# Saddle Contracts Audit

DECEMBER 11, 2020 | IN SECURITY AUDITS | BY OPENZEPPELIN SECURITY


The Saddle team asked us to review and audit their smart contracts. We looked at the code and now publish our results.

*Note: All links to the project's code in this audit report refer to a repository that is private at the time of writing. They are therefore only accessible to the Saddle team.*

We audited commit 0b76f7fb519e34b878aa1d58cffc8d8dc0572c12 of the saddle-finance/saddle-contract repository. In scope were the contracts directly inside the `/contracts` folder, however the contracts within `/contract/helper` and `/contracts/interfaces` were out of scope.

***Update:*** *The Saddle team has fixed the issues identified in this report through pull requests in the audited Github repository.*

## System Overview

Saddle have created a Solidity implementation of Curve Finance's StableSwap – which Curve wrote in Vyper. Designed to help combat the price instability that stablecoins face in DeFi, the whitepaper states that StableSwap is *"a fully-autonomous market-maker for stablecoins with very minimal price slippage, as well as an efficient "fiat savings account" for liquidity providers on the other side"*.

Since writing the whitepaper, Curve have altered the exact formulae they use to calculate prices on-chain. Saddle therefore based their exact algorithms on Curve's `SwapTemplateBase.vy`. The scope of this audit was to ensure that Saddle's `Swap.sol` and `SwapUtils.sol` together function the same as commit `010a12b369a8d15b3ff917b51f4b93ec1c5cff47` of Curve's `SwapTemplateBase.vy`, with a few notable exceptions:

- Saddle have implemented an additional withdrawal fee for liquidity providers, that decreases over a 4 week period to 0

from when they deposit.

- Saddle have removed the 72 hour delay between proposing and applying admin changes that is implemented in Curve. Currently, Saddle's admin changes are implemented immediately, but the team plans to implement a 48 hour delayed Timelock contract for such changes in the future. They state that this will be based off of Compound's `Timelock.sol`.
- The order in which some mathematical calculations are performed has been altered – leading to **[L07] Loss of accuracy caused when dividing**. ***Update:*** *This issue has been fixed. See below.*

# Overall Health

As can be seen from the detailed report that follows, we found a number of `Medium` and `Low` severity issues, although no `High` or `Critical` vulnerabilities were found. This is generally the sign of a healthy and well-written codebase.

We found that throughout the codebase, large chunks of complex mathematics were left uncommented and unexplained. While this is in part because the code is merely a re-implementation of Curve's StableSwap, the code would greatly benefit from having such complex sections of code well explained and documented. Improving readability of the code is important to help future developers, auditors, and community contributors get to grips with the code quickly and easily, and will help all involved be more certain of the code's security.

It should be noted that the purpose of this audit was to ensure the Saddle contracts function in the same way as the Curve contracts. Since the mathematics in Saddle were not actually designed by the Saddle team, there could therefore exist nuances of the algorithm that Saddle are unaware of and that could become issues if Saddle choose to use this algorithm in a way that Curve do not. Additionally, any unknown vulnerabilities in Curve may similarly exist in Saddle. Conversely, Saddle benefits from the battle-tested Curve codebase, and by implementing as close to the same thing as possible, can leverage any previous auditing effort as it pertains to Curve.

# Privileged Roles

The Saddle contracts have an Owner role within the Swap contract, which will be controlled by a 3-of-5 Gnosis Safe multi-signature wallet. Details about multi-sig's structure and ownership can be found in the Saddle documentation. This multi-sig has the ability to:

- Pause the contract. This prevents all swaps and adding of liquidity, as well as disabling **some** functions for removal of liquidity. Note that liquidity can always be removed when paused via the `removeLiquidity` function, even when the contract is paused.
- Update the fees charged throughout the protocol.
- Update the protocol's "amplification coefficient" which influences how the pricing algorithm works.

Below are our audit results, in order of severity.

# Critical severity

None.

# High severity

None.

