PUBLIC

Code Assessment

of the BAMM Smart Contracts

June 19, 2024

Produced for Frax Finance

by

Contents

1 Executive Summary

Dear Frax Finance,

Thank you for trusting us to help you with this security audit. Our executive summary provides an overview of subjects covered in our audit of the latest reviewed contracts of BAMM according to Scope to support you in forming an opinion on their security risks.

Frax Finance implements BAMM, a Borrow AMM, that wraps Fraxswap LP tokens and allows users to borrow the two underlying assets of the pair.

The most critical subjects covered in our audit are the system's solvency, the precision and correctness of arithmetic operations and oracle manipulation resistance. We found that the security of the former two topics is high. Oracle manipulation resistance is high, especially since the `BAMM` does not rely on an oracle as a traditional Lending protocol would, however, we emphasize the costs and risks of oracle manipulation in Oracle manipulation on FIFO L2s.

Other general subjects covered are rounding direction correctness and denial of service. We found that the rounding direction has generally been implemented correctly and only minor denial of service patterns were found and documented in Denial of Service against liquidations and Denial of Service against redeeming and executing actions.

Frax Finance has been very responsive to our findings and has addressed most of the issues we reported. The remaining issues are minor and do not pose a significant problem.

In summary, we find that the codebase provides a high level of security.

It is important to note that security audits are time-boxed and cannot uncover all vulnerabilities. They complement but don't replace other vital measures to secure a project.

The following sections will give an overview of the system, our methodology, the issues uncovered and how they have been addressed. We are happy to receive questions and feedback to improve our service.

Sincerely yours,

ChainSecurity

1.1 Overview of the Findings

Below we provide a brief numerical overview of the findings and how they have been addressed.

Critical-Severity Findings	0
High-Severity Findings	0
Medium-Severity Findings	2
• Code Corrected	2
Low-Severity Findings	8
• Code Corrected	6
• Risk Accepted	2

2 Assessment Overview

In this section, we briefly describe the overall structure and scope of the engagement, including the code commit which is referenced throughout this report.

2.1 Scope

The assessment was performed on the source code files inside the BAMM repository based on the documentation files. The table below indicates the code versions relevant to this report and when they were received.

dev-frax-bamm

V	Date	Commit Hash	Note
1	18 April 2024	10de29e545a5e68735785f302103190f2636dad5	Initial Version
2	20 May 2024	ca3dba642e7e93e63cdbf1b9c89b57c54dd40ace	Version 2
3	14 June 2024	876080e6ae0fd5d364e4b1a39bb2407820a57deb	Version 3

dev-fraxswap-v3

V	Date	Commit Hash	Note
1	28 March 2024	7b4826fcfc1d8bddb07dfebfe7e973be934ad048	Initial Version

For the solidity smart contracts, Solidity 0.8.23 is specified. As no EVM version is fixed, the project compiles to Shanghai, Solidity 0.8.23 default EVM version.

The following solidity files were considered in scope:

From the `dev-frax-bamm` repository:

- src/contracts/BAMMERC20.sol
- src/contracts/BAMM.sol
- src/contracts/factories/BAMMFactory.sol
- src/contracts/FraxswapOracle.sol
- src/contracts/interfaces/IRateCalculatorV2.sol
- src/contracts/interfaces/IVariableInterestRateV2.sol
- src/contracts/libraries/BitMath.sol
- src/contracts/libraries/FixedPoint.sol
- src/contracts/libraries/UQ112x112.sol
- src/contracts/VariableInterestRate.sol

From the `dev-fraxswap-v3` repository, only the `FraxswapRouterMultihop.sol` was considered. It was not fully reviewed, only its interaction with BAMM, whether it is functioning as expected in the context of BAMM swapping tokens, as well as ensuring that no value is leaking during the interactions was considered.

2.1.1 Excluded from scope

Anything not cited above was considered out of scope. Economic model of the project and the sanity of parameters is out of scope as well.

2.2 System Overview

This system overview describes the initially received version (Version 1) of the contracts as defined in the Assessment Overview.

Furthermore, in the findings section, we have added a version icon to each of the findings to increase the readability of the report.

Frax Finance offers BAMM, a Borrow AMM, that wraps Fraxswap LP tokens and allows users to borrow the two underlying assets of the pair. BAMM specificity relies on the fact that the loan-to-value ratio of the position is independent of the underlying borrowed assets' price, and the only way for a vault to be insolvent is because of the accumulated interest.

2.2.1 BAMM

The BAMM is designed to be interacted with by 3 different classes of users:

- **Liquidity providers** (LPs) who provide liquidity to the BAMM by depositing Uniswap/Fraxswap LP tokens. Their share of the liquidity is accounted for by minting BAMM tokens. LPs can redeem LP tokens by burning their BAMM tokens.
- **Borrowers** can borrow the underlying assets of the LP token by overcollateralizing their vault and unwrapping the LP tokens into the underlying assets. Borrowers can borrow or repay loans, add or remove tokens from their vault and swap them into the underlying pool.
- **Liquidators** can liquidate vaults that are undercollateralized by swapping the vault's tokens in the underlying pool to balance the vault before wrapping them into LP tokens, effectively closing the position. Liquidators get a portion of the liquidated position for their action.

