DDA DRCT Standard Audit

May, 2018

Alexander Wade, Alex Towle, Nikhil Sakhamuri

Contents

1. Overview

This document serves as the official audit report of a set of factory, token, and swap contracts, created and published by the Decentralized Derivatives Association. DDA seeks to create a decentralized platform for derivatives trading, where authorized users can create their own swap contracts.

The project audited is the on-chain trading platform: a series of contracts that will be deployed to the main Ethereum network, and will contain an exchange through which users can purchase DRCT tokens from existing swaps as well as a swap contract that pays out holders of DRCT tokens after a swap has ended.

2. Introduction

2.1 Authenticity

The audited contracts are in the Decentralized Derivatives Association DRCT_Standard repository:

https://github.com/DecentralizedDerivatives/DRCT_standard

The version used for this audit is commit 2b65d2d22f26e884a176ed1629d649c5c6297494.

2.2 Scope

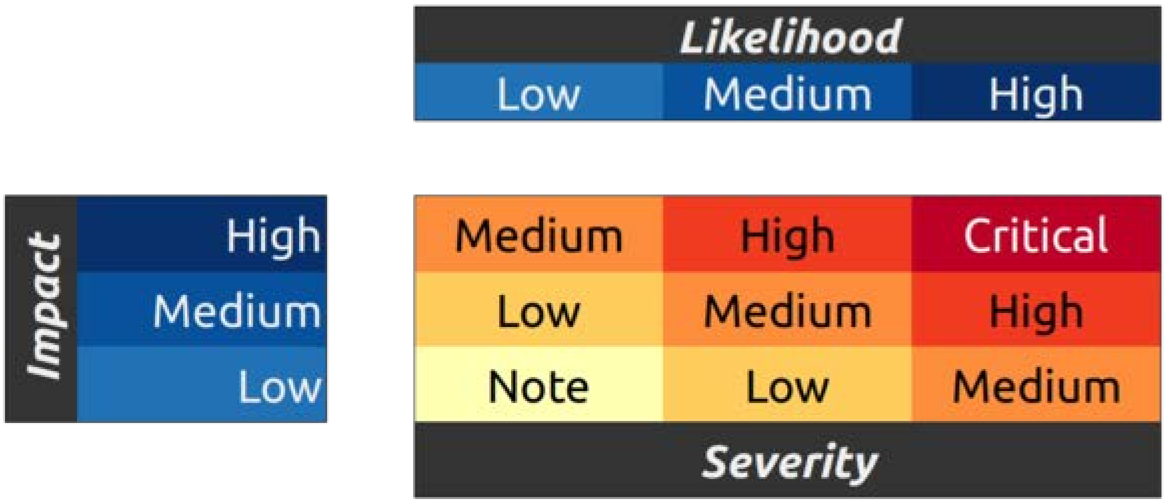
This audit covered all of the solidity files located in the contracts folder of the DRCT_Standard repository: https://github.com/DecentralizedDerivatives/DRCT_standard/tree/master/contracts

This audit does not cover files located in any other folder.

2.3 Methodology

This audit focuses heavily on not only inspecting the smart contracts for vulnerabilities and potential for losses in funds, but also on working closely with the DDA team to scrutinize the contracts for execution of intent. The end goal of this audit is to help the team not only secure their contracts, but also to ensure their vision for the project is best represented by the project they put forward. As a result, additional concerns such as efficiency and design are included in this report as well.

2.4 Terminology



	Likelihood		
Impact	Low	Medium	High
High	Medium	High	Critical
Medium	Low	Medium	High
Low	Note	Low	Medium
	Severity		

This audit categorizes vulnerabilities using the OWASP risk rating method based on impact and likelihood. Each vulnerability is given impact and vulnerability scores, which are used to give a more accurate estimation of a vulnerability's overall severity. An additional factor in severity is the relative ease with which a vulnerability is fixed: an issue which requires extreme refactoring will be weighted higher than one with the same severity which is a quick fix.

2.5 Disclaimer

This document reflects the understanding of security flaws and vulnerabilities as they are known to the Authio team, and as they relate to the reviewed project. This document makes no statements on the viability of the project, or the safety of its contracts. This audit is not intended to represent investment advice and should not be taken as such.