# Medium severity

## [M01] Pools may have duplicate tokens

Within the `Swap` contract, the check on line 104 is intended to prevent Saddle pools from having duplicate instances of the same token. However, this check will not correctly revert in the case that the token at index `0` is duplicated in any index `> 0`.

This is because at the end of the for loop on line 115, an element in `tokenIndexes` can be assigned a value of `0`, which is identical to the default value when an index in the mapping has not been assigned. Thus, the check that a new token has not been assigned an index will pass if this same token has already been assigned an index of `0`. Having duplicate tokens in a pool can lead to a plethora of issues when calculating swap prices and providing liquidity.

Consider modifying this pattern to catch the case where the token at index `0` is duplicated in another index. One solution may be to instead track `tokenIndexesPlusOne` rather than `tokenIndexes`, and assign values of `i+1`. This will ensure that all valid entries have a non-zero value, avoiding any confusion as to whether a value has been initialized or not. Note that this may necessitate changing the code within `getTokenIndex` to utilize this workaround. Another potential solution is to add an additional check to the constructor that each token with index `> 0` is not equal to the token with index `0`.

***Update:*** *Fixed in pull request 184.*

## [M02] Initial fee rates and amplification coefficient can be set out of bounds

Typically, when setting the following values, limits (such as this one in `setAdminFee`) are imposed:

- `initialA`
- `swapFee`
- `defaultWithdrawFee`
- `adminFee`

However, upon initializing a new Swap contract, these values are set directly, and can therefore be outside of the limits typically imposed. This could lead to errors in calculations of fees and of trade prices, and has the potential to freeze certain functions if fees are too high.

Consider ensuring that maximum limits are applied to the fee and amplification coefficient parameters of a new Swap contract, much as they are applied when the values are updated.

***Update:*** *Fixed in pull request 189.*

## [M03] Approximations may finish with inaccurate values

Within the `SwapUtils` contract, there are three spots where an iterative approximation is performed: within `getYD`, within `getD`, and within `getY`. In each case, the result of the `within1` function is used as the "breaking" condition. However, also in each instance, if the breaking condition is never satisfied, the `for` loop will eventually finish iterating after `MAX_LOOP_LIMIT` iterations. When this occurs, the estimated value will still be treated as correct, even though it could be relatively inaccurate.

Consider implementing some logic such that if the `for` loop finishes iterating and the breaking condition has not been met, execution reverts. This will avoid treating the approximated value as accurate in the case that it is not.

***Update:*** *Partially fixed in pull request 201. The instance within `getD` has been fixed.*

## [M04] Token fees can lead to incorrect calculations

Certain tokens, such as USDT, have the ability to charge fees on transfers. Although the fee rate is currently `0`, this may change in the future. The result of this is that if `x` tokens are transferred, an amount less than `x` will be received. For any functions which utilize `transferFrom`, the expectation that the amount received will match the amount transferred cannot be relied upon.

Within `SwapUtils` functions `swap` and `addLiquidity` the `safeTransferFrom` function is called to `transferFrom` tokens into the Saddle pool contract. While `safeTransferFrom` does provide error handling for unsuccessful calls, it will provide no indication of whether a token charges internal fees on transfers.

For tokens which may charge fees on transfers, consider checking the recipient's balance before and after `safeTransferFrom` calls, and using the change in balance for any calculations which may follow. In both cases mentioned above, the transfer of fee-charging tokens should occur before any relevant calculations are made, to avoid calculations which assume more tokens were received than actually were. For reference, the Curve 3pool Pool uses a variable called `FEE_INDEX` to identify USDT as a token which may charge fees, and incorporates `FEE_INDEX` into logic accordingly.

***Update:*** *Fixed in pull request 191.*

# Low severity

## [L01] Crude Allowlist check not secure

In the `constructor` of the `Allowlist` contract, `poolCaps[address(0x0)]` is set to `0x54dd1e`. The comment on the line above states that this value is "a way of crude checking whether an address holds [the] Allowlist contract". Then, in the constructor of the `Swap` contract, a check is done which requires that the inputted `_allowlist` contract has the value `0x54dd1e` stored in its `poolCap` mapping at key `address(0x0)`. This is to ensure that the `_allowlist` variable is the address of an instance of the `Allowlist` contract.