2.2.1.1 Liquidity providers:

The two functions that can be used by liquidity providers are `mint()` and `redeem()`. Before anything, in both functions, interests are accrued in the system by calling `_addInterest()` and the underlying pool is synced to execute TWAMMs with `pair.sync()`.

- `mint()`: Given an amount of LP tokens, transfer the tokens from the message sender to the contract and mint the corresponding amount of BAMM tokens to the specified recipient.
- `redeem()`: Given an amount of BAMM tokens, burn the tokens from the message sender and transfer the corresponding amount of LP tokens to the specified recipient.

When the borrower borrows from the BAMM, the LP tokens are burned and the borrower receives underlying tokens. As a consequence, the liquidity providers won't receive pool swap fees for the borrowed LPs. The interest generated by the BAMM protocol should be enough for the LPs to be incentivized to provide liquidity to the protocol as opposed to just holding their LP tokens. In theory, a liquidity provider can receive fewer tokens during redemptions, than what he put into BAMM during mint.

2.2.1.2 Borrowers:

For each borrower, the contract keeps track of his position with a vault. The vault is a struct that contains the following fields:

- `token0`: The amount of token0 in the vault. This includes `token0` being rented from the protocol as well as potential collateral that the borrower has added to the vault.

- `token1`: The amount of token0 in the vault. This includes `token1` being rented from the protocol as well as potential collateral that the borrower has added to the vault.
- `rented`: The unscaled amount of `sqrt(#token0 * #token1)` that is being rented. This field scaled by `rentedMultiplier` represents the real amount being rented.

The solvency of a vault is determined by the following formula:

$$ltv := \frac{rented * rentedMultiplier}{\sqrt{token0 * token1}}$$

$$solvent := (rented = 0 \,||\, ltv < solvencyThreshold)$$

Each borrower can perform the following actions as long as the updated vault remains solvent:

- Add tokens to the vault.
- Remove tokens from the vault.
- Swap tokens from the vault.
- Borrow tokens from the protocol and have them added to the vault (increase `rented`).
- Repay tokens to the protocol and have them removed from the vault (decrease `rented`).
- Close their position, effectively repaying the user loan, and sending the remaining tokens to the user.

All of these actions can be performed by calling `executeActions()` (or `executeActionsAndSwap` when a swap should be executed). The function allows any combination of the actions described above in a single call, except for borrowing and repaying/closing the position in the same call. The actions are executed in the following order and the solvency of the vault is only checked at the very end of the function:

1. Borrow tokens.
2. Add tokens to the vault.
3. Execute the swap.
4. Repay the user's loan.
5. Send tokens from the vault to the user. (in case the position is being closed: all tokens).

Adding and removing tokens

A user can always add more tokens to their vault as this essentially increases the collateralization ratio of their vault. The user can also remove tokens from their vault as long as the vault remains solvent given the formula described above.

Borrowing tokens

Given some amount of `sqrt(x*y)` to be borrowed, the contract computes the amount of LP to burn and adds the corresponding amount of tokens to the vault. Given that the amount of tokens obtained follows the ratio of the underlying pool, should the user want to only borrow one of the two tokens and have their collateral in the other token, they will have to add the desired collateral and swap some tokens for the desired token. The debt of the user is denominated in terms of `sqrt(x*y)` and not in LP tokens to avoid malicious actors creating bad debt in the system by increasing the value of the LP token by donating to the underlying pool for example.

Repaying tokens

Given the amount of `sqrt(x*y)` to be repaid, including the interest accrued, the contract computes the amount of LP to mint and removes the corresponding amount of tokens from the vault. A position can be partially or fully repaid.

Swapping tokens

Given some swap parameters, the contract will swap the user vault's tokens through the Fraxswap Router Multihop. The caller can specify the amount of tokens to be swapped, the minimum amount of tokens to receive, and the path of the swap. The contract will execute the swap and update the vault accordingly.

2.2.1.3 Liquidators:

In the BAMM, insolvent positions can be liquidated by anyone. Liquidations can be triggered by calling `liquidate()` and consist of selling one of the Vault's tokens to the underlying pool to balance the Vault before minting some LP to repay part or all the position. It works as follows:

1. The spot price of the underlying pools is compared to its TWAP to ensure that the pool is being manipulated at the time of the liquidation, this is done using the `FraxswapOracle`.
2. In the case the amount of token to be sold is larger than one of the following, it is capped to the smallest amount of:
    - 1/10th of the vault's balance
    - 1/10th of the underlying pool's total liquidity