3. Findings

3.1 General

Overall, the implementation presented by the DDA team was thorough and comprehensive. Their code proved that with some changes and fixes, this implementation could prove to become a robust platform for derivative trading on the main Ethereum network. However, there was one critical severity issue in the payout functionality that could cause major issues if certain functions are called incorrectly. Additionally, the documentation surrounding the payout functionality will lead the user to call these functions improperly, triggering the vulnerability.

Outside of this flaw, there were but a handful of minor issues that can be corrected easily. Suggestions made in the course of this audit attempted to best remedy these issues while not compromising the integrity of the DRCT project.

3.2 Contract Explanation

3.2.1 Function - Token

`DRCTLibrary.sol`**:** This library houses all of the logic for any DRCT token. It contains the logic for adding a swap to a given token, as well as the logic for basic transfer, transferFrom, and approve functions for a token. It also contains all getters for the token and its related swaps.

`DRCT_Token.sol`**:** This contract invokes all of the logic that is written in DRCTLibrary and represents the given DRCT token in interactions with external contracts.

`WrappedEther.sol`**:** An alternative to using an ERC20 token, this contract gives the user the ability to create the base token for a swap using some of their personal ether.

3.2.2 Function - Token Swap

`TokenLibrary.sol`**:** This library houses all of the logic and functions needed to implement a token swap including the ability to start and create a swap. It also houses the logic to pay out participants in a specified swap.

`TokentoTokenSwap.sol`**:** This contract invokes all of the logic that is written in TokenLibrary and is called by other contracts in order to initiate and interact with a swap.

`Exchange.sol`: This contract allows users to place a certain amount of a DRCT token up for sale at a specified price, facilitating direct sale of DRCT tokens between users.

3.2.3 Function - Swap Contract Registration and Deployment

`Factory.sol`: This contract deploys and registers all of the contracts to be used during the course of the derivative contract. Standardized variables about the swap contract are set within the Factory contract.

`UserContract.sol`: This contract simplifies the swap registration process within the Factory contract by creating an interface that uses some user input and some data available through the Factory contract to properly register the swap contract.

3.3 Critical Severity

Faulty payout in forcePay: After a swap contract has ended, anyone can call `forcePay` with the `_begin` parameter being set to a number greater than the lowest index of token holders that have not been paid. This will lead to the non-payment of all token holders with an index lower than `_begin` that haven't been paid already paid, and the swap being set to an `ended` state. All of the remaining funds--some of which belong to the token holders--will be transferred from the swap to the factory address. In the worst case scenario, this vulnerability could lead to none of the DRCT token holders being paid out, and all of the collateral ether being seized by the Factory contract.

To fix some of these issues, the function parameters to `forcePay` should be changed to accept the `SwapStorage` storage pointer **`self`** and an unsigned integer **`amount`** that represents the maximum amount of either kind of DRCT Token holders to pay out. Two unsigned integer variables should be added to this swap's `SwapStorage` struct to represent the number of long token holders and short token holders that have been paid out. During each call to `forcePay`, these variables should be updated. Whenever `forcePay` is called, the function should start paying out at the index equal to the number of token holders paid out added to one. At the end of the `is(ready)` clause, there should be a check included to determine whether or not the number of long token holders and short token holders paid out are equal to their respective token contract's `addressCount` value. Finally, the factory should be paid and the swap can be finished.

Update: This issue is fixed per our advice. The function should still be refactored to make it more readable and maintainable.

3.4 High Severity

No high severity issues were found.

3.5 Medium Severity

oracle callback reference rate error: Anyone can call the oracle's function `pushData`. If the oracle has not been queried by the end of the day, a user may query the oracle right before the change of day, such as at 11:59 PM. This opens up the possibility that the API callback will be returned on the day after the original query. The queried mapping value at the index of the first day would be set to `true`, but the `oracle_values` mapping at the index of the first day would be left as `0`. This has the potential to cause either the short or the long token payout to be equal to `0` despite the fact that `0` may not be the correct starting or ending reference rate.

This issue can be fixed by creating a struct with two fields: the `queryID` and the date that the query was sent. Then, we can replace the `queryID` private variable with a private struct variable. In `pushData`, this struct will be used to check that the `queryID` of the callback is correct and the date to use as a key when storing the value in the `oracle_values` mapping.

Update: Advice was taken and `QueryInfo` struct was created in order to fix the previous hazard in which there was possibility that the API callback will be returned on the day after the original query.

