1inch: Fixed Rate Swap Audit

OpenZeppelin | security

Introduction

The 1inch team asked us to review and audit their Fixed Rate Swap smart contracts. We looked at the code and now publish our results.

Scope

We audited commit `b1600f61b77b6051388e6fb2cb0be776c5bcf2d1` of the `1inch/fixed-rate-swap` repository. In scope was the following contract:

```
- FixedRateSwap.sol
```

All other project files and directories, including tests, external dependencies and projects, game theory, and incentive design, were excluded from the scope of this audit. External code and contract dependencies were assumed to work as documented.

Overall Health

We found the codebase to be complex, often lacking sufficient documentation to explain the non-trivial math and logic used throughout. For a codebase of this size, we consider the sheer number of issues included in this report to be indicative of unnecessary complexity. Together with a dearth of documentation, the codebase is not at the level of refinement we would expect for a production-ready system. Eliminating the, albeit clever, continuous fee function and replacing it with three separate linear functions could make much of the codebase far easier to reason about. A simpler design could eliminate many of the most complex code blocks where issues were identified. Refactoring and simplifying the protocol is highly advisable, in which case we recommend it undergo additional audits.

All said, the 1inch team was always ready to address any questions during the audit and they were very easy to work with. They are receptive to feedback and suggestions for continuous improvement.

Gas Consumption Considerations

There are some particularly gas-intensive parts of the codebase, such as the `_powerHelper` function and binary searches, which we observed using over 400k gas in some edge cases. Given the nature of the protocol and the stable coins it intends to deal with, the high gas consumption that the complexity of the project leads to could undermine the protocol's purpose and could certainly impact its general usability.

System Overview

The Fixed Rate Swap protocol is an Automatic Market Maker (AMM) that uses a constant sum price curve to facilitate swaps between assets whose prices are stable to each other. It also implements a variable fee mechanism that *generally* charges 0.01% as fees. When the AMM's token balances go to extreme conditions, fees are either reduced to 0% or raised up to 0.2% depending on if the activity is desirable or not, respectively, for the asset composition of the pool. Such fees are kept in the pool, where liquidity providers will benefit from them by holding the shares (the AMM's own `ERC20` LP token).

Privileged Roles

There are no privileged roles in the audited version of the protocol.

External dependencies and trust assumptions

The protocol does not intend to support `ERC20` tokens that charge fees on transfer.

Also, it is worth mentioning that there might be assets that, depending on their nature, could produce unexpected behaviors similar to those described in this report, e.g. stable coins with an incongruous number of decimals between each other or assets that could allow unbounded flash loans.

Findings

Here we present our findings.

Update: *The 1inch team applied several fixes based on our recommendations and provided us with a set of commits that target the respective issues found. These commits are stated on each respective issue and we address the fixes introduced under each individual commit. However, we only reviewed* *specific patches to the issues we reported. The codebase underwent some other changes we have not audited and we recommend that those changes are reviewed in depth in a future audit.*

Critical severity

None.

High severity

None.

Medium severity

[M01] Lack of output safeguards could lead to loss of funds

The `FixedRateSwap` contract facilitates the swapping of one stable coin for another, and also allows users to deposit and withdraw assets for "shares" of the liquidity pool (LP tokens).

However, neither swaps, withdrawals, nor deposits provide a mechanism for users to set minimum acceptable return values from interactions with the pool.

The pool assesses variable fees based on how interactions impact the pool's composition. Nevertheless, with this design, since assessed fees would go directly to pool share holders, there is an economic incentive for the majority share holder to front-run large token swaps, manipulating the pool composition in order to force higher fees.

Similarly, front-running could also be used to take advantage of protocol rounding within token output calculations. Although unlikely given the current state of the ecosystem in regards to popular stable coins, if the `fromBalance` of a token within the pool could be manipulated to be on the order of `1e36 * inputAmount` for any given swap, then this line of the `_getReturn` function would be truncated to `zero`, leading to a zero-value `outputAmount`.

