CONSENSYS
Diligence

Lien Protocol Audit

Date	May 2020
Lead Auditor	John Mardlin
Co-auditors	Shayan Eskandari

1 Executive Summary

This report presents the results of our engagement with Lien Finance to review their smart contract system, including: MainContracts, 3 FairSwap systems, Oracle interface, and their Token implementation which is a modified version of ERC20 with maturity date.

The review was conducted over the course of two weeks, from May 25, 2020 to June 6, 2020 by John Mardlin and Shayan Eskandari. A total of 20 person-days were spent.

During the project kick-off call, and the first week of the engagement, we directed most of our attention towards the documentation and code review of the three Fairswap exchanges, and the Oracle contracts.

Towards the end of the first week, and during the second week we began to split our efforts in order to cover the large codebase more evenly, with John focusing on the exchange contracts, and Shayan focusing on the LienToken and MainContracts repos.

1.1 Scope

At the outset, our review focused on the commit hashes in the 6 repositories below.

Repository	Commit
https://github.com/LienProtocol/Fairswap_iDOLvsLien	`6bb79f9e4790ad6d5cd3a6`
https://github.com/LienProtocol/Fairswap_iDOLvsETH	`18792430a456277ea9ec14`
https://github.com/LienProtocol/Fairswap_iDOLvsImmortalOptions	`dccd3f8bf750a847f35b32`
https://github.com/LienProtocol/LienToken/	`acf89ca918b8d47eeff1c8`
https://github.com/LienProtocol/Oracle	`d8f15219a05a9f49c4ec34`
https://github.com/LienProtocol/MainContracts	`ace052936b5590b4a0e548`

These repositories were updated multiple times during the course of the review, as incomplete portions of the code were identified. A list of these updates can be found in the Appendix.

Together with the Lien team, it was established that the main focus for this review would be the high-level business logic and best practices present in the code. Lien stated that they are *"confident in mathematical components through bunch of test cases including edge cases"*

2 Recommendations

2.1 Consider an iterative approach to launching

The system has many components with complex functionality and no apparent upgrade path.

We recommend identifying which components are crucial for a minimum viable system, then focusing efforts on ensuring the security of those components first, and then moving on to the others.

2.2 Be aware of and prepare for worst-case scenarios

During the early life of the system, have a method for pausing and upgrading the system. The details of some scenarios were discussed with the client on the delivery meeting.

2.3 Testing

Document how to run tests in the README

The steps required to execute the test suite (using truffle) should be clearly documented in the `README` of each repository.

It would also be helpful to define an `npm test` script in the `package.json` file.

Fix failing tests

A significant number of tests are failing. These should be fixed or removed if they are no longer relevant.

Extend the test suite

We recommend writing unit tests for the `_calculate...()` functions, and other functions which contain complex mathematics or business logic. If functions are private, they can be exposed by writing test contracts.

In order to discover edge-cases, consider running a large number of tests by running a loop over an array with many different inputs. Here is an example.

2.4 Move the Fairswap implementations to single repository

There are three separate Fairswap repositories which share much of their logic and internal structure. We recommend moving the three implementations into a single repository, and extract any functionality which is common between them into parent contracts which can be inherited.

Instead of having duplicate code in two different parts of the system, we recommend having one implementation to prevent future mistakes while updating the code, or possibly missing bug fixes in either of the implementations.

2.5 Remove unused code

Unused code can distract reviewers and reduce the readability of the system. For example in `Fairswap_iDOLvsETH/contracts/BoxExchange.sol`, the `onlyLienToken` modifier is defined but never used.

In addition to some functions not being used, there are some input arguments that are defined but not used. It is suggested to clean up these instances.

