the pool is already imbalanced between the tokens, then trades that move it towards equilibrium will be rewards. This is done to encourage arbitragers to keep the stable pools in balance.

This graph helps us visualize how slippage works:

- Point (5,5) represents a balanced pool, trading around that point would incur little slippage in either direction.
- Point (~2,~9), The Green Dot, represents an out of balance pool with too much of asset X.
- Point (~9,~2), The Red Dot, represents an out of balance pool with too much of asset Y.
- Trading from equilibrium to either dot will incur slippage such that the `amountOut` will be less than the `amountIn`.
- Trading from either dot will incur slippage such that `amountOut` will be greater than the `amountIn`.
- We assume the red shaded area represents invalid trades for the spark-controller (based on `maxSlippage`) if the trade moves from any point on the curve near equilibrium to any point further from equilibrium.

It is likely that the Spark system will be a large participant in the pools the spark-controller is interacting with, which means that it is reasonable to consider the pools to be a relatively "low liquidity" environment, relative to the potential size of the trades. This will be bounded some by the Spark `RateLimit` contract, but moving the market with swaps should be considered likely for the spark-controller swaps.

Pairing sDAI and other Maker/Sky stable coins with stable coins in the Curve Stable Pools will have impact on the price peg of the Maker/Sky stable coins and thus, it should be desirable that the spark-controller system ideally improves the health of the Curve pools or at least does not harm it.

It should also be desirable, as partially expressed by the `maxSlippage`, that the swaps done by the `RELAYER` should benefit or at least not harm the spark-controller system (i.e. do not execute trades that cost the system "too much").

With this background in place, we can evaluate the efficacy of the `maxSlippage` protection with a couple of scenarios.

For ease of description, we will make a couple of assumptions:

- `maxSlippage` is `0.98e18` unless otherwise stated.
- The `store_price` function returns a 1:1 price pairing for the assets unless otherwise noted (so the only deviation from 1:1 pricing is due to Curve pool slippage).
- the pool is in equilibrium at the point (5,5).
- We are ignoring Curve fees for this analysis (so the only deviation from 1:1 pricing is due to Curve pool slippage).

Scenario 1: Imbalanced pool:

In this scenario, we assume that regular market forces have pushed the Pool into an imbalanced state, lets assume that current balance between (x,y) is skewed towards asset Y into the red zone past the green dot. Any trade that moves the balance towards equilibrium will be rewarded. Because our current pool balance is in the red zone, any `RELAYER` swap from asset Y to asset X will be blocked by `maxSlippage` because it would be more than a 2% discount. However, almost any `RELAYER` swap from asset X to asset Y will be accepted by the Controller since it would have a slippage above -2%.

If the `RELAYER` swaps enough asset Y to asset X to move the pool back into equilibrium, then it would be rewarded a bonus in `amountOut`. However, if the `RELAYER` trades even more of asset Y for asset X, such that the pool starts to be imbalanced in the other direction, the spark-controller will allow this too. Assuming a swap that leaves the pool containing more asset X than asset Y, the swap would still have slippage such that the `amountOut` is greater than the `amountIn`, but this second `amountOut` would be **less** than the first, balancing trade's `amountOut` because the Curve pool would reward it up to the equilibrium point, but then start punishing it for the portion of the trade that moves beyond that. The `RELAYER`'s swap could entirely flip the pool with a trade that does not violate the `maxSlippage` protection effectively it would have an end state that is the reverse of the initial state. Visually we could say this trade moved all the way from the Green dot to the Red dot (or visa versa).

If the Curve pool is filling up with Asset X, it is likely that Asset X is loosing its value relative to Asset Y and because this swap changes the balance of the pool such that it ends up with more Asset Y, the spark-controller will accumulate Asset X, thus putting itself at risk of "*holding the bag*". Though if the price of Y is too high in Curve compared to other markets, the `RELAYER` may see an arbitrage opportunity.

