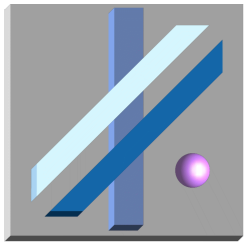

Fixed Point Solutions, LLC

B.AMM Protocol Liquity Integration Assessment

Originally published: 2021/07/07
Updated: 2022/01/31
Prepared by: Kurt Barry

Scope

The B.AMM protocol has been integrated with the Liquity protocol to provide opt-in automatic rebalancing of funds in Liquidty's stability pool following liquidations. This is done via an AMM that sells ETH from liquidations for LUSD at a discount to the current ChainLink oracle price. The integration also proportionally distributes LQTY incentives to B.AMM LPs.

Fixed Point Solutions was retained to audit the B.AMM Liquity-specific smart contracts found in https://github.com/backstop-protocol/dev/tree/main/packages/contracts/contracts/B.Protocol at commit hash 4d82e3ef011df10930064f2e11e4c2219a5e8f48. The main focus was economic and smart contract security. Roughly 20 person-hours of effort were committed.

Security assumptions taken for the audit include:

- The ChainLink oracle price is trusted.
- The ETH/USD price is assumed to accurately reflect the ETH/LUSD price.
- The Liquity liquidation discount (10%) is assumed to always be much larger than any deviation of the oracle price from the market price.

Update: in early 2022, LUSD experienced a depegging event, violating the second security assumption above. The following commit, intended to allow the B.AMM to rebalance even in such case, was reviewed: https://github.com/backstop-protocol/dev/pull/36/commits/3059a3a09f6cfb11113f90627921c7465372cebe

See the section titled "Addendum 1", for notes from the review of this commit.

Findings

Findings and recommendations are listed in this section, grouped into broad categories. It is up to the team behind the code to ultimately decide whether the items listed here qualify as issues that need to be fixed, and whether any suggested changes are worth adopting. When a response from the team regarding an issue is available, it is provided.

Security and Correctness

SC.1 - ETH Sent to Contract Prior to First Deposit Makes Withdrawals and Reward Claims Impossible

https://github.com/backstop-protocol/dev/blob/4d82e3ef011df10930064f2e11e4c2219a5e8f48/packages/contracts/contracts/B.Protocol/BAMM.sol#L109

Team Response: Added logic to prevent this in commit 3459b04.

Description: In BAMM.deposit(), for the `newShare` variable to be non-zero, either `totalValue` must be zero or the `total` state variable must be non-zero. If some amount of ETH with a USD value that does not round to zero is sent to the contract prior to the first deposit (this is possible due to the non-reverting `receive()` function, and need only be a tiny amount at current prices), then all attempts to deposit will result in `newShare` equal to zero indefinitely, because `totalValue` will henceforth always be nonzero, while `total` will remain at its initial value of zero indefinitely since it can only be increased if `newShare` is ever non-zero. The consequences are that `withdraw()` will always fail (due to an underflow-triggered revert) and attempts to claim LQTY rewards will not transfer any tokens (a result of `total` remaining zero and the `harvest()` logic of `CropJoin`). This griefing attack is cheap and could be high impact if users make large deposits that are lost, but it is not strongly incentivized and is easily preventable by ensuring that at least one valid deposit is made into each deployed `BAMM` instance before ETH is sent to it (e.g. atomically in the creation transaction, or just prior to advertising it in a UI). The `deposit()` function could also be modified to eliminate this possibility with special-case logic for `totalValue > 0 && total == 0`.

Usability

U.1 - No Slippage Protection on Swap Function

https://github.com/backstop-protocol/dev/blob/4d82e3ef011df10930064f2e11e4c2219a5e8f48/packages/contracts/contracts/B.Protocol/BAMM.sol#L187

Team Response: Added a parameter for slippage protection in commit 3459b04.

Description: While the `getSwapEthAmount()` logic will limit the discount given to traders based on the `maxDiscount` parameter, there is no limit on slippage that is unfavorable to the trader. In particular, any time the ETH amount to purchase exceeds the available ETH, the purchase amount is set to be the available ETH with no modification to the amount of LUSD taken as payment. This could lead to scenarios such as two traders attempting to purchase the total ETH available simultaneously, with one winning and the other paying the same amount of LUSD as the first but receiving nothing in return. Depositors in the pool are in fact incentivized to front-run swaps as they benefit from such misexecution. Block producers (be they miners in PoW or validators in PoS) that are also depositors in particular have a strong combination of incentive and ability. Traders can protect themselves by creating their own proxy contracts to bid through that enforce slippage limits. However, this makes integrating with the protocol more complicated and expensive than necessary, and potentially reduces its rate of adoption by the DeFi ecosystem. Thus, it may be beneficial to add slippage protection, most simply in the form of an additional argument used to put a lower bound on the amount of ETH purchased. A more complex option would be to dynamically adjust the amount of LUSD taken from the trader if the available ETH is less valuable than the provided LUSD amount.