Example: The second argument `uint256` is unnamed and not used.
`MainContracts/contracts/BondMaker.sol`

```
function liquidateBond(uint256 bondGroupID, uint256) public override {
        _distributeETH2BondTokenContract(bondGroupID);
    }
```

2.6 Review the Code Quality recommendations in Appendix 1

Other comments related to readability and best practices are listed in Appendix 1

3 Issues

Each issue has an assigned severity:

- Minor issues are subjective in nature. They are typically suggestions around best practices or readability. Code maintainers should use their own judgment as to whether to address such issues.
- Medium issues are objective in nature but are not security vulnerabilities. These should be addressed unless there is a clear reason not to.
- Major issues are security vulnerabilities that may not be directly exploitable or may require certain conditions in order to be exploited. All major issues should be addressed.
- Critical issues are directly exploitable security vulnerabilities that need to be fixed.

3.1 A reverting fallback function will lock up all payouts Critical

Description

In BoxExchange.sol, the internal function `_transferEth()` reverts if the transfer does not succeed:

code/Fairswap_iDOLvsETH/contracts/BoxExchange.sol:L958-L963

```
function _transferETH(address _recipient, uint256 _amount) private {
    (bool success, ) = _recipient.call{value: _amount}(
        abi.encodeWithSignature("")
    );
    require(success, "Transfer Failed");
}
```

The `_payment()` function processes a list of transfers to settle the transactions in an `ExchangeBox`. If any of the recipients of an Eth transfer is a smart contract that reverts, then the entire payout will fail and will be unrecoverable.

Recommendation

1. Implement a queuing mechanism to allow buyers/sellers to initiate the withdrawal on their own using a 'pull-over-push pattern.'
2. Ignore a failed transfer and leave the responsibility up to users to receive them properly.

3.2 WIP: Force traders to mint gas token Major

Description

Attack scenario:

1. Alice makes a large trade via the Fairswap_iDOLvsEth exchange. This will tie up her iDOL until the box is executed.

2. Mallory makes a small trades to buy ETH immediately afterwards, the trades are routed through an attack contract.
3. Alice needs to execute the box to get her iDOL out.
4. Because the gas amount is unlimited, when you Mallory's ETH is paid out to her attack contract, mint a lot of GasToken.

If Alice has $100 worth of ETH tied up in the exchange, you can basically ransom her for $99 of gas token or else she'll never see her funds again.

Examples

code/Fairswap_iDOLvsETH/contracts/BoxExchange.sol:L958

```
function _transferETH(address _recipient, uint256 _amount) private {
```

Recommendation

When sending ETH, a pull-payment model is generally preferable.

This would require setting up a queue, allowing users to call a function to initiate a withdrawal.

3.3 Missing Proper Access Control Major

Description

Some functions do not have proper access control and are `public`, meaning that anyone can call them. This will result in system take over depending on how critical those functionalities are.

Examples

Anyone can set `IDOLContract` in `MainContracts.Auction.sol`, which is a critical aspect of the auction contract, and it cannot be changed after it is set:

code/MainContracts/contracts/Auction.sol:L144-L148

```
 */
function setIDOLContract(address contractAddress) public {
    require(address(_IDOLContract) == address(0), "IDOL contract is already
registered");
```

```
        _setStableCoinContract(contractAddress);
    }
```

Recommendation

Make the `setIDOLContract()` function `internal` and call it from the constructor, or only allow the `deployer` to set the value.

3.4 Code is not production-ready Major Pending

Resolution
The code reported in this issue was originally included for testing purposes. It was removed and replaced by the proper code during the second week of the engagement. We will close this issue when no more testing code is present in the codebase.

Description

Similar to other discussed issues, several areas of the code suggest that the system is not production-ready. This results in narrow test scenarios that do not cover production code flow.

Examples

In `MainContracts/contracts/AuctionTimeControl.sol` the following functions are commented out and replaced with same name functions that simply return True for testing purposes:

- isNotStartedAuction
- inAcceptingBidsPeriod
- inRevealingValuationPeriod
- inReceivingBidsPeriod

code/MainContracts/contracts/AuctionTimeControl.sol:L30-L39

```
/*
// Indicates any auction has never held for a specified BondID
function isNotStartedAuction(bytes32 auctionID) public virtual override returns
(bool) {
    uint256 closingTime = _auctionClosingTime[auctionID];
    return closingTime == 0;
}

// Indicates if the auctionID is in bid acceptance status
function inAcceptingBidsPeriod(bytes32 auctionID) public virtual override returns
(bool) {
    uint256 closingTime = _auctionClosingTime[auctionID];
```

code/MainContracts/contracts/AuctionTimeControl.sol:L67-L78

```
// TEST
function isNotStartedAuction(bytes32 auctionID)
    public
    virtual
    override
    returns (bool)
{
    return true;
}

// TEST
function inAcceptingBidsPeriod(bytes32 auctionID)
```