Even setting aside this sort of more extreme market situation, allowing the `RELAYER` to move the market from one Dot to another is not the most efficient, profitable trade. The spark-controller is missing out on the slippage reward it would have received by moving the market from an imbalanced state to a balanced one. This lost profit is roughly `amountIn * (percent market is imbalance + 100e18 - maxSlippage)`.

Scenario 2: De-pegging asset + Other pool whale.

This scenario shares some similar aspects to the first scenario, but includes a 3rd party actor who is also a large liquidity provider. In this scenario, we can assume there is a major de-peg event that is just starting. The pool is still relatively in equilibrium. Seeing a trade from the `RELAYER`, the 3rd party calls `remove_liquidity_one_coin` to pull all their liquidity out in the form of **asset X**. Assuming they are large enough, this could swing the pool to be imbalanced with too much asset Y. This 3rd party would lose a quantity of asset X tokens due to slippage on this removal. If the reduction in quantity is less than the predicted reduction in value for asset X during a black swan de-peg and/or less than the cost of pulling out asset Y, the actor might choose to accept this loss. Further, arbitraging the pool back to equilibrium after the

RELAYER's swap could help off set some of these costs. The RELAYER's trade could then swing the pool from a state where it had too much asset Y to one where it had too much asset X. The 3rd party could then use the received asset X to arbitrage the pool back to equilibrium, swapping their soon to be worth less asset X for asset Y. Lets outline the steps here:

```
1. The X/Y pool is at a 1:1 ratio.
2. A third-party liquidity provider decides to `remove_liquidity_one_coin` due to market conditions, removing
↪  X, which is not yet reflected in the Curve pool.

    - Removing only one coin in large amounts will change the price and result in extra fees compared to
    ↪  removing both coins equally (the standard removal function).

3. The pool now contains more Y than X, meaning X has a higher price. (removing X here would be similar to
↪  buying X with Y).
4. The relayer wants to sell X or Y and is front-run by step 3.
    - If selling Y for X, the relayer's swap would be blocked because of the `maxSlippage` constraint, so only
    ↪  possible to sell X.
    - Selling X for Y, if the price after the trade returns to 1:1, the relayer would have made more money than
    ↪  without the front-running.
    - However, if the relayer pushes the price in the other direction, the overall trade would still satisfy
    ↪  the `maxSlippage` constraint but with even less money returned for the swap.
```

Recommendation: At a minimum, RELAYERs should have off-chain calculations to ensure they are swapping an amount that results in the "right amount" of slippage and setting a `minAmount` appropriately. Spark should also monitor RELAYER swaps to ensure they are performed in the correct way.

Onchain validation could look at the Pool balances before and after to determine whether a trade is making it healthier or less healthy as that is a measure for both whether the swap receives closer to the optimal amount of slippage and/or if the Pool is being utilized in such a way as to support the stability of DAI by pairing it in a healthy way with other stable coins.

Maker: Acknowledged. After internal discussion we've decided not to add any additional onchain validation for these swaps. As long as the slippage value does not cause value lost in NAV terms above the slippage amount it will be considered a valid trade. Offchain components will analyze swaps before performing to ensure that they are profitable so this case will not occur unless a RELAYER is compromised, in which case they will be rate limited and can be frozen if necessary.

Cantina Managed: Acknowledged.

3.1.3 Incorrect slippage protection in `addLiquidityCurve` and `removeLiquidityCurve` due to missing virtual price conversion

Severity: Medium Risk

Context: MainnetController.sol#L642, MainnetController.sol#L682

Description: The `addLiquidityCurve` and `removeLiquidityCurve` functions contain incorrect slippage protection checks that compare LP token amounts directly with token values without accounting for the LP token's virtual price. In `addLiquidityCurve`, the function checks:

```
require(
    minLpAmount >= valueDeposited * maxSlippage / 1e18,
    "MainnetController/min-amount-not-met"
);
```