    In most cases, even swapping a small amount of the vault balance is sufficient to bring the vault to an overcollateralized state. Limiting the sold value also prevents high slippage swaps.
3. The amount of tokens to be sold is swapped in the underlying pool.
4. Several checks are performed:
    - In case of a partial liquidation the token that has been sold should still be in excess in the vault.
    - In case of a full liquidation, the vault's assets should be in the same ratio as the underlying pool given some error margin.
5. The liquidator is given a fee of 1% of the liquidated assets.
6. The liquidity is added back to the pool and the LP tokens are minted to repay the position:
    - If the liquidation covers the entire debt of the vault, the debt is nullified and the remaining tokens are left in the vault. This case also includes partial liquidations if they were able to cover the entire debt.
    - Otherwise, in case of a partial liquidation that could not cover the entire debt, the debt is reduced by the amount of tokens sold and the remaining tokens are left in the vault.
    - Finally if none of the cases above match, the liquidation is not able to cover the entire debt and the part of the debt that was not covered is left in the system as bad debt. The bad debt is socialized among the LPs of the system. Given that the BAMM's solvency is independent of the price of the two underlying tokens, Such a case should not frequently since, for it to happen, the following conditions should be met, and liquidators should not have liquidated the vault before:
        - Interests have accumulated to the point that the vault is not only insolvent (according to `_solvent()`) but also has a loan-to-value ratio greater than 1.
        - The liquidation trade should not bring the vault back to solvency. This can happen when the token proportions of the vault are already very close to the one of the underlying pool, meaning that the trade will not have a significant impact on the vault's solvency (`sqrt(x*y)` will not increase a lot after the trade).
7. The vault doesn't need to be solvent after a liquidation, it could be that several partial liquidations are needed to make the vault solvent again.

2.2.1.4 Utilization and interest rates:

The interest rate of the protocol is determined by the utilization rate. The utilization is defined as the ratio of the total amount of `sqrt(x*y)` borrowed to the total amount of `sqrt(x*y)` managed by the protocol. The logic in the `VariableInterestRate` contract is used to calculate the interest rate and is similar to the one being used in `FraxLend` which is explained in detail in the FraxLend documentation.

The model used here is a linear rate that allows for two linear functions to be defined. The slope of these two functions is updated over time with a Time-weighted Variable Interest rate. When the utilization is low, the slope will decrease. If utilization is high, the Vertex and Max Rate will increase. In short the less the utilization is, and the longer the time it has been low, the lower the new interest rate will be. On the other hand, the higher the utilization is, and the longer the time it has been, the higher the new interest rate will be. In the case of the BAMM protocol, when the utilization is between 72.5% and 82.5%, the target utilization range, the interest rate is constant over time given that the utilization does not change.

2.2.2 BAMM Factory

The BAMM factory is a simple factory contract that allows the creation of new BAMM contracts. The implementation contract for `BAMM` can be set using `setCreationCode()` by the owner of the factory only once. Anyone can then create new BAMM contracts by calling `createBAMM()` with a given pair address. The pair address must be a valid Fraxswap pair address. The factory allows for deploying contracts whose init bytecode size is greater than 24KB by storing it in 2 separate contracts and concatenating their content before deploying the contract.

2.2.3 FraxswapOracle

The FraxswapOracle is a stateless contract that allows for querying the TWAP of a Fraxswap pair given some period. The contract queries the pair's observation history in a binary search fashion to find the TWAP of the period given.

2.2.4 Trust Model

The underlying tokens of the pool are expected to be regular ERC20 tokens that are not rebasing and don't have fees on transfers. The owner of `BAMMFactory` is assumed to be trusted as they can set the implementation contract for the BAMM contracts.

2.2.5 Changes in Version 2 and Version 3

Both versions do not introduce any significant changes to the system, the modifications are implementations of fixes to the issues found in the initial version of the contracts.

3 Limitations and use of report

Security assessments cannot uncover all existing vulnerabilities; even an assessment in which no vulnerabilities are found is not a guarantee of a secure system. However, code assessments enable the discovery of vulnerabilities that were overlooked during development and areas where additional security measures are necessary. In most cases, applications are either fully protected against a certain type of attack, or they are completely unprotected against it. Some of the issues may affect the entire application, while some lack protection only in certain areas. This is why we carry out a source code assessment aimed at determining all locations that need to be fixed. Within the customer-determined time frame, ChainSecurity has performed an assessment in order to discover as many vulnerabilities as possible.

The focus of our assessment was limited to the code parts defined in the engagement letter. We assessed whether the project follows the provided specifications. These assessments are based on the provided threat model and trust assumptions. We draw attention to the fact that due to inherent limitations in any software development process and software product, an inherent risk exists that even major failures or malfunctions can remain undetected. Further uncertainties exist in any software product or application used during the development, which itself cannot be free from any error or failures. These preconditions can have an impact on the system's code and/or functions and/or operation. We did not assess the underlying third-party infrastructure which adds further inherent risks as we rely on the correct execution of the included third-party technology stack itself. Report readers should also take into account that over the life cycle of any software, changes to the product itself or to the environment in which it is operated can have an impact leading to operational behaviors other than those initially determined in the business specification.

4 Terminology

For the purpose of this assessment, we adopt the following terminology. To classify the severity of our findings, we determine the likelihood and impact (according to the CVSS risk rating methodology).