These commented-out functions contain essential functionality for the Auction contract. For example, `inRevealingValuationPeriod` is used to allow revealing of the bid price publicly:

code/MainContracts/contracts/Auction.sol:L403-L406

```
require(
    inRevealingValuationPeriod(auctionID),
    "it is not the time to reveal the value of bids"
);
```

Recommendation

Remove the test functions and use the production code for testing. The tests must have full coverage of the production code to be considered complete.

3.5 Unable to compile contracts Major ✓ Fixed

Resolution

The related code was updated on the 7th day of the audit, and fixed this issue for `Fairswap_iDOLvsImmortalOptions` as far as we reviewed.

Description

In the `Fairswap_iDOLvsImmortalOptions` repository:

Compilation with truffle fails due to a missing file: `contracts/testTokens/TestBondMaker.sol`.
Compilation with `solc` fails due to an undefined interface function:

```
Error: Member "calculatePrice" not found or not visible after argument-dependent lookup
in contract CalculatorInterface.
   --> contracts/BoxExchange.sol:821:36:
    |
821 |         uint256[5] memory Prices = calc.calculatePrice(
    |                                    ^^^^^^^^^^^^^^^^^^^
```

In the `Fairswap_iDOLvsLien` repository:

Compilation with truffle fails due to a missing file: `./ERC20RegularlyRecord.sol`. The correct filename is `./TestERC20RegularlyRecord.sol`.

Recommendation

Ensure all contracts are easily compilable by following simple instructions in the README.

3.6 Unreachable code due to checked conditions Medium

Description

The code flow in `MainContracts.Auction.sol` `revealBid()` is that it first checks if the function has been called during the reveal period, which means "after closing" and "before the end of the reveal period."

code/MainContracts/contracts/Auction.sol:L508-L517

```
function revealBid(
    bytes32 auctionID,
    uint256 price,
    uint256 targetSBTAmount,
    uint256 random
) public override {
    require(
        inRevealingValuationPeriod(auctionID),
```

```
            "it is not the time to reveal the value of bids"
        );
```

However, later in the same function, code exists to introduce “Penalties for revealing too early.” This checks to see if the function was called before closing, which should not be possible given the previous check.

code/MainContracts/contracts/Auction.sol:L523-L537

```
/**
 * @dev Penalties for revealing too early.
 * Some participants may not follow the rule and publicate their bid information
before the reveal process.
 * In such a case, the bid price is overwritten by the bid with the strike price
(slightly unfavored price).
 */
uint256 bidPrice = price;

/**
 * @dev FOR TEST CODE RUNNING: The following if statement in L533 should be replaced
by the comment out
 */
if (inAcceptingBidsPeriod(auctionID)) {
    // if (false) {
    (, , uint256 solidStrikePriceE4, ) = _getBondFromAuctionID(auctionID);
    bidPrice = _exchangeSBT2IDOL(solidStrikePriceE4.mul(10**18));
}
```

Recommendation

Double-check the logic in these functions. If revealing should be allowed (but penalized in the earlier stage), the first check should be changed. However, based on our understanding, the first check is correct, and the second check for early reveal is redundant and should be removed.

3.7 TODO tags present in the code Medium

Description

There are a few instances of `TODO` tags in the codebase that must be addressed before production as they correspond to commented-out code that makes up essential parts of the system.

