Solo Margin Protocol Audit

APRIL 30, 2019 | IN SECURITY AUDITS | BY OPENZEPPELIN SECURITY

The dYdX team asked us to review and audit their Solo project. We looked at the code and our results are published below.

The audited code is located in the `./contracts/protocol/` folder of the Solo repository. The commit used for this report is `17df84db351d5438e1b7437572722b4f52c8b2b4` .

Here are our assessment and recommendations, in order of importance.

Update: *the dYdX team made some fixes based on our recommendations. We address below the fixes introduced as part of this audit.*

Critical Severity

None.

High Severity

Contracts using the experimental ABIEncoderV2

The Solo project uses features from the ABIEncoderV2 of Solidity. This new version of the encoder is still experimental. Since the release of Solidity v0.5.4 (the one used by the Solo project), two new versions of Solidity have been released fixing important issues in this encoder.

Because the ABIEncoderV2 is experimental, it would be risky to release the project using it. Moreover, the recent findings show that it is likely that other important bugs are yet to be found.

As mentioned in the recent bug announcement, most of the issues of the encoder will have impact on the functionality of the contracts. So the risk can be mitigated by being extra thorough on the testing process of the project at all levels.

However, even with great tests there is always a chance to miss important issues that will affect the project. Consider also more conservative options, like implementing upgrade, migration or pause functionalities, delaying the release until the ABIEncoderV2 is stable, or rewriting the project to use the current stable encoder.

Update: *Statement from the dYdX team about this issue: "The AbiEncoderV2 has been used in production for months without incident by other high-profile protocols such as 0x Version 2. We do not see its use as a larger security risk than using the Solidity compiler in general. We have also upgraded the compiler version to v0.5.7 since beginning the Audit (which fixes the aforementioned bugs)."*

Malicious AutoTrader contracts may steal funds

The Solo contract allows a user to set any contract as their `AutoTrader`. If a user makes a trade with an attacker using a malicious `AutoTrader`, the attacker may front-run the trade with a transaction that changes the rate returned by the `AutoTrader`'s `getTradeCost()` effectively allowing the attacker to steal the full amount of the trade.

This can be prevented by only allowing users to interact with approved `AutoTrader` contracts on the front-end. However, it would be best to prevent this attack on-chain rather than relying on off-chain protections.

Consider adding a whitelist of `AutoTrader` contracts or `AutoTrader` factories to restrict the possible implementations on-chain.

Note: This issue was downgraded from critical severity because the dYdX team is aware of the issue and has plans for off-chain mitigation.

Update: *Statement from the dYdX team about this issue: "By using the `TradeData` field, `AutoTrader` contracts can be written so that they do not suffer from any of the security issues mentioned (front running or otherwise). The `ExchangeWrapper` contracts that we have been using in production for months are secured in this manner.*
As with all smart contracts, users should only use contracts that they trust; it is clearly unsafe to use any arbitrary address for an `AutoTrader`. Passing in the address of an `AutoTrader` is not less secure than specifying any other data in an Ethereum transaction. An on-chain whitelist of `AutoTraders` would not prevent malformed or malicious transaction data from producing unintended results."

Medium Severity

Missing docstrings

Some areas of the code base were difficult to understand or were not self-explanatory. The layout of the project makes this a bigger problem because the reader has to jump through many files to understand a single function.

Consider adding Natspec docstrings to everything that is part of the contracts' public API, including structs and struct fields. In addition to that, consider documenting in the code any side-effects of the functions, and the conditions that will make them revert. If a new concept or a business rule is introduced in a high-level function, consider briefly explaining it and adding a link to the user documentation for more details.

Update: *Fixed in pull request #234.*

Encapsulation issues make the code hard to read

Encapsulation is essential for writing clean and readable code. However, in a language like Solidity it is very hard to encapsulate the code. Contract oriented programming is not quite like object oriented programming, so its flexibility and

limitations affect the design of the project.

The Solo team decided to heavily use structs and libraries for encapsulation. This is a nice idea, but it brings its own challenges due to important details for understanding functions being spread across many files.

Most of the readability problems can be mitigated by adding extensive comments as explained in the *Missing docstrings* issue reported above. Some other parts can be improved by following the rule of "low coupling and high cohesion". And some others by making layers of code as minimal as possible, which has the added benefit of reducing the attack surface and making each layer easier to test.