- *Likelihood* represents the likelihood of a finding to be triggered or exploited in practice
- *Impact* specifies the technical and business-related consequences of a finding
- *Severity* is derived based on the likelihood and the impact

We categorize the findings into four distinct categories, depending on their severity. These severities are derived from the likelihood and the impact using the following table, following a standard risk assessment procedure.

Likelihood	Impact		
	High	Medium	Low
High	Critical	High	Medium
Medium	High	Medium	Low
Low	Medium	Low	Low

As seen in the table above, findings that have both a high likelihood and a high impact are classified as critical. Intuitively, such findings are likely to be triggered and cause significant disruption. Overall, the severity correlates with the associated risk. However, every finding's risk should always be closely checked, regardless of severity.

5 Findings

In this section, we describe any open findings. Findings that have been resolved have been moved to the Resolved Findings section. The findings are split into these different categories:

- Design: Architectural shortcomings and design inefficiencies
- Correctness: Mismatches between specification and implementation

Below we provide a numerical overview of the identified findings, split up by their severity.

Critical-Severity Findings	0
High-Severity Findings	0
Medium-Severity Findings	0
Low-Severity Findings	2

5.1 Denial of Service Against Redeeming and Executing Actions

Design Low Version 1 Risk Accepted

CS-FRAX_BAMM-003

The function `BAMM.redeem()` and `executeActionsAndSwap()` performs the following check before returning:

```
if (!_isValidUtilityRate({ reserve0: reserve0, reserve1: reserve1, pairTotalSupply: pairTotalSupply })) {
    revert InvalidUtilityRate();
}
```

A malicious actor could front-run call to those functions to make them revert by either:

- redeeming a large amount of LP tokens to increase the utilization of the contract and make the check fail. Note that this requires the attacker to hold a large amount of BAMM tokens in the first place.
- borrowing a large amount of tokens to max out the utilization and repaying just after the target transaction. This could be made feeless given that the interest would not accrue in the meantime if all transactions are made in the same block.

Risk accepted

Frax Finance answered:

> Not possible and expensive on mainnet because you can go around it using flashbots. On L2s like Fraxtal it might be possible, but the attacker needs to do it blindly for every block, because the attacker can not know when users are going to use the BAMM.

5.2 `BAMMFactory` Does Not Support Contracts Smaller Than 20500 Bytes

Design | Low | Version 1 | Risk Accepted

CS-FRAX_BAMM-011

The `BAMMFactory.setCreationCode()` function includes some logic in the case `_creationCode.length < 20_500`. However, in such case, `setCreationCode()` would always revert with a `slice_outOfBounds` error, from the call to `BytesLib.slice(_creationCode, 0, 20_500)`. Similarly, `createBamm()` cannot handle such a case given that `SSTORE2.read(contractAddress2)` would revert with an underflow since the contract would be empty in that case, while `SSTORE2.read()` expects at least 1 byte that is used as `DATA_OFFSET`.

Risk accepted

Frax Finance answered:

> This change is only required if `BAMM.sol` contract size is less than 20.5kb. This will not happen in the current spec.

6 Resolved Findings

Here, we list findings that have been resolved during the course of the engagement. Their categories are explained in the Findings section.

Below we provide a numerical overview of the identified findings, split up by their severity.

Critical-Severity Findings	0
High-Severity Findings	0
Medium-Severity Findings	2

- Incorrect Fee Computation Code Corrected
- BAMM Does Not Always Account for Potential Pair Fees Code Corrected

Low-Severity Findings	6

- Contract Does Not Implement Interface Code Corrected
- Floating Dependencies Versions Code Corrected
- Floating Pragma Code Corrected
- Rounding Direction When Minting Code Corrected
- BAMM._addTokensToVault() Permit When Closing the Vault Can Cause Reverts Code Corrected
- FraxOracle Does Not Handle Potential Overflows in Pairs Code Corrected

Informational Findings	7

- Initializable Used for Non-Upgradeable Contract Code Corrected
- Misleading Comments Code Corrected
- Project Uses a Deprecated Library Code Corrected
- Contract Still Has TODOs Code Corrected
- Incorrect NatSpecs Code Corrected
- Gas Savings Code Corrected
- calcRent() Missing Reentrancy Modifier Code Corrected

6.1 Incorrect Fee Computation

Correctness Medium Version 1 Code Corrected

CS-FRAX_BAMM-001

In `BAMM._addInterest()`, the fee to be sent to the factory's fee receiver if specified is computed as follows:

```
address feeTo = IBAMMFactory(factory).feeTo();
if (feeTo != address(0)) {
    // accrue fee
    uint256 fee = (sqrtInterest * FEE_SHARE) / 10_000;
    uint256 feeMintAmount = (fee * iBammErc20.totalSupply()) /
        (sqrtBalance + ((uint256(sqrtRented) * rentedMultiplier) / PRECISION));
```

```
    iBammErc20.mint(feeTo, feeMintAmount);
}
```

However, the computation of `feeMintAmount` does not account for the `rentedMultiplier` in the numerator. This means that as the interest accrues and the `rentedMultiplier` increases, the fee sent to the factory's fee receiver will decrease given some fixed rented amount of `sqrt(x,y)`.