This check, while being potentially useful in preventing user error, cannot be relied upon for security. Any contract can easily implement a `getPoolCap` function such that this check succeeds. We report this so users and future developers are aware that this check only provides convenience, but cannot guarantee that the correct version of `Allowlist` is linked to the instance of `Swap`. To verify, advanced users could check that the value of the public `allowlist` variable correctly corresponds to the contract. Consider instead implementing EIP1820 in order to leverage and comply with industry standards, but note that even with EIP1820, any contract can assert that it implements the desired interface.

***Update:*** *Acknowledged, and will not fix. Saddle's statement for the issue:*

> This issue is acknowledged by the team. The check is primarily for preventing user error. Once allowlist is set on initialization, it cannot be changed. We will make sure that the allowlist contract is the right version and owned by the correct owner upon deployment of Swap.

## [L02] Low test coverage on some contracts

Running `npm run coverage` within the Saddle repository shows that some areas of the code base do not have thorough test coverage. In particular, there are no tests at all for the code within `CERC20.sol`, and the branch coverage for 6 of the 8 contracts audited is at 80% or below.

Consider writing additional tests for the repository to fully test your contracts. We advise having at least 95% test coverage.

***Update:*** *Fixed in pull request 197. The Saddle team has significantly increased test coverage.*

## [L03] Improper math correction in `_updateUserWithdrawFee`

In the `SwapUtils` function `_updateUserWithdrawFee`, a calculation is performed to determine the user's `withdrawFeeMultiplier`. In this calculation, the value `1` is added in two places to attempt to counteract the truncation effects of the `div` function. However, the way it is being done is incorrect, and can actually exaggerate math errors.

Since truncation occurs whenever a division is performed such that there is a "remainder" (when `A / B` is performed and `A % B != 0`), consider adding a `1` only after a division is performed, and perhaps only if there is a remainder in the `div` calculation. The other additions of `1` should be removed.

***Update:*** *Fixed in pull request 157.*

## [L04] No return values for some functions

Saddle's `Swap` contract interface is intended to closely match that of Curve's `SwapTemplateBase` contract. However, a number of functions return a value in Curve's implementation, while their equivalents in Saddle do not return a value. The following sets of functions return a value in Curve, but do not in Saddle:

- The `swap` function in Saddle and the `exchange` function in Curve.
- The `addLiquidity` function in Saddle and the `add_liquidity` function in Curve.
- The `removeLiquidity` function in Saddle and the `remove_liquidity` function in Curve.
- The `removeLiquidityOneToken` function in Saddle and the `remove_liquidity_one_coin` function in Curve.
- The `removeLiquidityImbalance` function in Saddle and the `remove_liquidity_imbalance` function in Curve.

Since these functions are supposed to mimic those of Curve, they should return values like the functions in Curve do. Importantly, this will provide those values for smart contracts that interact with the Saddle code to trade or provide liquidity, and will result in less confusion for users who are expecting a similar interface to that of Curve.

Consider implementing return values for the functions highlighted above.

***Update:*** *Fixed in pull request 196.*

## [L05] Outdated Solidity version in use

The codebase is currently using solidity version 0.5.17. As Solidity is now under a fast-release cycle, we advise using the latest version of the compiler at the time of deployment (presently 0.7.6). Alternatively, if there are reasons for using version 0.5.17, consider making this clear in the documentation.

