Stackly

Final Audit Report

June 20, 2023

Team Omega

teamomega.eth.limo

Summary

Stackly has asked Team Omega to audit their DCA Contracts.

We found **no high severity issues** - these are issues that can lead to a loss of funds, and are essential to fix. We classified **1** issue as “medium” - this is an issue we believe you should definitely address. In addition, **5** issues were classified as “low”, and **5** issues were classified as “info” - we believe the code would improve if these issues were addressed as well.

Severity	**Number of issues**	**Number of resolved issues**
High	0	0
Medium	1	1

Low	5	5
Info	5	4

Scope of the Audit

Scope of the Audit
The scope of the audit concerns an DCA (Dollar-Cost Averaging) tool that utilizes the CoW (Conditional Order Workflow) Protocol to place TWAP (Time-Weighted Average Price) orders.

The contract code is developed here:

```
https://github.com/SwaprHQim/stackly/tree/development/packages/contracts
```

And specifically the following files

```
src
├── DCAOrder.sol
├── OrderFactory.sol
├── interfaces
│   ├── EIP1271Verifier.sol
│   ├── IConditionalOrder.sol
│   ├── IDCAOrder.sol
│   └── IGPv2Settlement.sol
└── libraries
    └── GPv2Order.sol
```

These are about 310 “real lines of code” (i.e. excluding comments and empty lines).

We audited the following commit hash:

```
a729eb790300cfa2cef36d55e91ac3c99f46c92e
```

Resolution

The Stackly team subsequently addressed the issues in the following commit:

```
3e44abc1f98b592f705b2676971f86ebd4b3c10f
```

We have reviewed the fixes and reported our findings in this report at the end of each issue.

Methods Used

Code Review

We manually inspected the source code to identify potential security flaws.

The contracts were compiled, deployed, and tested in a test environment.

Automatic analysis

We have used static analysis tools to detect common potential vulnerabilities. The tools have detected a number of low severity issues, concerning mostly the variables naming and external calls, were found. We have included any relevant issues below in the appropriate parts of the report.

Disclaimer

The audit makes no statements or warranties about utility of the code, safety of the code, suitability of the business model, regulatory regime for the business model, or any other statements about fitness of the contracts to purpose, or their bug free status. The audit documentation is for discussion purposes only.

Severity definitions

High	Vulnerabilities that can lead to loss of assets or data manipulations.
Medium	Vulnerabilities that are essential to fix, but that do not lead to assets loss or data manipulations
Low	Issues that do not represent direct exploit, such as poor implementations, deviations from best practice, high gas costs, etc

Info	Matters of opinion

Findings

G1. Fee calculations use wrong denominator [medium] [resolved]

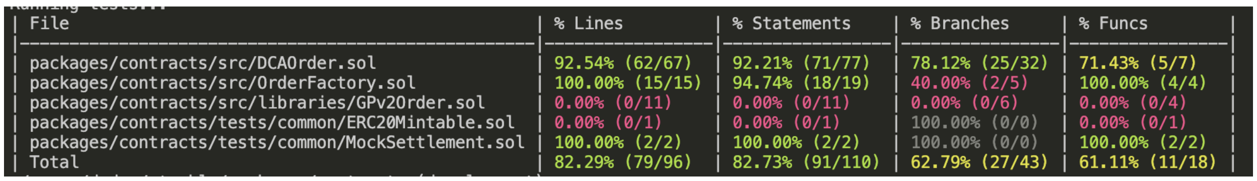
According to the documentation, the calculation of fee percentage should use 10,000 as 100%, meaning 1 would equal 0.01%. The `setProtocolFee` function follows this behavior, allowing to set the `protocolFee` percentage to be a number up to 10,000. However, in all calculations of the fee (lines 56, 59 in OrderFactory.sol and line 115 in DCAOrder.sol), the denominator used is 100, when it should be 10,000. This means the fee taken will be 100x higher than the fee intended.
Recommendation: Replace the division by 100 with 10,000 in the mentioned lines. You could also define a HUNDRED_PERCENT constant and set it to 10,000, then use it instead of putting the literal number multiple times.
Severity: Medium
Resolution: The issue was resolved as recommended.