Code corrected

The computation of `feeMintAmount` has been corrected to account for the `rentedMultiplier` in the numerator.

6.2 BAMM Does Not Always Account for Potential Pair Fees

Design | Medium | Version 1 | Code Corrected

CS-FRAX_BAMM-002

BAMM contract intended to work with `FraxswapPair` contracts, that might have fee enabled, where 1/6th of the growth in sqrt(k) is minted as liquidity. This fee is accounted for in `BAMM._calcMintedSupplyFromPairMintFee()` function which is used during borrowing and repayment of the debt. However, in functions, this fee is not accounted for:

- In `mint()`, `bammOut` will be overestimated.
- in `redeem()`, `lpOut` will be underestimated.
- in `_addInterest()`, since `sqrtBalance` will be overestimated both the utility rate and the interest rate will be underestimated.
- in `_currentUtilityRate()`, since `sqrtBalance` will be overestimated, the utility rate will be underestimated.
- In `liquidate()`, the computation of `lpTokenToLiquidity` and `sqrtToLiquidity` also do not account for the fee.

This means for example that it is more beneficial for a BAMM liquidity provider to `mint()` BAMM tokens when the fee hasn't been minted for a long time, and to `redeem()` when the fee has been minted recently.

Code corrected

The pair's fees are now accrued in `Bamm._addInterest()` which is called at the beginning of each external function and after each action that could have changed the reserves or the supply of the pair.

6.3 Contract Does Not Implement Interface

Design | Low | Version 1 | Code Corrected

CS-FRAX_BAMM-013

The `BAMM` contract uses `IFraxswapOracle` to call the `FraxswapOracle` contract. However, the `FraxswapOracle` contract does not implement `IFraxswapOracle`. Such separation of interface and

implementation is not a good practice as changing in implementation may break the contract that uses the old interface and the compiler would not be able to catch such errors.

Code corrected

The `FraxswapOracle` contract now implements the `IFraxswapOracle` interface.

Additionally, changes were made such that `BAMM`, `BAMMFactory` and `BAMMERC20` now implement their respective interfaces. Similarly `VariableInterestRate` implements `IVariableInterestRate` instead of `IRateCalculatorV2` as done previously.

6.4 Floating Dependencies Versions

Design | Low | Version 1 | Code Corrected

CS-FRAX_BAMM-004

The versions of the contract libraries in `package.json` are not fixed. Please consider the following example:

```
"@openzeppelin/contracts": "^5.0.2",
```

The caret `^version` will accept all future minor and patch versions while fixing the major version. With new versions being pushed to the dependency registry, the compiled smart contracts can change. This may lead to incompatibilities with older compiled contracts. If the imported and parent contracts change the storage slot order or change the parameter order, the child contracts might have different storage slots or different interfaces due to inheritance.

In addition, this can lead to issues when trying to recreate the exact bytecode.

Code corrected

`package.json` has been updated to fix the versions of the contract libraries.

6.5 Floating Pragma

Design | Low | Version 1 | Code Corrected

CS-FRAX_BAMM-005

Frax Finance uses floating pragmas in the various contracts and `foundry.toml` does not fix a Solidity version. Contracts should be deployed with the same compiler version and flags that have been used during testing and audit. Locking the pragma helps to ensure that contracts do not accidentally get deployed using, for example, an outdated compiler version that might introduce bugs that affect the contract system negatively (https://github.com/SmartContractSecurity/SWC-registry/blob/b408709/entries/SWC-103.md).

Code corrected

The floating pragma versions have been replaced by fixed pragmas in the contracts, and the Solidity version has been set to `0.8.23` in `foundry.toml` file.

6.6 Rounding Direction When Minting

Design | Low | Version 1 | Code Corrected

CS-FRAX_BAMM-008

Given the following code snippet from the `BAMM.mint()` function:

```
uint256 sqrtBalance = (balance * sqrtReserve) / pairTotalSupply;
uint256 sqrtRentedReal = (uint256(sqrtRented) * rentedMultiplier) / PRECISION;
bammOut = (sqrtAmount * totalSupply_) / (sqrtBalance + sqrtRentedReal);
```

The `sqrtRentedReal` and `sqrtBalance` are calculated using divisions that truncate the result. Thus, the `bammOut` is effectively rounded up, since the `sqrtBalance + sqrtRentedReal` is rounded down. For the sake of `BAMM` solvency, the value that leaves the pool should be rounded down and the value that enters the pool should be rounded up.

Code corrected

The two variables are now computed using `mulDiv()` from `OpenZeppelin`, which performs the computation with the `Expand` rounding direction (rounds away from zero).

6.7 `BAMM._addTokensToVault()` Permit When Closing the Vault Can Cause Reverts

Design | Low | Version 1 | Code Corrected

CS-FRAX_BAMM-009

Consider the scenario where a user plans to add a token to their vault, repay the debt and close the vault.