Examples

code/MainContracts/contracts/Auction.sol:L310-L311

```
// require(strikePriceIDOLAmount > 10**10, 'at least 100 iDOL is required for the bid
Amount'); // require $100 for spam protection // TODO
require(
```

code/MainContracts/contracts/BondMaker.sol:L392-L394

```
bytes32[] storage bondIDs = bondGroup.bondIDs;
// require(msg.value.mul(998).div(1000) > amount, 'fail to transfer Ether'); // TODO
```

code/MainContracts/contracts/BondMaker.sol:L402-L404

```
    _issueNewBond(bondID, msg.sender, amount);
    // transferETH(bondTokenAddress, msg.value - amount); // TODO
}
```

3.8 Documented function `getERC20TokenDividend()` does not exist Medium

Description

In the README of Fairswap_iDOLvsLien, a function is listed which is not implemented in the codebase:

> *`getERC20TokenDividend()` function withdraws ETH and baseToken dividends for the Lien token stored in the exchange.(the dividends are stored in the contract at this moment)*

Recommendation

Implement the function, or update the documentation

3.9 Fairswap interfaces are inconsistent Medium

Description

There are unexpected inconsistencies between the three Fairswap contract interfaces, which may cause issues for composability with external contracts.

Examples

The function used to submit orders between the base and settlement currency has a different name across the three exchanges:

1. In `Fairswap_iDOLvsETH` it is called: `orderEThToToken()`.
2. In `Fairswap_iDOLvsLien` it is called: `OrderBaseToSettlement()` (capitalized).
3. In `Fairswap_iDOLvsImmmortalOptions` it is called: `orderBaseToSettlement()`.

Recommendation

Implement the desired interface in a separate file, and inherit it on the exchange contracts to ensure they are implemented as intended.

3.10 Fairswap: inconsistent checks on `_executionOrder()` Minor

Description

The `_executionOrder()` function should only be called under specific conditions. However, these conditions are not always consistently defined.

Examples

code/Fairswap_iDOLvsLien/contracts/BoxExchange.sol:L218

```
if (nextBoxNumber > 1 && nextBoxNumber > nextExecuteBoxNumber) {
```

code/Fairswap_iDOLvsLien/contracts/BoxExchange.sol:L312

```
if (nextBoxNumber > 1 && nextBoxNumber > nextExecuteBoxNumber) {
```

code/Fairswap_iDOLvsLien/contracts/BoxExchange.sol:L647

```
if (nextBoxNumber > 1 && nextBoxNumber >= nextExecuteBoxNumber) {
```

Recommendation

Reduce duplicate code by defining an internal function to perform this check. A clear, descriptive name will help to clarify the intention.

3.11 Inconsistency in `DecimalSafeMath` implementations Minor

Description

There are two different implementations of `DecimalSafeMath` in the 3 FairSwap repositories.

Examples

`FairSwap_iDOLvsLien/contracts/util/DecimalSafeMath.sol#L4-L11`

```
library DecimalSafeMath {
    function decimalDiv(uint256 a, uint256 b)internal pure returns (uint256) {
        // assert(b > 0); // Solidity automatically throws when dividing by 0
        uint256 a_ = a * 1000000000000000000;
        uint256 c = a_ / b;
        // assert(a == b * c + a % b); // There is no case in which this doesn't hold
        return c;
    }
```

`Fairswap_iDOLvsETH/contracts/util/DecimalSafeMath.sol#L3-L11`

```
library DecimalSafeMath {

    function decimalDiv(uint256 a, uint256 b)internal pure returns (uint256) {
        // assert(b > 0); // Solidity automatically throws when dividing by 0

        uint256 c = (a * 1000000000000000000) / b;
        // assert(a == b * c + a % b); // There is no case in which this doesn't hold
        return c;
    }
```

Recommendation

Try removing duplicate code/libraries and using a better inheritance model to include one file in all FairSwaps.

Appendix 1 - Code Quality Recommendations

A.1.1 Use structs to reduce the number of mappings

`MainContracts.Auction.sol` has 31 mappings which use `auctionID` as the key. This can be simplified by using a small number of mappings which return large structs. This would significantly increases the code readability.

Another example is the mappings in `StableCoin.sol` which use `poolID` as the key, and can be refactored to use a `Pool` struct.