Following are examples of encapsulation pain points that made it difficult to review the Solo code, or possible improvements for the encapsulation design.

- Rather than passing the entire `State` struct into functions, only pass in the specific pieces of state that will be operated on. This makes it easier to understand what a function is doing and ensures only the intended pieces of state are changed.
- There is a circular dependency between the `Storage` and `Cache` contracts. As suggested above, consider passing in only the necessary parameters rather a `State` struct to `Cache.addMarket()` to remove `Cache`'s dependency on `Storage`.
- There is a contract called `Getters`. This fails at cohesion because it is too generic and it is too far from the setter functions and the state it is querying. Consider moving all the getter and setter functions, and state variables they manipulate, to the same contract.
- The same functionality to revert if a market does not exist is implemented in two different contracts: `requireValidMarket` in `Getters.sol` and `validateMarketId` in `AdminImpl.sol`. Consider moving this function to a single place, either the `Storage` library or the `State` contract.
- The `getMarketCurrentIndex` function calls the `fetchNewIndex` function of the `Storage` library, passing as an argument the return value of a function of the same `Storage` library. Instead of calling two functions from the same library in a single statement, consider defining a new function `fetchCurrentIndex` in the `Storage` library.
- The `Admin` contract is just a thin layer that adds security modifiers to the `AdminImpl` contract where it forwards all the calls. This means that `AdminImpl` is not usable on its own because it is not safe. Consider moving all the implementations into the `Admin` contract and dropping `AdminImpl`. It could make sense to define an interface to specify the Administrator functions in a clear way, and to make it easy to have alternate implementations.
- When an index is updated, the corresponding event is emitted by `OperationImpl`. Consider emitting the event inside the `updateIndex` function. This would be a clearer assignment of responsibilities, and it ensures that it is not possible to update the index and forget to emit the event.

When the more readable design cannot be implemented because of the Ethereum contract size limitations, consider explaining this in the comments of the source code, and supplement the sub-optimal implementation with extra comments to guide the readers and future contributors.

Low Severity

README is missing important information

The README.md files on the root of the git repositories are the first documents that most developers will read, so they should be complete, clear, concise and accurate.

The README.md of the Solo project has little information about what is the purpose of the project and how to use it.

Consider following Standard Readme to define the structure and contents for the README.md file. Consider including an explanation of the core concepts of the repository, the usage workflows, the public APIs, instructions to test and deploy it, and how it relates to other parts of the project.

Make sure to include instructions for the responsible disclosure of any security vulnerabilities found in the project.

Update: *Fixed in pull requests #219 and #243.*

Allowed non-standard ERC20 tokens should be explicitly specified

Since non-standard ERC20 tokens are allowed to be used in markets, the behavior of these tokens should be explicitly specified in comments or the README. All ERC20 tokens that make up markets should abide by these specified conditions in order to be accepted as a market. Certain non-standard implementations may cause undesired effects on the dYdX contracts.

As mentioned in the comments in the code, "a custom ERC20 interface is used in order to deal with tokens that don't adhere strictly to the ERC20 standard (for example tokens that don't return a boolean value on success)". Because of this lack of return value, the code allows for a number of non-standard ERC20 implementations, rather than just the one mentioned in the comments.

An example of potentially vulnerable code can be found in Token.sol. `checkSuccess()` will return `true` if the specific ERC20 implementation neither throws nor returns `false` on `transfer()`, `transferFrom()`, or `approve()`, regardless of the outcome of the transaction.

This ERC20 implementation would cause issues with Dapps other than dYdX, so it is expected that this type of token never makes it into production on the main Ethereum network. Nevertheless, we suggest being explicit about the types of tokens that are allowed to make up a market and checking that tokens meet these conditions prior to being accepted as a market.

Global operators are not restricted to contract addresses

The Solo contract allows "global operators" to operate on behalf of any account. The motivation behind the global operator feature is to allow for things such as a wrapped Ether proxy and automatic loan expiry. Because the intention is for the global operator to always be a contract, consider adding a sanity check using OpenZeppelin's `isContract()` function to ensure regular accounts can not be added as global operators and to be more explicit about the intention of the feature.

There are magic constants in the code

There are magic constants in several Solo contracts. For example, Require.sol, line 203 and Require.sol, line 207. These values make the code harder to understand and to maintain.