However, because zero-value swaps are protected against, truncation within the `outputAmount` calculation would occur before the worst-case inputs. Specifically, a `fromBalance` anywhere within the range of [`1e26` to `1e35`] `* the input amount` could lead to truncation and a reduction of the `outputAmount` that could be profitable for pool share holders at the expense of users.

Although such a scenario may currently be unlikely, future ecosystem changes and unbounded flash loans could make this a more viable attack vector.

To avoid the unintentional loss of funds and mitigate possible front-running attacks, consider allowing users to specify the minimum acceptable return value for any interaction with the protocol that has monetary implications.

Update: *Fixed in commit `ea75a86`. However, no specific test has been added to validate that the fix implementation prevents this attack.*

Low severity

[L01] Deposit can revert on underflow

If a pool is dramatically unbalanced, deposits will revert during the calculation of the `shift` variable in the `_getVirtualAmountsForDepositImpl` function unless the total amount deposited is greater than 1/1000th of the existing balance in the pool.

For example, attempting to deposit a ratio of tokens totaling fewer than 10, in a pool with existing balances of 10,000 `token0` and .0001 `token1`, will revert. Depositing a greater amount will succeed.

This could be used to create a barrier to keep other users from joining the pool. In such a scenario, it is still possible to withdraw from and swap tokens via the pool. It is also possible to deposit at exactly the same ratio of the pool. If the pool shifts to be less one sided, then smaller deposits become possible again.

To avoid the potential for denial-of-service attacks, consider revising the codebase to better accommodate these edge cases. At the very least, consider explicitly checking for these conditions and returning helpful error messages when reverting.

Update: *Fixed in commit `4aa5210`.*

[L02] Incomplete event emissions

The `Swap` event is emitted every time the protocol facilitates a token swap.

However, there are several public methods available to execute token swaps. The `swap0To1` and `swap1To0` functions send the result of the swap directly to the `msg.sender`, but the `swap0To1For` and `swap1To0For` functions send the results of the swap to an address specified explicitly by the `to` parameter.

Since the same event it emitted in either case, off-chain parties have no way to distinguish which kind of swap was executed, or to whom the result of the swap was delivered.

Similarly, the `Deposit` and `Withdrawal` events only accept a single `user` address parameter, even though they are also emitted in functions where the `msg.sender` may be different than the party receiving tokens.

Consider adding an indexed `recipient` address parameter to the events so that they can more fully convey what actions are being taken by the protocol.

Update: *Acknowledged, and will not fix.*

[L03] Lack of input validation

The `public` `getReturn` function is lacking some input validation. Specifically:

- It does not validate that the `tokenFrom` and `tokenTo` addresses are the tokens that comprise the pool.
- It does not validate that the `inputAmount` parameter is non-zero.
- It does not validate that the `fromBalance + toBalance` value is non-zero.

Additionally, neither the `withdrawFor` nor the `withdrawForWithRatio` functions calculate if a user has enough shares to withdraw their requested amount. In either case, a withdrawal will revert if the condition is not satisfied, but the error messages are unclear and emitted only after consuming unnecessary gas.

Lastly, the `withdrawForWithRatio` function neglects to ensure that the values used for virtual balance calculations are less than or equal to the actual balances of the contract. Such a scenario would lead to a cryptic arithmetic revert when calculating `balanceX` and `balanceY` for the `_getRealAmountsForWithdrawImpl` function.

A lack of validation on user-controlled parameters may result in erroneous or failing transactions that are difficult to debug. To avoid this, consider improving the clarity of error messages and adding input validation to address the concerns raised above.

Update: *Partially fixed in commit `63b6a95` and commit `7c0ade7`. It has only been validated if the `inputAmount` value is zero and it has been moved the order of operations to fail early when burning LP tokens to reduce the gas cost. 1inch team's statement for this issue:*

> tokenFrom and tokenTo validation is not mandatory as in swap method they are substituted from constants

[L04] Constants not declared or clearly named

In the `FixedRateSwap` contract, the literal values `998`, `1000`, and `1002`, which are used to step through fees in the `_getRealAmountsForWithdrawImpl` and `_getVirtualAmountsForDepositImpl` functions, have no accompanying explanation or inline documentation.