This is incorrect because `minLpAmount` represents LP tokens (shares of the pool), while `valueDeposited` represents the total value of the tokens that should be deposited into Curve. These values cannot be directly compared without converting the LP tokens to their underlying value using Curve's `get_virtual_price()` function. Similarly, in `removeLiquidityCurve`, the function checks:

```
require(
    valueMinWithdrawn >= lpBurnAmount * maxSlippage / 1e18,
    "MainnetController/min-amount-not-met"
);
```

Here, `valueMinWithdrawn` represents token values, while `lpBurnAmount` represents LP tokens. Again, these cannot be directly compared without accounting for the virtual price.

Impact: These errors currently have the following impact assuming the `virtual price` only increases starting from `1e18`.

1. For `addLiquidityCurve`: The check is too strict. It will reject transactions with acceptable slippage levels because it's comparing LP tokens directly with token values. This is because after multiplying `minLpAmount` by the virtual price (which is > 1e18), the left side of the comparison would be greater, therefore making the check harder to satisfy than intended.
2. For `removeLiquidityCurve`: The check is too soft. It will allow transactions with excessive slippage to pass because it's comparing token values with LP tokens. This allows a lower `valueMinWithdrawn` to pass than should be permitted.

Recommendation: For `addLiquidityCurve`, convert the LP tokens to their value equivalent:

```
require(
    minLpAmount * curvePool.get_virtual_price() / 1e18 >= valueDeposited * maxSlippage / 1e18,
    "MainnetController/min-amount-not-met"
);
```

For `removeLiquidityCurve`, the correct check would be:

```
require(
    valueMinWithdrawn >= lpBurnAmount * curvePool.get_virtual_price() * maxSlippage / 1e36,
    "MainnetController/min-amount-not-met"
);
```

Note: The division by `1e36` in the second case accounts for the fact that `get_virtual_price()` returns a value with 18 decimals, and we're also multiplying by `maxSlippage` which is in `1e18` format.

Maker: Fixed in commit 4d63f59.

Cantina Managed: Fixed.

3.2 Low Risk

3.2.1 `maxSlippage` mechanism prevents selling `depegged` tokens via `swapCurve`

Severity: Low Risk

Context: MainnetController.sol#L581

Description: The `maxSlippage` parameter ensures that the `minAmountOut` value passed by the relayer is always greater than or equal to a `minimumMinAmountOut`. This `minimumMinAmountOut` calculation incorporates a `maxSlippage` based on a 1:1 exchange rate between two assets.

However, if one token has depegged, it may become impossible to sell that token. While this might be the desired behavior in some cases, there are scenarios where allowing the token sale would be preferred by all parties.

For example, let's say we have another stablecoin, S, and there is a DAI/S Curve pool. Suppose S is already trading at 0.92. However, S is in a death spiral like Terra/Luna, and its price is expected to decrease further. For some reason, Sky is holding some S tokens and decides they want to exit their position, regardless of the price; however, `maxSlippage` would prevent this.

Recommendation: Consider implementing an emergency mechanism that would allow selling tokens below the configured `maxSlippage` during extreme market events.

Note that adding such a mechanism introduces new trust assumptions, so carefully consider the governance implications and whether the added flexibility justifies the increased centralization risk.

Maker: We have determined that adding this new functionality does indeed introduce new trust assumptions that might make the system less secure as a whole. We will leave the functionality as is.

Cantina Managed: Acknowledged.

3.2.2 Precision loss in `addLiquidityCurve` and `removeLiquidityCurve` when calculating `valueDeposited` and `valueWithdrawn`

Severity: Low Risk

Context: MainnetController.sol#L636

Description: In `addLiquidityCurve`, it is required to calculate the total value deposited for the maxSlippage check and for updating the `rateLimits`.

```
uint256 valueDeposited;
for (uint256 i = 0; i < depositAmounts.length; i++) {
    _approve(curvePool.coins(i), pool, depositAmounts[i]);
    valueDeposited += depositAmounts[i] * rates[i] / 1e18;
}
```

Therefore, the `depositAmounts[i]` is multiplied with the `stored_rates` from Curve. The `stored_rates` are not only used for scaling to the correct precision, they also can include an oracle price for assets of type X.