***Update:*** *Fixed in pull request 185. Saddle chose to upgrade to Solidity version 0.6.12. Saddle's statement for this issue:*

> Solidity 0.7 fixes various issues, two notable bugs are the below ones.
>
> https://blog.soliditylang.org/2020/10/07/solidity-dynamic-array-cleanup-bug/
>
> https://blog.soliditylang.org/2020/10/19/empty-byte-array-copy-bug/
>
> However as our codebase does not rely on resizing an array without inserting values, we are not exposed to the bugs currently. List of known bugs and fixes can be found here
>
> https://docs.soliditylang.org/en/v0.7.6/bugs.html
>
> We will follow crytic's development guidelines and move to 0.6.12
>
> https://github.com/crytic/building-secure-contracts/blob/master/development-guidelines/guidelines.md#solidity

## [L06] Token precisions not verified

In the `constructor` of the `Swap` contract, `precisions` is an input parameter. This value is intended to be `10**decimals`, where `decimals` is the number of decimal places that the corresponding token has in the `_pooledTokens` array. However, the inputted values are not verified to be a power of 10, meaning that currently the constructor will accept any value. This would then impart calculation errors when `tokenPrecisionMultipliers` are used, since `tokenPrecisionMultipliers` is set to `precisions`.

Although it cannot be assumed that a `decimals` function is implemented for each token, since it is optional according to the ERC20 specification, the vast majority of tokens do have this property. Consider vetting tokens for the existence of `decimals` before listing them. This will then enable `precisions` to be assigned by referencing the token contracts instead of via user-supplied values, which are error prone.

***Update:*** *Fixed in pull request 195.* *`tokenPrecisionMultipliers` is now always set to a power of 10 based on `decimals`.*

## [L07] Loss of accuracy caused when dividing

Throughout the Saddle contracts, formulae used are mathematically equivalent to those performed in the Curve contracts. However due to the fact that division truncates in the EVM, a number of the Saddle functions contain calculations that are less precise than the Curve equivalents due to the fact that the Saddle contracts divide earlier or more often than the Curve contracts. Consider the following list of examples:

- Line 296 of `SwapUtils`' `getYD` is mathematically equivalent to line 632 of Curve's `get_y_D`. However Saddle achieves this by dividing the denominator by `A_PRECISION` instead of multiplying the numerator.
- Line 465 of `SwapUtils`' `getY` and line 402 of Curve's `get_y` are identical to the lines mentioned in the previous point. They therefore suffer from the same issue of "double dividing" as `getYD`.
- `SwapUtils.getD` lines 343-344 are mathematically equivalent to line 222 of Curve's `get_D`. The difference in the formulae comes down to Saddle performing `(nA/A_PRECISION - 1) * D` where Curve performs `((Ann-A_PRECISION)*D)/A_PRECISION` (where `nA` and `Ann` are equivalent variables). Dividing by `A_PRECISION` as the first operation, as Saddle does, causes a loss of precision that is then amplified by the following multiplication. Curve's method of dividing by `A_PRECISION` as the final operation means that the precision loss is as minimal as possible.

Consider updating calculations throughout the codebase to always perform divisions as rarely and as late as possible.

***Update:*** *Fixed in pull request 194.*

## [L08] Unclear comments and naming

Throughout the codebase, there are many spots in which variable names or comments are unclear. The examples we identified are:

- The constant `DENOMINATORS` and the mappings `multipliers`, `poolCaps`, and `accountLimits` within the `Allowlist` contract have no comments explaining their intended use.
- The variables `swapStorage`, `allowlist`, and `isGuarded` and the mapping `tokenIndexes` within the `Swap` contract have no comments explaining their intended use.
- The acronym `TVL` on line 84 and line 86 of the `Allowlist` contract is not explicitly stated.
- On lines 193 and 196 of the `SwapUtils` contract, `dy` is mentioned but isn't explained.
- The variables `xp`, `A`, `nA`, `D`, `dP`, `b`, `c`, `y`, `xpi`, `D0`, `D1`, `v`, `A0`, `A1`, `t0`, `t1`, `dx`, and `dy` within the `SwapUtils` contract are used frequently, but are not descriptive in their naming. Consider renaming such variables to be more descriptive, or highlighting their analog values in Curve.
- The choice of value `0x54dd1e` in `Allowlist` and `Swap` is arbitrary, and comments or documentation should exist to

indicate this so that it is clear the value has no particular meaning.

- The comments before `getYD` mention precision-adjusted balances duplicately. Consider removing one mention of precision-adjusted balances in this comment.