Consider defining a constant variable for every hard-coded value (including booleans), giving it a clear and explanatory name. For complex values, consider adding a comment explaining how they were calculated or why they were chosen.

Update: *Comments were added to the constants in pull request #233.*

stringify() for bytes32 may unexpectedly truncate data

In Require.sol, the `stringify()` function for the `bytes32` type may unexpectedly truncate data. The function is meant to take `bytes32` data and truncate any trail zero bytes. However, the function will truncate the data at the first zero byte. A zero byte may appear in the middle of `bytes32` data causing all data after it to be truncated. Consider iterating the `bytes32` from the end of the array and truncating the data after the first non-zero byte to avoid truncating data unexpectedly.

Update: *Fixed in pull request #214.*

Interest rate calculation may be error prone

The Solo contracts calculate interest accumulated over time by incrementing an `index` which represents the total accumulated interest with a starting value of 1. The index is updated by taking the per-second interest rate and multiplying

by the number of seconds elapsed since the last time the index was updated (`percentageInterestSinceLastUpdate = perSecondRate * (currentTime - lastUpdatedTime)`). This number represents the percentage gained since the last calculation and is multiplied by the previous index value to calculate the updated index value (`index = index * (1 + percentageInterestSinceLastUpdate)`).

This calculation differs from the true calculation which would calculate `percentageInterestSinceLastUpdate` like so: `percentageInterestSinceLastUpdate = (currentTime - lastUpdated) ^ marginalRate`. The differences between the calculation used and the true calculation are negligible when the `index` is updated fairly frequently but start to diverge as the `index` is updated less frequently. Consider implementing the true interest calculation or properly documenting the current interest calculation.

Update: *The function was better documented in pull request #218.*

Nonreentrant functions should be marked external

As stated in ReentrancyGuard.sol, "calling a `nonReentrant` function from another `nonReentrant` function is not supported." All nonreentrant functions should be marked as `external` rather than `public` to enforce this more explicitly.

However, Solidity does not yet support structs as arguments of external functions (see issue #5479). Consider adding a comment to prevent developers to call these `nonReentrant` functions from the same contract.

Using Wei to refer to the minimum unit of a token

The Solo project uses the word "Wei" to refer to the minimum and indivisible unit of a token. 1 wei in Ethereum is equal to 10^-18 ether. While most tokens follow the same convention of having a "human-readable" unit with 18 decimals, many tokens define a different number of decimals. In addition to that, most tokens leave their minimum unit nameless, using the prefix of the International System of Units to refer to it (for example, attoToken for 1 token * 10^-18), instead of calling it Wei. This important detail is only mentioned once in the codebase.

There is no consistent way to call this minimum unit, and it could be very confusing to use Wei when the token has a different number of decimals. Consider using an alternative name that is clearer and less loaded, like BaseUnit or (as Matt Condon has suggested) TokenBits. Also consider documenting the expected unit on all the functions that receive a token amount as an argument.

Multiple operations in single statement

To increase code readability, avoid combining independent operations into a single line of code. In Require.sol, in the `stringify(uint256)` function, the variable `k` is decremented on the same line as it is used to access an array. Consider decrementing `k` on the line following the array access.

Update: *Fixed in pull request #214.*

Not following the Checks-Effects-Interactions Pattern

Solidity recommends the usage of the Check-Effects-Interaction Pattern to avoid potential problems, like reentrancy.

In a couple of places the code of Solo the checks are not done first:

- The invalid oracle price check in `setPriceOracle`.
- The primary account check in `verifyFinalState`.

While in these cases there is no risk of reentrancy, consider moving the checks to the start of the corresponding block to make sure that no issues are introduced by later changes.

Update: *Partially fixed in pull request #193. The dYdX team does not plan to update the `verifyFinalState` function.*

Unexpected return value from `Time.hasHappened()`

The `hasHappened()` function in the Time library will return `false` if the time is `0` instead of reverting. In the future, this may lead to a false positive if the function is being used to check if something hasn't happened yet. Consider reverting when the input value is `0`.

Update: *Fixed in pull request #220.*

Unused import

In Monetary.sol it is unnecessary to import the SafeMath and Math libraries as they are never used.