Gas Optimizations

G.1 - `CropJoin` Simplifications

https://github.com/backstop-protocol/dev/blob/4d82e3ef011df10930064f2e11e4c2219a5e8f48/packages/contracts/contracts/B.Protocol/crop.sol

Description: Given that only limited functionality of `CropJoin` is being used, the contract could be simplified further to reduce gas costs--in particular, by removing the `vat`, `ilk`, `dec`, and `gem` variables and all lines of code associated with them. The events could be eliminated as well, since the values they log are covered by events emitted in `BAMM.deposit()` and `BAMM.withdraw()`.

G.2 - Numerical Method Convergence Conditions

https://github.com/backstop-protocol/dev/blob/4d82e3ef011df10930064f2e11e4c2219a5e8f48/packages/contracts/contracts/B.Protocol/PriceFormula.sol#L26
https://github.com/backstop-protocol/dev/blob/4d82e3ef011df10930064f2e11e4c2219a5e8f48/packages/contracts/contracts/B.Protocol/PriceFormula.sol#L47

Team Response: Changed the conditions to be equivalent to Curve's logic in commit 3459b04. Will not pursue maximal optimization as ROI is likely to be low given that these functions are already quite gas-intensive.

Description: The conditions for convergence of the two numerical methods are both of the form `current_value <= previous_value.add(1) && previous_value.add(1) <= current_value`, which is equivalent to just `current_value == previous-value.add(1)`.

The most straightforward improvement is that the conditions could be simplified to the latter form to use less gas.

Less obviously, this condition differs subtly from the one that is implemented in the Curve contracts (example), which use statements mathematically equivalent to `| current_value - previous_value | <= 1`. Specifically, one can imagine a convergence scenario to a true fixed point F for either loop where for no i is the ith value one less than the (i+1)th value--e.g. stepping straight from F+1 or F-2 to F, then re-obtaining F repeatedly. The consequence of such a coincidence (for either loop) would be that the loop would run for a full 255 iterations unnecessarily--so a correct result would be obtained but with a significant waste of gas. It is unclear whether such a coincidence is numerically possible from qualitative inspection. The existing condition could be made equivalent to Curve's by changing it to `current_value <= previous_value.add(1) && previous_value <= current_value.add(1)`.

Generally, if gas is a significant concern, various forms of the condition should be tested to see which has the best average-case performance. E.g. something like the following (which allows using unchecked mathematical operations) might perform the best:

```
if (previous_value < current_value) {
    if (current_value - previous_value <= 1) break;
} else {
    if (previous_value - current_value <= 1) break;
}
```

Miscellaneous

M.1 - Unnecessary `payable` modifier on `swap()`

https://github.com/backstop-protocol/dev/blob/4d82e3ef011df10930064f2e11e4c2219a5e8f48/packages/contracts/contracts/B.Protocol/BAMM.sol#L187

Team Response: Fixed in commit 3459b04.

Description: The `swap()` function shouldn't need to be `payable`--it doesn't receive ETH.

Notes

This section contains general considerations for interacting with the system and various conclusions reached or discoveries made during the course of the assessment. Whereas findings generally represent things for the team to consider changing, notes are more informational and intended to be interesting also for anyone who might interact with the system.

N.1 - Resistance to Manipulation

Since the `swap()` function checks the StableSwap invariant but the `deposit()` and `withdraw()` functions do not, it is natural to consider whether malicious sequences of deposits, withdrawals, and swaps might allow draining value from the contract, possibly by using flash loans. It was found that under the assumption that the oracle price always accurately represents the ETH/LUSD market price (the case in which it does not is considered below), harmful manipulation seems to be impossible, for the following reasons.