G2. Success of token transfers is not verified [low] [resolved]

The contracts perform token transfers using `transfer` and `transferFrom` in several places. The ERC20 standard does not require that unsuccessful transfers revert, but rather that these functions return `false` if the transfer is not successful. The returned value of the `transfer` and `transferFrom` calls is not checked, which means a transfer might fail, but the transaction will not revert.
Recommendation: Use OpenZeppelin `safeTransfer` and `safeTransferFrom` methods to ensure token transfers' success.
Severity: Low
Resolution: The issue was resolved as recommended.

G3. Test Coverage is incomplete [low] [not resolved]

The test coverage is incomplete:



```
| File                                              | % Lines         | % Statements     | % Branches      | % Funcs         |
|---------------------------------------------------|-----------------|------------------|-----------------|-----------------|
| packages/contracts/src/DCAOrder.sol               | 92.54% (62/67)  | 92.21% (71/77)   | 78.12% (25/32)  | 71.43% (5/7)    |
| packages/contracts/src/OrderFactory.sol           | 100.00% (15/15) | 94.74% (18/19)   | 40.00% (2/5)    | 100.00% (4/4)   |
| packages/contracts/src/libraries/GPv2Order.sol    | 0.00% (0/11)    | 0.00% (0/11)     | 0.00% (0/6)     | 0.00% (0/4)     |
| packages/contracts/tests/common/ERC20Mintable.sol | 0.00% (0/1)     | 0.00% (0/1)      | 100.00% (0/0)   | 0.00% (0/1)     |
| packages/contracts/tests/common/MockSettlement.sol| 100.00% (2/2)   | 100.00% (2/2)    | 100.00% (0/0)   | 100.00% (2/2)   |
| Total                                             | 82.29% (79/96)  | 82.73% (91/110)  | 62.79% (27/43)  | 61.11% (11/18)  |
```

Recommendation: We recommend testing all your code. In particular, it would be good to add some tests for the behavior of the function `DCAOrder.isValidSignature`, which is essential for the functioning of the code, and is not tested at all.
Severity: Low
Resolution: The issue was not resolved. Test coverage was increased but is not complete.

G4. Upgrade Solidity to the latest version [info] [resolved]

The project uses version 0.8.17 of the solidity compiler, while the latest version available is 0.8.19.
Recommendation: We recommend upgrading to the latest version, which as of writing is 0.8.19.
Severity: Info
Resolution: The issue was resolved as recommended. The contracts now use Solidity version 0.8.20.

G5. Upgrade OpenZeppelin to the latest version [info] [resolved]

The project uses version 4.8.1 of the OpenZeppelin library, while the latest version available is 4.8.3.
Recommendation: Use the most recent version, which is 4.8.3 at the time of writing.
Severity: Info
Resolution: The issue was resolved as recommended.

DCAOrder.sol

D1. DCA Orders do not specify buyAmount [low] [not resolved]

On line 166, where the `GPV2Order` object is constructed, the order gets a value of 1 wei for the `buyAmount`. This means that it is theoretically possible that the order will get filled for any amount.

CowSwap has mechanisms in place that will make it (very) likely that an order will be filled at or close to the current market price. However, filling orders at market price is not in any way guaranteed - the

CowSap orchestration of solvers is complex and may fail. The discussion here: https://forum.cow.fi/t/ideas-for-ensuring-fair-solutions-envy-freeness-of-amms-and-capped-surplus/1029 is particularly relevant.

The Stackly orders are more vulnerable to attack also in light of the fact that the orders and their timing are publicly known, as we note in D5.

Also, even if the order is filled at “market price”, the order has no slippage protection. This means that also when the CoWswap is working well, the order could be filled with any amount of slippage when selling or buying tokens that have low liquidity.