In the `stored_rates` function, the `oracle_price` is multiplied with `scale_factor`.

This means the actual `stored_rates` can have values like `1123456789012345678` without trailing zeros at the end (example is for a standard token with a `1e18` precision).

Multiplying the `stored_rates` with a `depositAmount` and afterwards dividing it by `1e18` can lead to a precision loss in some cases.

- Example:

```
function testPrecision() public {
    // scaleFactor for standard 1e18 precision token
    uint scale_factor = 1e18;
    // oracle price in 1e18
    uint price = 1123456789012345678;
    // curve.stored_rates
    console.log("oracle_price :", price);
    console.log("scale_factor  :", scale_factor);
    uint stored_rate = price * scale_factor / 1e18;
    console.log("stored_rate  :", stored_rate);
    // alm controller
    // depositAmount in 1e18
    uint depositAmount = 111_111111111111111111;
    console.log("depositAmount:", depositAmount);
    console.log("result 1e36: :", depositAmount * stored_rate);
    console.log("result 1e18: :", depositAmount * stored_rate / 1e18);
}
```

Logs:

```
Logs:
oracle_price : 1123456789012345678
scaleFactor  : 1000000000000000000
depositAmount: 111111111111111111111
stored_rate  : 1123456789012345678
result 1e36: : 124828532112482853111098628257899862825 8
result 1e18: : 12482853211248285311 10
```

In the current implementation, it is always a `round-down` operation. However, since the `valueDeposited` is used to decrease the `rateLimit`, it makes more sense to round up for such a limit.

```
// Reduce the rate limit by the aggregated underlying asset value of the deposit (e.g. USD)
  rateLimits.triggerRateLimitDecrease(
      RateLimitHelpers.makeAssetKey(LIMIT_CURVE_DEPOSIT, pool),
      valueDeposited
  );
```

The conversion to `1e18` precision should also happen after the for loop. Because adding together multiple `valueDeposited` in `1e36` precision can result in multiple full units of `1e18` precision.

Impact: Since `valueDeposited` is used to decrease rate limits, consistently rounding down could allow relayers to deposit slightly more value than intended by the rate limit mechanism.

The same issue occurs in the `removeLiquidityCurve` function as well by calculating the total `valueWithdrawn` after the curve call:

```
// Aggregate value withdrawn to reduce the rate limit
uint256 valueWithdrawn;
for (uint256 i = 0; i < withdrawnTokens.length; i++) {
    valueWithdrawn += withdrawnTokens[i] * rates[i] / 1e18;
}
```

Recommendation:

1. Accumulate the full-precision sum before dividing by `1e18`:

```
uint256 valueDeposited;
for (uint256 i = 0; i < depositAmounts.length; i++) {
    _approve(curvePool.coins(i), pool, depositAmounts[i]);
    valueDeposited += depositAmounts[i] * rates[i];
}
// Divide once after accumulating all the `valueDeposited` with for example OpenZeppelin's Math library
valueDeposited = Math.mulDiv(valueDeposited, 1, 1e18, Math.Rounding.Ceil);
```

2. This ensures that any fractional amounts are always rounded up, preventing potential rate limit bypass through precision loss.

The same recommendation can be implemented for `removeLiquidityCurve` as well. However, in the `removeLiquidityCurve` there are two different total sum calculations:

1. For the slippage check, the current implementation (rounding down) is appropriate in `valueMinWithdrawn`.
2. For rate limit tracking, the recommendation to accumulate full precision and then round up is valid for `valueWithdrawn`.

Maker: Division outside of the loop fixed in commit 07f4b85.The impact of rounding up is negligible.