The `executeActions()` function is called with the following steps:

1. `BAMM._addTokensToVault()` is called to add tokens to the vault
2. `BAMM._repay()` is called to repay the debt
3. `BAMM._addTokensToVault()` is called again if the user has added not enough tokens during step 1.

If the tokens in step 1. were added using the `permit()` of the token, the same permit will be called again in step 3. Since the nonce of the permits can be consumed only once, the second call will revert.

Code corrected

The function `BAMM._addTokensToVault()` has been modified and now sets `_action.v` to `0` after the permit is called. This forces the second call to the function in the example above to use `BAMM._moveTokenForVault()`

6.8 `FraxOracle` Does Not Handle Potential Overflows in Pairs

Design | Low | Version 1 | Code Corrected

CS-FRAX_BAMM-012

The `FraxswapPair` contracts are built with certain overflows considered as possible:

1. The timestamp is tracked as `uint32` and can overflow after 82 years.
2. `price0CumulativeLast` and `price1CumulativeLast` are tracked as `uint256` and can potentially overflow, depending on reserve ratios and decimals difference between the tokens.

The function `FraxOracle.getPrice()` does not support such overflows and for example will revert in this case, if any of the above-mentioned overflows happen:

```
(lastObservation.price0CumulativeLast - foundObservation.price0CumulativeLast) /
    (lastObservation.timestamp - foundObservation.timestamp)
```

While this issue is possible in theory, in practice it can happen only after a very long time. The problem will persist only when the price lookup window overlaps the block, where the overflow happened.

Code corrected

Unchecked blocks are now used to allow the computations to overflow.

6.9 Contract Still Has TODOs

Informational **Version 1** **Code Corrected**

CS-FRAX_BAMM-014

The contracts `BAMM` and `FraxswapOracle` still have TODOs in the code. This is not a security issue, but it is a good practice to address all TODOs from the code before deployment.

Code corrected

All TODOs have been removed from the codebase.

6.10 Gas Savings

Informational **Version 1** **Code Corrected**

CS-FRAX_BAMM-016

The following gas savings were found in the `BAMM` contract.

1. In `_syncVault()`, `rentedMultiplier` is read twice from storage, a SLOAD could be saved.
2. In `liquidate()`, `rentedMultiplier` is read twice from storage, in the case where `liquidationMath.partialLiquidationOut > 0`, a SLOAD could be saved.
3. `variableInterestRate` is a storage variable, but is only set in the constructor, it could be an immutable variable.
4. In `calcRent()`, `rentedMultiplier` is read twice from storage, a SLOAD could be saved.
5. In `_addInterest()`, `sqrtRented` is read 6 times from storage, 5 SLOAD could be saved. Additionally, `rentedMultiplier` is read twice from storage before being updated and then once again, 2 SLOAD could be saved.
6. In `_currentUtilityRate()`, `sqrtRented` is read twice from storage, a SLOAD could be saved.

In other parts of the project following gas savings were found:

1. The function `BAMMFactory.createBamm()` is defined as `public` but is never called internally, it could be made `external`.

Code corrected

All gas savings were applied to the code.

6.11 Incorrect NatSpecs

Informational **Version 1** **Code Corrected**

CS-FRAX_BAMM-017

The NatSpec of the function `BAMM._calcMintedSupplyFromPairMintFee()` is incorrect:

```
@notice public view function to view the calculated interest rate at a given utilization
```

Additionally, the following functions are missing some NatSpec fields:

- `_calcMintedSupplyFromPairMintFee()` is missing the `totalSupply` parameter.
- `_solvent()` is missing the `solvencyThreshold` parameter.
- `executeActions()`, `executeActionsAndSwap()` and `_currentUtilityRate()` are missing the NatSpec return field.

Code corrected

All mentioned NatSpecs have been corrected.

6.12 Misleading Comments

Informational **Version 1** **Code Corrected**

CS-FRAX_BAMM-018

The following comments appear to be misleading:

- In `BAMM._repay()`, a comment is duplicated and does not match the code in the second instance.

```
// token0Amount and token1Amount are negative as they subtracted balances from the vault, so we need to convert positive
token0ToAddToLp = uint256(-token0Amount);
token1ToAddToLp = uint256(-token1Amount);

// token0Amount and token1Amount are negative as they subtract balances from the vault
if ((token0ToAddToLp < _token0AmountMin) || (token1ToAddToLp < _token1AmountMin)) {
    revert InsufficientAmount();
}
```

- In `BAMM.liquidate()`, the following comment is misleading as in the case of a partial liquidation, the liquidator's fee is a percentage of the liquidated amount, not the total amount.

```
// Give the liquidation fee to the liquidator as a percentage of the total vaults value
```

- In `BAMM._borrow()`, the following comment is incorrect as the tokens are being sent to the borrower if action.token0Amount and/or action.token1Amount is negative and not positive.

```
// The tokens will be given to the borrower later, assuming action.token0Amount and/or action.token1Amount is positive
```

Code corrected

The comments have been updated to match the code.