Moreover, a lack of inline documentation also affects the ambiguously named constants which are used in calculations throughout the codebase.

To improve the code's readability and facilitate refactoring, consider defining a constant for every "magic value" and ensuring that all constants have clear and self-explanatory names. For complex values, consider adding inline documentation explaining how they were calculated or why they were chosen.

Update: *Fixed in commit `7091076`.*

[L05] Misleading or incomplete inline documentation

Throughout the codebase, a few instances of misleading and/or incomplete inline documentation were identified and should be fixed.

The following are instances of incomplete inline documentation:

- All constants, events, and variables from the `FixedRateSwap` contract.
- The `decimals` overrider and the `constructor` function.

The following are instances of misleading inline documentation:

- The calculations of the private function `_powerHelper` do not explicitly state that the result of the multiplication is divided by `1e18` to prevent double scaling of the value. Furthermore, the exponent of the scaling factor also matches the power given by the function, which could produce additional confusion if left unnoted.

Clear inline documentation is fundamental for outlining the intentions of the code. Mismatches between the inline documentation and the implementation can lead to serious misconceptions about how the system is expected to behave. Consider fixing any errors and adding additional documentation where identified to avoid confusion for developers, users, and auditors alike.

Update: *Acknowledged, and will not fix.*

[L06] No accessible coverage report

Although the `README` file points to a coverage report, it is inaccessible.

Additionally, there are no instructions for running the test coverage scripts.

Consider making the coverage report accessible and explicitly documenting how to run the test coverage scripts.

Update: *Partially fixed. The coverage report is now accessible, however the `README` file still does not explain how to run the scripts.*

[L07] Potentially unsafe unchecked math

Throughout the `FixedRateSwap` contract, there are many uses of `unchecked` math. The main reason for using `unchecked` math is to remove overflow/underflow checks in cases that either rely on such behavior or are known not to underflow/overflow. This has the benefit of saving gas, but, if used incorrectly, can lead to unexpected results and potential vulnerabilities.

Instances of `unchecked` arithmetic that can potentially underflow/overflow were identified. For example:

- The `_getReturn` function can overflow in multiple places, potentially returning a highly diminished `outputAmount`.
- The `_checkVirtualAmountsFormula` function can overflow.
- The `_powerHelper` function can overflow, but it is only used by the `private` `_getReturn` function.

The necessary values required to overflow these calculations, paired with reasonable validations throughout the codebase, often prevent these overflows from being achievable in practice, but they are still *theoretically* possible. Consider not using `unchecked` math except whenever the possibility of overflows can be *completely* ruled out, in order to prevent unexpected results and reduce the overall attack surface of the protocol.

Update: *Acknowledged, and will not fix.*

[L08] Unsafe explicit casting of integers

The `Swap` event takes an `address` and two `int256` parameters and it is emitted in the `swap0To1`, `swap1To0`, `swap0To1For`, and `swap1To0For` functions.

During emission, however, the integer values being passed to the event are explicitly cast from `uint256` to `int256` values.

Although unlikely to be problematic in practice today, ecosystems developments such as unbounded flash loans of stablecoin assets could cause this design to exhibit undesirable behaviors in the future. On a given large enough `uint256` value, explicitly casting into an `int256` type would truncate its value. As a result, off-chain systems, dependent on the accuracy of the event emission, could be misled.

Consider redefining the `Swap` event to deal directly with `uint256` values so that the functions that emit the event can forego the explicit casts.

Update: *Fixed in commit `8436c6c`. 1inch team imported the OpenZeppelin's `SafeCast` library to safely cast the mentioned cases.*

[L09] Withdrawal process could result in flooring

When a share holder uses either the `withdraw` or the `withdrawFor` functions, the contract calculates the amount of assets that they are entitled to given an amount of shares. Those assets are then transferred to the specified recipient.

However, since `if` statements are being used rather than `require` statements to validate whether any asset should be sent to the recipient, if the pool is unbalanced and the amount of shares is small, the contract could floor the value to be sent for either one or both of the assets.