A.1.2 Reduce repetitive code

This code snippet occurs in all three FairSwaps in the `orderEthToToken` and `orderTokenToEth` functions:

e.g. `Fairswap_iDOLvsETH/contracts/BoxExchange.sol#L184-L198`

```
            nextBoxNumber += 1;
            Exchange[nextBoxNumber - 1].blockNumber = uint32(block.number);
            if (_isLimit) {
                Exchange[nextBoxNumber - 1].buyOrdersLimit[msg.sender] = msg
                    .value
                    .decimalDiv(DECIMAL + FEE_RATE);
                Exchange[nextBoxNumber - 1].buyersLimit.push(msg.sender);
                Exchange[nextBoxNumber - 1].totalBuyAmountLimit += msg.value;
            } else {
                Exchange[nextBoxNumber - 1].buyOrders[msg.sender] = msg
                    .value
                    .decimalDiv(DECIMAL + FEE_RATE);
                Exchange[nextBoxNumber - 1].buyers.push(msg.sender);
                Exchange[nextBoxNumber - 1].totalBuyAmount += msg.value;
            }
```

Each branch of the logic in these functions is nearly identical. We recommend using an internal function to handle these state changes and significantly reduce the total lines of code.

A.1.3 Make use of the contract type for additional compiler safety

In many cases known contract types (ie. an ERC20 token or an Oracle), are declared as an `address` type. This circumvents Solidity's type safety checks.

For example, in `Fairswap_iDOLvsETH/contracts/BoxExchange.sol` :

```
    // Bad
    address public tokenAddress; // this variable is unused after init, and is
redundant with the `token` variable below
```

```
    address payable public lienTokenAddress;

    IERC20 token;

    constructor(address _tokenAddress, address payable _lienTokenAddress)
        public
    {
        tokenAddress = _tokenAddress;
        token = IERC20(tokenAddress);
        lienTokenAddress = _lienTokenAddress;
    }

    // OK
    IERC20 token; // this is an
    LienToken payable public lienToken;

    constructor(IERC20 _token, LienToken payable _lienTokenAddress)
        public
    {
        token = _token
        lienToken = _lienToken;
    }
```

A.1.4 Restrict mutability on `_calculateX()` functions

In `BoxExchange.sol`, across the 3 Fairswap repos, many of the `calculateX()` functions should be labelled either `pure` or `view` to ensure that state changes are not introduced as side effects.

The same recommendation likely applies in other parts of the codebase and it is worth reviewing for.

A.1.5 Function Naming suggestions

Proper use of `_` as a function name prefix

A common pattern is to prefix `internal` and `private` function names with `_`. This pattern is applied in most of the code base, however there are inconsistencies, such as

- In the MainContracts repo:
  - UseAction.sol: `_startAuction` is a *public* function
  - Action.sol: `restartAuction` is an *internal* function
- FairSwap_iDOLvsLien
  - BoxExchange.sol: `getTokenPerShare`, `getCurrentPrice` are *internal* functions

Proper use of `get` as a function name prefix

Clear function names can increase readability. Follow a standard convertion function names such as using `get` for getter (view/pure) functions. Examples:

- `MainContracts/contracts/StableCoin.sol` **exchangeSBT2IDOL**: the name indicates the exchange will happen inside this function, however this function returns the exchange calculation. It might be helpful to use `calc` or similar keyword in such functions. Or as mentioned in another reported issue, to move all calculations to a calculator smart contract and use `calculator.functioName` to call those functions.
- `FairSwap_iDOLvsLien/contracts/BoxExchange.sol` **getDividend**, **getERC20TokenDividendLP**, and **getERC20TokenDividend**: the names suggest to be getter functions, however these function result in state changes and payouts (or receiving) of the dividends.

A.1.6 Variable naming suggestions

Some of the constants namings are inconsistent. For example the seconds in a day is defined as `A_DAY` in `TrustedPriceOracle`, and `SECONDS_IN_DAY` in `ChainlinkPriceOracle`.

The variable `nextBoxNumber` is initialized to `1` and is most often referenced as `nextBoxNumber - 1`. The code would be greatly simplified by replacing it with `currentBoxNumber` initialized to `0`.