6.13 Project Uses a Deprecated Library

Informational | Version 1 | Code Corrected

CS-FRAX_BAMM-019

The package `@rari-capital/solmate` is deprecated and no longer maintained. The successor is @transmissions11/solmate. However, the only used library from this package is `SSTORE2`, which is identical to the one in the successor package.

Code corrected

`@raricapital/solmate` has been replaced by `@transmissions11/solmate`.

6.14 `Initializable` Used for Non-Upgradeable Contract

Informational | Version 1 | Code Corrected

CS-FRAX_BAMM-021

`BAMMFactory` contract extends `Initializable` and is used for creating new BAMM contracts. The code of the `BAMM` contract is set using `setCreationCode()` which has the `initializer` modifier. While `Initializable` technically fulfills the goal in this contract and prevents calling `setCreationCode()` multiple times, it is not necessary to use it in this case. Given that the variable `contractAddress1` is always 0x0 before the first call to `setCreationCode()` and can never be set to 0x0 again, the use of `Initializable` is not necessary in this contract.

Code corrected

`Initializable` was removed and the following check was added at the beginning of `setCreationCode()`:

```
require(contractAddress1 == address(0));
```

6.15 `calcRent()` Missing Reentrancy Modifier

Informational | Version 1 | Code Corrected

CS-FRAX_BAMM-022

The function `BAMM.addInterest()` calls `_addInterest()` and is protected against reentrancy, however, the function `BAMM.calcRent()` which also calls `_addInterest()` is not. No security implications were found from this behavior.

Code corrected

The function `calcRent()` was moved to an external `BAMMUIHelper` contract that calls the function `BAMM.addInterest()`, triggering the reentrancy check.

7 Informational

We utilize this section to point out informational findings that are less severe than issues. These informational issues allow us to point out more theoretical findings. Their explanation hopefully improves the overall understanding of the project's security. Furthermore, we point out findings which are unrelated to security.

7.1 Denial of Service Against Liquidations

Informational | Version 1 | Acknowledged

CS-FRAX_BAMM-015

The function `BAMM.liquidate()` ensures that the underlying pool is not being manipulated using `BAMM.ammPriceCheck()`. This check can fail in two cases:

- the TWAP is too far from the spot price
- The `FraxswapOracle.getPrice()` function reverts due to price outliers in the lookup period

If any of those cases happen, the `liquidate()` function will revert. This means that a fast price change or a malicious actor could prevent liquidations from happening. As a position can only be insolvent if interest accrued too much and liquidations are independent of the underlying asset prices, this behavior is not a critical issue and only results in a delay of liquidations.

Acknowledged

Frax Finance answered:

> There is enough time to liquidate and TWAP manipulations are expensive and risky.

7.2 Migration Issues in Libraries Due to Solidity Version Upgrade

Informational | Version 1 | Acknowledged

CS-FRAX_BAMM-007

Some libraries were initially implemented in Solidity 0.6 and later the compiler version was updated to 0.8. Due to built-in SafeMath checks in solc 0.8, some libraries are not working as expected.

- The function `UQ112x112.uqdiv(uint224 x, uint112 y)` will revert if `y` is 0.
- `UQ112x112.encode(uint112 y)` performs a checked multiplication but the multiplication would never overflow.
- `FixedPoint.mul()` performs an overflow check, which is not required. It can never be triggered as it happens after the default SafeMath checks. In addition, if `y` is 0, it will revert, due to SafeMath division by zero check.
- `FixedPoint.muluq()` performs multiplication that would never overflow, with default SafeMath checks.
- `FixedPoint.divuq()`, `reciprocal()` and `fraction()` functions perform checked divisions by a number which is required to be non-zero.
- `BitMath.mostSignificantBit()` and `BitMath.leastSignificantBit()` perform several checked additions and subtractions that cannot overflow.

Acknowledged

Frax Finance answered:

> `UQ112x112.uqdiv(uint224 x, uint112 y)` will never have `y = 0` as the reserves of a pair can never be 0. The other comments only imply reduced safety requirements as safe math is built-in. No changes needed.

7.3 Utilization Rate Can Be Greater Than `0.9`

Informational **Version 1** **Acknowledged**

CS-FRAX_BAMM-020

The `BAMM` enforces a maximum utilization rate of 0.9. This is done by ensuring that the utilization rate is not above this value after calling `redeem()`, `executeActions` or `executeActionsAndSwap`. However, due to interests accruing, it might be possible that the utilization rate becomes greater than this value, for example by calling `mint()` after some time passed with the utilization rate being 0.9. In this case, the function `_getVariableInterestRate()` will call `VariableInterestRate`'s `getNewRate()` with a `utilScaled` greater than 1. The interest rate might hence be greater than the `MAX_FULL_UTIL_RATE` defined in the `VariableInterestRate`.

Acknowledged

Frax Finance answered:

> Not a problem, utilization cap is there to limit slippage during liquidations. Utilization will not remain above the cap for long, because the interest rate will spike and borrowers will repay the rented LP.