Given that the values involved would necessarily be quite small, and in consideration of the fact that the protocol cannot completely restrict withdrawals where one token output may in fact be zero, consider documenting this potential rounding behavior so that users are aware of it when withdrawing.

Update: *Acknowledged, and will not fix.*

Notes & Additional Information

[N01] Deprecated project dependencies

During the installation of the project's dependencies, NPM warns that one of the packages installed, `Highlight`, "will no longer be supported or receive security updates in the future".

Even though it is unlikely that this package could cause a security risk, consider upgrading the dependency that uses this package to a maintained version.

Update: *Fixed in commit `0a2b55d`. However, the installation now requires the usage of the `-force` flag on current LTS versions of node in order to succeed.*

[N02] Pool fees may incentivize imbalanced deposits

When depositing into the `FixedRateSwap` contract, the `_getVirtualAmountsForDeposit` function calculates the virtual value of the deposit. Virtual amounts are the original amounts scaled, after charging the fee according to the pool's current asset ratio.

If a user deposits funds at the current ratio of assets within the pool then they are not charged fees. Otherwise, they are charged fees based on the difference between their deposit ratio and the pool's current asset ratio.

This design implies that when the ratio of assets within the pool is imbalanced, users would be incentivized to deposit at the same imbalanced ratio, rather than deposit at a ratio which would make the pool more balanced.

To encourage a more balanced pool, consider incentivizing deposits that balance the pool rather than penalizing them. Alternatively, if this is not feasible, consider explaining why within the project's documentation.

Update: *Acknowledged, and will not fix.*

[N03] Undocumented implicit approval requirements

The `FixedRateSwap` contract implicitly assumes that it has been granted an appropriate allowance before executing swaps and deposits which necessarily transfer tokens.

In favor of explicitness and to improve the overall clarity of the codebase, consider documenting all approval requirements in the relevant functions' inline documentation.

Update: *Acknowledged, and will not fix.*

[N04] Confusing implicit validation of `outputAmount`

The `getReturn` function is provided three parameters, namely, a token to swap from (`tokenFrom`), a token to swap to (`tokenTo`), and an input amount (`inputAmount` value of the `tokenFrom` asset). The corresponding `outputAmount` value (resultant amount for the `tokenTo` asset) is then calculated and returned.

Before the calculation, the function requires that the `inputAmount` value is less than or equal to the pool's token balance of the `tokenTo` asset. However, the `outputAmount` value, arguably the more intuitive value to check against the `tokenTo` asset balance, is never explicitly checked for the same condition.

In fact, the math used to calculate `outputAmount` value ensures that it will be strictly less than or equal to the `inputAmount` value.

However, the intentionality of this behavior is unclear, i.e. it is not obvious whether this design is just meant to fail more quickly during execution to reduce the gas cost or not. Consider explicitly documenting the reasoning for the exact check used. Additionally, consider validating if the balance of the `to` asset is greater than the `outputAmount` value as opposed to the `inputAmount` value.

Update: *Fixed in commit `10f4d9c`.*

[N05] Naming inconsistency

The `FixedRateSwap` contract sets an explicit pair of tokens that the swap, withdrawal, and deposit operations are meant to operate with. Throughout the codebase, these tokens are labeled with an index of either `0` or `1`, as in `token0` and `token1`. Functions that are descriptively named to convey details about usage also make use of those token indices, such as the `swap0To1` function.

However, for the `withdrawForWithRatio` function, the parameter that defines the proportion to receive of `token0` against `token1` is named `firstTokenShare` which could introduce confusion that it is in reference to the `token1` asset and not the `token0` asset.

To improve overall readability and reduce potential confusion, consider keeping naming conventions consistent throughout the entire codebase.

Update: *Fixed in commit `57ad4cd`.*

[N06] Revert messages are inconsistently formatted

The `require` statements in the constructor of the `FixedRateSwap` contract are formatted differently than all of the other `require` statements in the contract.

As inconsistently formatted revert messages can introduce unnecessary confusion, consider ensuring that all `require` statements have revert messages that are consistently formatted, accurate, informative, and user-friendly.