Consider adding comments throughout your codebase to ensure the purpose of every aspect is clear, including clarifying the above examples. Doing so will aid future development efforts and reduce the surface for error.

***Update:*** *partially fixed in pull request 202. The variables `xp`, `A`, and others identified in the 5th bullet have not been renamed. Saddle's statement for this issue:*

> External document/wiki will include information about which function is analogous to those in curve's

# Notes & Additional Information

## [N01] Efficiency improvements

While the following code inefficiencies are not vulnerabilities in the Saddle contracts, we have listed them for the Saddle team to consider as potential improvements to the code.

- Both `calculateWithdrawOneTokenDY`, and `removeLiquidityImbalance` both make 3 calls to `_getAPrecise` with an identical state. Consider instead making one call to `_getAPrecise` and storing the result to then reuse each time it is needed.
- `removeLiquidity` contains two `for` loops in a row that each loop through all tokens. Consider combining them into just one `for` loop that performs all of the necessary actions.
- `addLiquidity` calls `getD(Swap)` which itself calls `_getAPrecise`. `addLiquidity` then later calls `_getAPrecise` to pass into `getD(uint256[], uint256)`. Consider performing just one call to `_getAPrecise`, and passing this stored value into `getD(uint256[], uint256)` twice.
- A few times throughout `SwapUtils` the code requires that some `x < = y` before then using SafeMath to perform `y.sub(x)`. Given that SafeMath already performs checks that the subtraction cannot overflow, consider removing the `require` statement preceding the subtraction. Two examples of where this occurs are: line 586 of `calculateTokenAmount`, and line 850 of `removeLiquidityImbalance`. Note also that `SafeMath` provides a `sub` function which can apply a custom error message. Consider making the suggested changes to improve the efficiency of the contracts. ***Update:*** *Fixed in pull request 200.*

## [N02] Missing, incorrect, or incomplete NatSpec comments

Within the codebase, many functions have missing, incorrect, or incomplete NatSpec comments.

- The `transfer` and `transferFrom` functions of `LPToken` have no NatSpec comments.
- The `calculateTokenAmount` function of `SwapUtils` is missing an explanation of it's `return` parameter.
- The `self` parameter of many functions in `SwapUtils` is undocumented.
- The `tokenAmount` parameter of `removeLiquidityOneToken` is incorrectly labelled as *"the amount of the token you want to receive"*, when it is actually the amount of liquidity tokens to burn. The solidity documentation recommends NatSpec for all public interfaces (everything in the ABI). Consider implementing complete NatSpec for all `public` and `external` functions.

***Update:*** *Fixed in pull request 199.*

## [N03] Inconsistent coding style

The code base does not follow a consistent coding style.
Namely:

- The use of leading underscores to mark internal functions. For example, `_updateUserWithdrawFee` is `internal` and marked with an underscore, whereas `feePerToken` is not.
- The use of underscores to mark a function parameter. For example, function `_xp` has parameters `_balances` marked with an underscore, and `precisionMultipliers` with no underscore.
- The use of lowercase letters for variable names. For example, struct `CalculateWithdrawOneTokenDYInfo` has fields `D0` and `D1` beginning with uppercase letters, and fields `newY` and `feePerToken` beginning with lowercase letters.
- Inconsistent style of wrapping long function signatures. For example see the wrapping of `calculateTokenAmount` versus the wrapping of `calculateRemoveLiquidity`. Consider always following the same style to improve the project's readability. As reference, consider following the style proposed in Solidity's Style Guide. Taking into consideration how much value a consistent coding style adds to the project's readability, enforcing a standard coding style with help of linter tools such as Solhint is recommended.

***Update:*** *Fixed in pull request 203.*

## [N04] LPToken copying inherited code

Within the `LPToken` contract, the `transfer` and `transferFrom` functions re-implement the code from OpenZeppelin's `ERC20` contract functions `transfer` and `transferFrom`. Instead of re-implementing this code in `LPToken`, consider replacing the identified lines with calls to those functions, using `ERC20.transfer` and `ERC20.transferFrom` instead. This way, the code becomes more readable and makes it easier to upgrade in the future. Additionally, copying code carries with it the risk that the code is copied incorrectly, which these `super` calls can eliminate.