3.6 Low Severity

WrappedEther is not ERC20 compliant: The `WrappedEther` contract does not contain a function `totalSupply` that is public and view that returns an unsigned integer. Since the ERC20 specification includes this function, `WrappedEther` is currently not ERC20 compliant, and moreover, it is not compliant with its own interface, `WrappedEther_Interface`. This issue could pose problems when using `WrappedEther` as a base token contract; however, the fix for this issue is very simple. Simply add a getter for the total supply with the correct function signature: `function totalSupply() public view returns (uint)` to the `WrappedEther` contract.

Update: Added the public getter `totalSupply` in order to become `ERC_20` compliant and compliant with its own `Wrapped_Ether`interface.

Member overwriting is not checked in Membership: In the `Membership` contract, the `updateMemberAddress` function can be used by the owner to overwrite a current member's information if the **`_from`** parameter is a member and the **`_to`** parameter is also. However, even if **`_to`** is a member, it can be overwritten by a **`_from`** address that does not correspond to a member. The member with address **`_from`** should be a member and address **`_to`** should not correspond to a member.

A simple check for membership at the beginning of `updateMemberAddress` would fix this issue.

Update: Added a require statement that ensures that the `_to` address is not currently in the members mapping in order to fix the flaw that allowed member overwriting.

Integer underflow in forcePay: If the second element of the **`_range`** parameter is equal to `0`, then the variable **`loop_count`** will be set equal to zero both times it is set. This will cause an underflow in both for loops that will almost always cause an out of gas exception to be thrown. After refactoring `forcePay` using the above suggestions, this underflow could still be an issue. There should be a check in place to ensure that the unsigned integer **`amount`** being passed in is not zero to prevent an underflow from occurring.

Update: Issue fixed, Integer underflows will not happen anymore.

3.7 Notes and Recommendations

Exchange - List function: The `userOrders` mapping does not have its entries 1-indexed, meaning that a 0 zero entry is not pushed to the beginning of every `uint[]` entry. This is inconsistent to the way arrays of similar nature are set up in this contract and in other contracts.

Update: Recommendation to 1-index the userOrders mapping was not taken, this does not affect proper functionality- simply a style inconsistency.

Exchange - Buy function: The buy function does not check for a valid `_orderId` input. This could get past the first two require statements if msg.sender is not blacklisted and sends no wei with the function call. This would get caught in the `assert` statement within the if-clause but it would be better to fail immediately by calling the order struct corresponding to `orders[_orderId]` and checking that none of this struct's attributes == 0.

Update: Recommendation to check for valid `_orderId` input was taken and the necessary require statements were inputted.

Exchange - UnLister function: The way that the function is currently written, in the case that `_order` is the only order in it's token's `forSale` entry, then the first 5 lines of code are unnecessary- simply re-assigning the last order struct to itself within `forSale` and re-assigning it's index to itself in `forSaleIndex`. To remedy this redundancy in this case, the if-statement can be moved to the first line of the function and changed to read `(forSale[_order.asset].length == 2)` checking for the existence of only the 0-order-struct and `_order`. Then, change the last line of the if-statement to be `forSale[_order.asset].length -= 2`. Then the first 6 lines of the code in the function can then be refactored as a corresponding else-statement for this if-statement. Lines 7-8 of the function can then be placed after the if-else clauses so that they are executed in both cases.

Update: This function was refactored as suggested in order to save gas in the case that the order to unlist was the only order in its token's `forSale` entry.The other suggestion was ignored, however this does not impact functionality.

Exchange - UnLister function: Similar to the case in which `_order` is the only order associated with a given token, in the case that `_order` is the only order associated with a given user there is no means to remove that user from the `userOrders` mapping.

DRCTLibrary - createToken function: Although the checks for valid input are already done in other functions that in turn call `createToken`, it would increase safety to add a check that ensures that `_swap` has not already been added into the `tokenStorage`struct (i.e. in `swap_balances`) prior to this call- because if not then the `swap_balances` entry for `_swap` would not be functional.

Update: Recommendation to add additional checks for valid input were ignored, this does not impact functionality however since the requisite checks are done by the functions that call `createToken`. However, it would improve safety to add checks within the function itself.