1. Deposits must be single-sided in LUSD.
2. Withdrawals are always balanced between LUSD and ETH based on the current dollar value of each held by the system, so withdrawals do not affect the degree of imbalance in the pool.
3. The `swap()` function is unidirectional--one can only swap LUSD for ETH, and not vice-versa. This prevents a sequence like deposit LUSD → sell ETH at a favorable exchange rate → withdraw full original value. LUSD-only deposits thus only worsen the execution price for traders.
4. The `swap()` function limits unfavorable (for the protocol) slippage to be no more than the Liquity liquidation discount--this prevents flash loan attacks where an attacker might be able to purchase with a higher gain than their loss from liquidation.

N.2 - Oracle Risk Considerations

Since the current ChainLink oracle price is treated as trusted by the `BAMM` contract, an erroneously low price in excess of the profit margin from Liquity liquidations can result in losses for users (ETH sold too cheaply), and an erroneously high oracle price can prevent rebalancing. The security model assumed in this assessment takes as given that the oracle price error will remain well within the profit margin of Liquity liquidations, but users should be aware that this may not necessarily always hold in practice (e.g. in the event of successful manipulation of ChainLink prices) and gauge their risk accordingly. It is further assumed here that the ETH/USD market price accurately reflects the ETH/LUSD market price, another condition which is not guaranteed to hold in practice.

A concern within the threat model is whether an oracle price that deviates within the assumed margin of error might allow greater profits to be obtained via some deposit/withdrawal manipulation instead of swaps. This does not seem to be possible. First of all, no such manipulation is profitable if the oracle price is above the market price--only when it is below. In that case, one can deposit and then withdraw for a net-USD profit. A simple calculation shows that this is equivalent to a swap of the manipulator's LUSD to ETH at exactly the current oracle price. Since the AMM component generally sells ETH at a discount to the oracle price, it will nearly always be more profitable to call `swap()` than to perform this manipulation. The one possible exception is if the trading fee is larger than the effective swap discount, but that is a negligible risk unless the fee is set extremely high. This result is consistent with claims made by the team prior to this assessment.

N.2 - StableSwap Implementation

The StableSwap implementation seems to match the formulas presented in the Curve whitepaper under the transformation $2A \rightarrow A$ (ignoring unavoidable inaccuracies from performing calculations at finite numerical precision). Specifically, the Newton-Raphson method is applied to find numerical solutions for both the fixed-point sum value and the post-swap balance of a purchased asset. The calculations are structured in a way that should not have significant issues with either phantom overflow or precision loss in the likely numerical ranges for the function inputs.

Addendum 1

In January 2022, the LUSD market price sustained a level greater than $1 for significant periods of time, preventing the B.AMM from selling ETH for LUSD because the price offered was based on an assumption that 1 LUSD = $1, making swaps unprofitable. This was fixed by adjusting the ETH return of the swap function to be higher by the LUSD/USD ratio given by a Chainlink price feed for LUSD (only when this ratio exceeds one, i.e. no correction in the sub-$1 regime, where swaps instead give a deeper discount, which will not prevent rebalancing). Additionally, protocol fees are now assessed in LUSD and drawn from the pool, rather than the trader, to avoid affecting the price of trades directly.

These changes were made in the following commit:
https://github.com/backstop-protocol/dev/pull/36/commits/3059a3a09f6cfb11113f90627921c7465372cebe

Fixed Point Solutions reviewed this commit, and found the changes to be sound for achieving the intended goal. The following findings are noted:

- (Low Risk) L181, L182: `10 ** chainlinkDecimals` expressions could overflow if the `chainlinkDecimals` value fetched is anomalously high (implies a compromise of or bug in the Chainlink contracts). **Response**: Liquity makes the same assumption, oracle is assumed to be reliable.
- (Low Risk) L178: Cast of `answer` to `uint` could underflow if `answer` is negative. This again implies either a compromise of or bug in the Chainlink contracts. **Response**: Liquity makes the same assumption, oracle is assumed to be reliable.
- (Low Risk) The `maxReturn` limitation is more restrictive in a >$1 regime, since it does not correct for the LUSD value (i.e. it is equivalent to a lower `maxDiscount` parameter). This might deter rebalancing in some edge cases (e.g. very large LUSD dollar values).
- (Low Risk) Since the LUSD price is not taken into account on deposits, it will be unattractive to deposit liquidity when the LUSD price is above $1. Further it might be attractive to withdraw and sell LUSD under the same conditions. Passive LPs are unlikely to care much about the latter, and the former is only an issue if LUSD is consistently priced above $1 by the market, rather than transiently.
- (Informational) L180: Comment mentions USDC when USD is intended. **Response**: Fixed.