Cantina Managed: Fixed and Acknowledged.

3.3 Gas Optimization

3.3.1 Slight gas savings for `addLiquidityCurve`

Severity: Gas Optimization

Context: MainnetController.sol#L635

Description: Since the relayer has to pass an array of tokens whose length is the same as the pool's COINS, some `depositAmounts` might be 0.

Recommendation: It would be a slight improvement to only call `_approve` if `depositAmounts[i] > 0`.

Maker: Will keep as-is for the sake of simplicity.

Cantina Managed: Acknowledged, savings would be minimal and function call is probably not super frequent

3.4 Informational

3.4.1 `RateLimit` can possibly be more restrictive than necessary

Severity: Informational

Context: MainnetController.sol#L585-L588

Description: The current design for the `RateLimit` for `curveSwap` does not distinguish directionality nor does it specify which coins. This could result in over-limiting the `swap` function since swapping from `Token A` → `Token B` also limits swaps from `Token B` → `Token A` and from `Token C` → `Token D` (in a multi-coin pool).

Recommendation: Consider if this behavior is desirable or if a less broad rate limit is more appropriate.

Maker: Acknowledged. We will keep this rate limit as is and if we determine that we need to make it more granular in the future we can add that in a later release.

Cantina Managed: Acknowledged.

3.4.2 `minAmountOut >= minimumMinAmountOut` check in `swapCurve` is too strict for some edge cases

Severity: Informational

Context: MainnetController.sol#L581

Description: For a `swapCurve` call, the relayer can specify a `minAmountOut` parameter which will be passed to the `curve.exchange` call. If the `amountOut` would be less than the `minAmountOut`, the curve call would revert.

However, the admin of this contract can limit the freedom of the `minAmountOut` with a `maxSlippage` parameter, which is used to calculate a `minimumMinAmountOut`.

This is essentially a lower bound for the `minAmountOut`. The `minAmountOut` needs to be greater than or equal to the `minimumMinAmountOut`.

```
require(
    minAmountOut >= minimumMinAmountOut,
    "MainnetController/min-amount-not-met"
);
```

It is important to notice that the `minAmountOut` defines a minimum for the curve trade and the actual `amountOut` can be higher. Therefore, the relevant variable is the actual `amountOut` because it will contain the actual slippage.

Recommendation: If the actual `amountOut >= minimumMinAmountOut` all constaints would be satified. Consider, adding this check after the trade. This would enable trades which are currently blocked. For example:

```
minAmountOut = 95 Dai
minimumMinAmountOut = 97 Dai
amountOut = 98
```

In the current design these call would revert because `minAmountOut >= minimumMinAmountOut` is not satisfied even when the actual `amountOut` is higher than the `minimumMinAmountOut`.

Maker: Acknowledged. We understand the point about comparing to actual `amountOut` vs `minAmountOut`. However, keeping it as-is ensures that relayers must explicitly set an acceptable `minAmountOut` value. This protects against MEV bots that would exploit misconfigured `minAmountOut` values.

For example, if a relayer accidentally passes 0 for `minAmountOut`, the current implementation always reverts. If we compared to `amountOut` instead, a transaction could proceed with the minimum slippage, allowing MEV bots to extract value up to that threshold. The current implementation provides stronger safety guarantees against relayer configuration errors.

Cantina Managed: Acknowledged.

3.4.3 `swapCurve` doesn't validate the `inputIndex` and `outputIndex` for an early revert

Severity: Informational

Context: MainnetController.sol#L552

Description: The `swapCurve` function has the parameters `inputIndex` and `outputIndex` which are the indexes for the swap in Curve. However, these parameters are not immediately validated. The indexes could be higher than the number of tokens in the Curve pool, and `inputIndex` should not equal `outputIndex`.

Recommendation: Consider adding these sanity checks to revert early.

Maker: Fixed in commit be430d3.

Cantina Managed: Fixed.