DRCTLibrary - Pay function: The function should check if it is passed valid inputs by ensuring that `_party` is a valid token owner in `_swap`. Although `_party` should be a valid token owner in `_swap` due to the nature it is obtained in `forcePay`, due to the importance of the pay function it would be a good idea to have an independent check in pay in order to prevent any unexpected behavior from other functions.

Update: Recommendation to add additional checks for valid input were once again ignored. The absence of these checks should not affect functionality since the appropriate checks are done in `paySwap` and `forcePay`, but due to the importance of this function it wouldn't hurt to add additional checks within its own body.

WrappedEther - transfer function: Since the if-statement at the beginning of this function checks that the balance of the sender is greater than or equal to the amount being transferred and that the recipients balance will not overflow, safe-addition and safe-subtraction are unnecessary.

Update: The recommendation to remove SafeMath from operations was followed.

WrappedEther - transferFrom function: Since the if-statement at the beginning of the function checks that the balance of the sender is greater than or equal to the amount, the recipients balance will not overflow, and the sender's allowance from the owner account is greater than or equal to the amount being spent, safe-addition and safe-subtraction are unnecessary in the body of the function.

Update: The recommendation to remove SafeMath from operations was followed.

UserContract - Initiate function: Safemath should be used to ensure that the multiplication "amount * 2" does not overflow. This error would be caught later when createSwap is called, but adding a check sooner would be more in line with a 'fail early, fail hard' mentality.

SafeMath was added to the multiply operation within the require statement.

TokenLibrary: contract_details array: Although this array is declared to have 7 elements, `contract_details[6]` is never defined. `Contract_details[6]` is used in the Calculate function to determine `self.token_amount` and currently it is always zero. If this value is supposed to be zero, this element of the array should be removed.

Update: `Contract_details[6]` is now set to `self.factory.getVariables()` making it a valid variable.

TokenLibrary: oracleQuery function: The nested if-clause `if(self.contract_details[4] == 0)` is unnecessary. This is because in order to enter the overall if-clause `if(_today >= self.contract_details[0] && self.contract_details[4] == 0)`, `self.contract_details[4] must equal 0. The only way self.contract_details[4]` is altered to not equal 0 is if it enters the if-clause within the for-loop. However, in this case the function returns true directly after altering `self.contract_details[4]` so our aforementioned nested if-cause is never reached. Therefore `self.contract_details[4]` must always equal 0 if it gets to this nested if-clause and this if-clause can be removed.

Update: The unnecessary nested if-clause was removed.

TokenLibrary: Calculate function: In the second line of code, the phrase `self.token_amount.mul(10000-self.contract_details[6]).div(10000)` should use safe-subtraction for `10000 - self.contract_details[6]`. Although this is not a problem now since `self.contract_details[6]` is never defined, it may arise later.

Update: The recommendation to use `SafeSubtraction` for `10000-self.contract_details[6]` was ignored. `SafeSubtraction`should be used in order to catch an error in the case of integer underflow.

MasterDeployer - removeFactory function: There should be checks in place to make it impossible to remove `address(0)` or an address that does not correspond to a deployed Factory from the list of deployed factories. Currently, this possibility poses an issue because it makes it difficult for outside contracts to determine which factories were deployed by this MasterDeployer.

Update: Require statements were added in order to make sure that the `0-address` can never be removed and that a factory not in the list of deployed factories cannot be removed.

MasterDeployer - deployFactory function: The function signature of `deployFactory` says that `deployFactory` returns an address, but this function does not return anything. Add a line, `return _new_fac;` to fix this issue.

Update: `_new_fac` is returned so the function signature is satisfied.

Oracle - constructor: The string API is being set to the incorrect URL. The correct URL is "**https://api.gdax.com/products/BTC-USD/ticker**".

TestOracle - constructor: The string API is being set to the incorrect URL. The correct URL is "**https://api.gdax.com/products/BTC-USD/ticker**".

4. Documents & Resources

4.1 Line-By-Line Comments

https://github.com/authio-ethereum/Audits/tree/master/DDA

4.2 Project Codebase

https://github.com/DecentralizedDerivatives/DRCT_standard

5. Conclusion

The Authio team would like to commend the DDA on their DRCT_standard project. With review of the mentioned issues, the codebase has the potential to represent a secure, robust platform derivatives on the Ethereum blockchain. We would like to recommend that the DDA continue with the process of securing their code by posting public bug bounties and soliciting community feedback.