Update: *Fixed in pull request #210.*

Notes

- Some contract files include the `pragma experimental ABIEncoderV2;` (for example, `Admin.sol`) and some others do not include it (for example, `Decimal.sol`). The effect of declaring the experimental pragma only on some files is not very clear. Consider declaring the usage of `ABIEncoderV2` on all the Solidity files, for consistency, to make it clear that the new version of the encoder is used in the project, and to avoid any complications that can come for not declaring it in some files.
  Update: *Fixed in pull request #229.*
- Explicitly type cast integers before making a comparison between different types in the following locations:
  In Math.sol:
  – line 77
  – line 93
  – line 109
  In Types.sol:
  – line 98
  – line 102
  – line 105
  – line 211
  – line 215
  – line 218
- In Decimal.sol `add()` takes two `D256` and returns a `D256`. However, `mul()` and `div()` take a `uint256` and a `D256` and return a `uint256`. This may be confusing when comparable libraries such as SafeMath have consistency across arithmetic functions.
- All the copyright headers include "Copyright 2018 dYdX Trading Inc." According to the Free Software Foundation, "You should add the proper year for each past release; for example, 'Copyright 1998, 1999 Terry Jones' if some releases were finished in 1998 and some were finished in 1999." Consider updating the copyright headers to add the year 2019.
  Update: *Fixed in pull request #223.*
- There are a couple of typos in the comments:
  – `IErc20.sol` L26 and `Token.sol` L167: "dont" instead of "don't".
  – `SoloMargin.sol` L34: "inherets" instead of "inherits".
  Consider running codespell on pull requests.
  Update: *Fixed in pull request #225.*
- To favor explicitness and readability, several parts of the contracts may benefit from better naming. Our suggestions are:
  – `ttype` to `type` in `Actions.sol`, line 152 and `Actions.sol`, line 172.
  – `x` to `number` in `Math.sol`, lines 69, 85 and 100.
  – `r` to `result` in `Math.sol`, lines 75, 91 and 107.

– In `Admin.sol` and `AdminImpl.sol`, drop the word "owner" from the function names. For example, `ownerWithdrawExcessTokens` to `withdrawExcessTokens`.
– `OperatorArg` to `Operator`.
– `operator` to `account`.
– `getAccountValues` to `getAccountSupplyAndBorrowValues` in `Storage.sol`, line 296 and `Getters.sol`, line 319.
– `getAdjustedAccountValues` to `getAccountSupplyAndBorrowValuesAdjusted`.
– `getIsLocalOperator` to `isLocalOperator`.
– `getIsGlobalOperator` to `isGlobalOperator`.
– `g_state` to `globalState`.
– `arg` to `action`.

Update: *Partially fixed in pull request #228. The dYdX team prefers to keep some of these variable names.*

Conclusion

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

The code of the contracts was carefully written by the Solo team, following a consistent and nice style, considering all the possible conditions, and with extensive tests. The idea of their protocol is very interesting, and the way they implemented it simplifies many details that in other similar projects become hard to understand, test, and audit.

However, the use of structs and libraries, the shared global state and the side-effects to keep it up-to-date, the split of responsibilities between multiple contracts (sometimes forced by Ethereum limitations), and the lack of comments, made the codebase hard to read and navigate, forcing us to jump through many files to fully understand every function. In addition to that, the functions did not specify their expected results, making them harder to audit for correctness. Most of these problems can be solved or mitigated by simply adding more comments to guide the readers, or with small tweaks of the design.

An important thing to notice for readers of this report and users of the Solo system is that, while most parts are non-custodial and allow free peer-to-peer interactions, the administrators are in full control of the fundamental parameters of the system. Also, to improve the usability and usefulness of the system, the Solo team decided to implement global operators that will be able to execute actions that can affect user accounts without waiting for their approval. These are important and necessary decisions to build a functioning system. The Solo team has ensured the transparency of their system, and can easily implement time delays, present notifications on the user interface, and document every aspect of the system, to make sure that their users will have a clear idea of what to expect, what to monitor, and how to take full advantage of the available features.

Note that as of the date of publishing, the above review reflects the current understanding of known security patterns as they relate to the Solo contracts. The above should not be construed as investment advice. For general information about smart contract security, check out our thoughts here.

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

ADD COMMENT

Name *

Email *

Website

Save my name, email, and website in this browser for the next time I comment.

Notify me of follow-up comments by email.

Notify me of new posts by email.

POST COMMENT

SECURITY AUDITS

e