7.4 `BAMM._addTokensToVault()` Two Tokens and Permit Fail

Informational **Version 1** **Acknowledged**

CS-FRAX_BAMM-010

The function `BAMM._addTokensToVault()` can be used to transfer both `token1` and `token0` in a single call. For that, it relies on the `transferFrom` function of the token contract and approval from the user. The approval on a token can be set by providing a signature to `BAMM._addTokensToVault()` to be verified by the `permit` function. However, if both tokens are transferred in a single call, the permit will default to the `token0`. Thus, setting permit on a `token1`, when both tokens need to be transferred, will not work.

Acknowledged

Frax Finance answered:

> Normal use is to only transfer one token per action. A user could use two transactions or do a separate approval when it wants to transfer in two tokens.

8 Notes

We leverage this section to highlight further findings that are not necessarily issues. The mentioned topics serve to clarify or support the report, but do not require an immediate modification inside the project. Instead, they should raise awareness in order to improve the overall understanding.

8.1 Oracle Manipulation on FIFO L2s

Note Version 1

The liquidations rely on an oracle to ensure that the underlying pool's price is accurate. If the oracle price is not accurate, `BAMM` users could be liquidated at a lower price and lose assets.

On Ethereum mainnet, the cost of manipulating such a TWAP oracle is high as this requires manipulating the pool at the end of multiple blocks and controlling each time the next blocks to ensure that the manipulation can be safely reverted without being arbitraged. The following shows that on L2s using a FIFO ordering, the cost of manipulating the oracle is significantly lower.

Price oracles must be robust and manipulation-resistant. It must be expensive to manipulate the markets which are used as price sources.

Usually, the factor that makes price manipulation expensive is arbitrage. If a manipulator pushes the price of an asset too high or too low, arbitrageurs will see this and make a profit by moving the price back to the "true price". Any profit made by arbitrageurs will be a loss to the manipulator.

This "arbitrage assumption" breaks down in two cases:

1. All markets for the token are manipulated simultaneously, so it is difficult to determine the "true price". There is no other market to arbitrage against.
2. Arbitrageurs are not able to see the manipulated price quickly enough, so they cannot take advantage of it.

Attacks that target condition 2. are known as "Multi-block MEV" attacks. The idea is that a manipulator could control the order of transactions in a block, which allows the following:

- In block `n`, the manipulator sends a transaction (through a private mempool like Flashbots) that manipulates the price of an asset.
- In block `n+1`, the manipulator ensures that the first transaction in the block is one where they revert the price back to the original value.

As a result, arbitrageurs will have no chance of reacting to the manipulation, as it will already be over by the time they can get a transaction included in the block. However, if there is an Oracle that reads the price at the beginning of block `n+1`, it will see the manipulated price.

This attack is well-known on Ethereum, but is generally deemed expensive to execute, as it requires being or having an agreement with the ETH staker that is chosen to propose block `n+1`. If the attack should be repeated multiple times, it requires being chosen as block proposer multiple times within a short time frame, which requires a significant amount of ETH staked.

However, on L2s block production works differently. Instead of a different proposer being chosen in each block, there is typically a single sequencer that decides on a block ordering policy. One commonly used policy used by chains such as Arbitrum, Optimism and Base is "FIFO" (First In, First Out), where transactions are included in the order they were received.

In FIFO ordering, the order of transactions is determined by time, not by the price a user is willing to pay. This can be taken advantage of to fulfill condition 2. above, without needing to be a block producer.

The FIFO attack looks as follows:

1. The manipulator experiments to figure out their latency to the sequencer (and ideally minimizes it).
2. The manipulator sends a manipulation transaction at a time such that it will arrive at the sequencer towards the end of the period in which it is building block `n`.
3. The manipulator sends a second transaction just before the end of block `n`, so that it reaches the sequencer at the beginning of the period in which it is building block `n+1`.

Arbitrageurs are only able to see the manipulated price once block `n` is published by the sequencer. By that time, the manipulator has already sent the second transaction that reverts the price back to the original value. As time is the only relevant factor, it is impossible for a transaction that is created later to be included in the block first (unless the arbitrageur has significantly lower latency to the sequencer).

The only cost to the attacker is the trading fees paid. As the attack cannot use a flashloan, they must also have sufficient capital available to manipulate the price by the percentage they aim for. The attack can be repeated as many times as the attacker wants, although repeated attacks could be speculatively frontrun by arbitrageurs if they detect a pattern. Repeated attacks can be used to circumvent outlier-detection mechanisms and TWAPs.

A policy that modifies transaction ordering to be based on a payment in addition to timing would make the attack significantly more expensive. For example, "Arbitrum time-boost" has been proposed, but not yet implemented. See Time Boost Medium post.

Note that in the case of `BAMM`, such an attack should hence be repeated multiple times to have a significant impact on the TWAP and circumvent checks performed by the `FraxswapOracle`, allowing the attacker to liquidate users at a lower price than the true price.

In summary, Multi-block MEV attacks are likely much more realistic to execute on FIFO L2s than on Ethereum, as they are possible without needing to be a block producer. They can affect on-chain TWAPs as well as any off-chain oracles that use L2 on-chain markets as a primary price source. This must be considered when deciding which assets have an oracle that is robust enough to allow lending.