***Update:*** *Fixed in pull request 193.*

## [N05] `calculateCurrentWithdrawFee` `else` case missing an explicit return

The `calculateCurrentWithdrawFee` function within `SwapUtils` returns a user's withdrawal fee at the current time as a `uint256`. This withdrawal fee decreases over a 4 week time frame, after which 0 fee is charged. The function checks whether 4 weeks has passed, and if not returns the calculated fee. However in the case that 4 weeks *has* passed, the function does not explicitly return a value, instead relying on the fact that under the hood of Solidity the return parameter is initialized as 0. While the function returns the expected value in the case that 4 weeks have passed, it makes the function harder to read and understand. To increase readability, consider adding an explicit `return` statement when the `if` statement is not executed.

***Update:*** *Fixed in pull request 156.*

## [N06] Writing custom Pausable and ERC20Mintable contracts

The contracts `OwnerPausable` and `LPToken` implement custom logic to enable pausing of contracts and a mintable ERC20 respectively. Although this does not pose a security risk, consider always inheriting functionality from the secure, community-vetted, OpenZeppelin Contracts library. In particular, consider inheriting from OpenZeppelin's `Pausable` and `ERC20Mintable` contracts. This will help reduce the code's attack surface.

***Update:*** *Fixed in pull request 198.* `ERC20Mintable` *was not implemented, Saddle's reasoning for this:*

> As ERC20Mintable contract no longer exists in OpenZeppelin v3.0+, we left LPToken.sol as is.

## [N07] Typos in comments

Within the codebase there are some instances of typos in comments and docstrings. Some examples are:

- On line 18 of `Swap.sol` `get` should be `gets`.
- On line 42 of `Swap.sol` `fromm` should be `from`.
- On line 249 of `Swap.sol` `tokens` should be `token`.
- On line 169 of `SwapUtils.sol` `tokens to` should be `tokens`.
- On line 178 of `SwapUtils.sol` the word `calculation` should be removed.

Consider updating the lines identified above. Furthermore consider applying an automated spelling and grammar checker to your codebase to identify further instances.

***Update:*** *Fixed in pull request 192. Line 178 of `SwapUtils.sol` was removed.*

### [N08] Superfluous `else` clause

Within the `SwapUtils` function `getYD`, there exists an `if` clause and accompanying `else` clause. The `else` clause is superfluous, and can be removed. Consider removing it to simplify the codebase and improve code readability.

***Update:*** *Fixed in PR#188*

### [N09] Unnecessary inheritance in `LPToken`

The `LPToken` contract inherits `ERC20` along with `ERC20Burnable`. However `ERC20Burnable` inherits `ERC20` already, so there is no need to inherit it again in `LPToken`.

Consider removing the inheritance of `ERC20` from `LPToken`.

***Update:*** *Fixed in PR#187.*

### [N10] Declare `uint` as `uint256`

To favor explicitness, consider declaring the instance of `uint` in the `CERC20Utils` contract as `uint256`.

***Update:*** *Fixed in PR#186*

## Conclusions

0 critical and 0 high severity issues were found during this audit, and the contracts were found to function much the same as Curve's StableSwap implementation. Some less-severe issues were discovered and changes were proposed to follow best practices and reduce potential attack surface. We advise the Saddle team conducts further research into Curve's StableSwap algorithm to ensure all potential side-effects and pitfalls of using their algorithm are considered.

interested in smart contract security, you can continue the discussion in our forum, or even better, join the team 🚀

- If you are building a project of your own and would like to request a security audit, please do so here.

RELATED POSTS

OpenZeppelin

**Products**

Contracts

Defender

**Security**

Security Audits

**Learn**

Docs

Forum

Ethernaut

**Company**

Website

About

Jobs

Logo Kit