Recommendation: There are several possible approaches here. One solution is to integrate an oracle, and set a `buyAmount` at or close to the price reported by the oracle on each slot/order. An approach that is easier to implement and that partially mitigates the risks, is to let the user specify a minimum price for the entire TWAP order and use that as the basis for calculating the `buyAmount` of each individual order. The last option is to do nothing and count on CowSwap internal mechanisms to fill the order at a good price.
Severity: Low
Resolution: The issue was not resolved.

D2. Unnecessary check order execution time is not greater than end time [info] [resolved]

In line 154 there is a check that will make the function revert in case the `orderExecutionTime` is greater than the `endTime`. But `orderExecutionTime` can never be greater than the `endTime`, and so the check is unnecessary.
Recommendation: Remove the check in line 154.
Severity: Info
Resolution: The issue was resolved as recommended.

D3. Start time minimum requirement does not match documentation [info] [resolved]

The comment in line 96 states that the start time must be at least 10 minutes in the future, but the code requires it to be only 3 minutes in the future.
Recommendation: Either update the comment or the code to be consistent.
Severity: Info

Resolution: The issue was resolved as recommended. The documentation was updated.

D4. DCA Order size and timing is publicly known [info] [not resolved]

The order generation and signing happens entirely on-chain. This means that all information about the order, including the amount and exact time it will become active, is public. This leaves the order vulnerable to front-running, especially if the trade is large or the liquidity of the token is low: an attacker can try to deflate the price just before the order is submitted, and then buy these tokens for this lower price.
Recommendation: One solution would be to sign orders off-chain and submit them to the Cowswap at semi-random intervals. However, this would imply a completely new approach and a complete refactor of the current code. The front-running problem is rather theoretical and only applies to low liquidity tokens; in addition, CowSwap solvers can use any type of liquidity, and so price manipulation needs to occur in all markets where the token is traded. Our recommendation is to leave the approach as-is, but make sure the contracts are only used to submit orders that are small relative to the total liquidity of the tokens traded.
Severity: Info
Resolution: The issue was not resolved.

OrderFactory.sol

F1. Owner can steal all funds on order creation [low] [resolved]

When a user creates an order by sending a `createOrderWithNonce`, the owner of the contract can front-run that transaction and call `setProtocolFee` to set the fee to 100%. This would mean that the entire order will be transferred to the factory contract, and the owner can call `withdrawTokens` to claim these tokens.
Recommendation: One mitigation is to set the maximum fee (way) lower than 100%, which in any case is not a setting that would be needed in any reasonable scenario. Other mitigations can be implemented on the owner account itself, which could be, for example, a timelock contract, which would make front-running very difficult.
Severity: Low
Resolution: The issue was resolved as recommended. The contract now limits the maximum fee to 5%.

F2. It is possible to create orders with a different fee [low] [resolved]

The function `createOrderWithNonce` takes an `initializer` argument, which can be any byte string. The contract, though, expects a very specific (and unusual) string, namely the abi-encoded call to the `initialize` function with all its arguments except the last one. In `createProxy`, then, the fee is added to the end of the `initializer` string, which is then passed to the newly created DCAOrder proxy contract.
The code does not implement any checks on the form or length of the `initializer` string. Specifically, calling `createOrderWithNonce` with an `initializer` string that includes a value for the fee will not revert. If this fee is different from the `protocolFee` set in the OrderFactory, the DCAOrder contract will try to sell more, or less, tokens than the amount that is sent to the order on creation, which may give unexpected results.
Recommendation: To resolve this specific issue, probably the most attractive approach is to remove the `_fee` argument, together with all calculations regarding the fees, from the DCAOrder contract. Instead, just pass it the amount it is expected to sell. In general, however, we think it is better to rely on Solidity's built in type checking: the code would be clearer, and more robust, if, for example, `createProxyWithNonce` takes all the parameters of `DCAOrder.initialize` plus the salt, instead of a generic byte string.
Resolution: The issue was resolved as recommended.

GPv2Order.sol

This file is, except for formatting, an exact copy of
https://github.com/gnosis/gp-v2-contracts/blob/main/src/contracts/libraries/GPv2Order.sol