Update: *Fixed in commit `0aa4e9d`.*

[N07] Inconsistent use of named return variables

There is an inconsistent use of named return variables in the `FixedRateSwap` contract.

Specifically, while most functions return named variables, the `decimals`, `_getVirtualAmountsForDepositImpl`, `_getRealAmountsForWithdrawImpl`, and `_checkVirtualAmountsFormula` functions return explicit values.

Consider adopting a consistent approach to return values throughout the codebase by removing all named return variables, explicitly declaring them as local variables, and adding the necessary return statements where appropriate. This would improve both the explicitness and readability of the code, and it may also help reduce regressions during future code refactors.

Update: *Acknowledged, and will not fix.*

[N08] Gas optimizations

Within the `FixedRateSwap` contract, there are opportunities for a few simple gas consumption reductions. For instance:

- The `_getVirtualAmountsForDeposit` `private` function is only ever called from one place in the codebase, and that is the `depositFor` function. Within the `depositFor` function there are calls to `token0.balanceOf(address(this))` and `token1.balanceOf(address(this))`. However, those exact same calls are made at the top of the `_getVirtualAmountsForDeposit` function. The latter function could simply be passed the required values, instead of reading the balances twice per `depositFor` call.

- Within the `_getRealAmountsForWithdrawImpl` `private` function, the `secondTokenShare` variable is defined as `_ONE - firstTokenShare`. On the very next line, the exact same subtraction is performed again when it could simply use the `secondTokenShare` variable.

To reduce gas costs and further simplify the codebase, consider addressing the instances raised above.

Update: *Fixed in commit `34974ee`.*

[N09] Incorrect function visibility

The `withdrawWithRatio` function is not called internally by any of the functions in the `FixedRateSwap` contract. Consider setting the visibility to `external` instead of `public`.

Update: *Fixed in commit `cb852f5`.*

[N10] Reliance on matching `decimals` may be problematic

The protocol implicitly requires that both tokens inside of a pool support the `decimals` method for pool construction to be successful. In reality, this method is almost ubiquitous, but it is, technically, an optional component of the ERC20 specification wherein it is explicitly stated that contracts "MUST NOT expect these values to be present". Currently, any tokens that do not support the optional `decimals` method will not be usable within the protocol.

Perhaps more problematically, as part of these calls to token `decimals`, the protocol further requires that both tokens return identical values.

However, the stable coin token space is not homogeneous in this regard. It is comprised of many tokens that return a variety of different values for `decimals`. For instance, while USDT and USDC report 6 decimals, DAI reports 18 decimals.

If these are intentional limitations of the protocol, consider touching on them explicitly and providing short justifications via the inline documentation. Also consider providing better error messages at construction time for tokens that do not support the `decimals` method. Alternatively, consider making the protocol more robust, so that it can handle tokens that do not support the `decimals` method or pairs reporting disparate `decimals` within the same pool.

Update: *Fixed in commit `b49d808`. Inline documentation was added.*

[N11] Typographical error

We identified the following typographical error in the codebase:

- The revert message on line 92 of `FixedRateSwap.sol` starts without a capital letter, making it inconsistent with the rest of the code.

To improve the overall consistency and readability of the codebase, consider correcting this and any other typographical errors throughout the codebase.

Update: *Fixed in commit `a92d16a`.*

[N12] Unnecessarily `virtual` function

The `FixedRateSwap` contract inherits from OpenZeppelin's `ERC20` contract but it overrides its `ERC20.decimals` function. This is required because the `FixedRateSwap`'s liquidity pool token, being dependent on the `decimals` of the assets that comprise the pool, is necessarily dynamic.

However, even though this overriding implementation of the function should be final, it is defined with the `virtual` keyword, signaling that it is not necessarily a final implementation and allowing for it to be overridden again.

To avoid confusion and clarify intent, consider removing the `virtual` keyword or documenting the reasons for keeping it.

Update: *Fixed in commit `8bce5ec`.*

Conclusions

No high severity issues were found. Some changes were proposed to follow best practices and reduce the potential attack surface.