A.1.7 Ordering of definitions

In many cases, events are defined in different places throughout a file. Follow the conventional ordering of definitions in a file:

1. Global constants
2. Variables
3. Events
4. Modifiers
5. Functions

A.1.8 Review and update misleading comments

Some inline comments in the code are misleading. This could be result of code updates without updating the associated comments.

Example: `/Fairswap_iDOLvsETH/contracts/BoxExchange.sol`

```
        //if refundrate > 0, refund baseToken
        _transferETH(orderer, refundAmount); // says baseToken but transfers Eth
    }
```

A.1.9 Use small functions to clarify the intention of recurring conditional statements

This check occurs 3 times in each Fairswap contract:

```
  Exchange[nextBoxNumber - 1].blockNumber != 0 &&
 Exchange[nextBoxNumber - 1].blockNumber + 1 >= block.number
```

Another example are the checks prior to call `_executionOrders`. This check occurs twice:

```
nextBoxNumber > 1 && nextBoxNumber > nextExecuteBoxNumber
```

... and this check which once, although it is presumably intended to be identical to the prior example.

```
nextBoxNumber > 1 && nextBoxNumber >= nextExecuteBoxNumber
```

Implementing these checks in a small function provides an opportunity to clarify the intention of the check, and will reduce the probability of making mistakes in the logic.

A.1.10 `_isAuctionEmergency` only needs mapping to boolean

`_isAuctionEmergency` mapping is used to determine if an auction should be treated in a special way (e.g. shorted reveal period, etc). It is currently defined as `bytes32 => uint`, where a boolean would suffice.

```
contract AuctionTimeControl is Time, AuctionTimeControlInterface {
    // 1: Emergency 0:Else
    mapping(bytes32 => uint256) _isAuctionEmergency;
```

A.1.11 Avoid Duplicate checks

There are many instances of duplicate and redundant checks throughout the code.

Examples:

- `Oracle/contracts/TrustedPriceOracle.sol` *isWorking()* is called in any instance that requires price from the oracle. The check `_latestId > 0` is checked once in the same function and another time in `_latestTimestamp()`.

  ```
  function isWorking() external override returns (bool) {
      return now.sub(_latestTimestamp()) < A_DAY && _latestId > 0;
  }

  function _latestTimestamp() private view returns (uint256) {
      require(_latestId > 0, "no prices found");
      return timestamps[_latestId];
  }
  ```

- `MainContracts/contracts/oracle/UseOracle.sol` `_getPriceOn(uint256 timestamp, uint256 hintID)` is only called from `_getPriceOn(uint256 timestamp)` which passes `latestId` as `hintID`:

  ```
  function _getPriceOn(uint256 timestamp)
      internal
  ```

```
    returns (uint256 rateETH2USDE8)
{
    uint256 hintID = _oracleContract.latestId();
    return _getPriceOn(timestamp, hintID);
}

 function _getPriceOn(uint256 timestamp, uint256 hintID)
    internal
    returns (uint256 rateETH2USDE8)
{
    require(
        address(_oracleContract) != address(0),
        "the oracle contract is not given"
    );

    uint256 latestID = _oracleContract.latestId(); //@audit hintID is already
latestID!
    require(
        latestID != 0,
        "system error: the ID of oracle data should not be zero"
    );

    require(hintID != 0, "the hint ID must not be zero");
    uint256 id = hintID; //@audit: does not need these checks as hintID =
latestID
    if (hintID > latestID) {
        id = latestID;
    }
```

A.1.12 `DecimalSafeMath` naming and usage are confusing

`DecimalSafeMath` the name indicates it is using safeMath, however it is not and it could possibly overflow. Other than this consideration, the use cases have `DECIMAL` in the formulas that use `decimalDiv` and `decimalMul`, which further reduces the readability of the code and calculation.

Example: `Fairswap_iDOLvsETH/contracts/BoxExchange.sol`

```
        uint256 tokensPerShare = (tokenPool.mul(DECIMAL)).decimalDiv(
            _totalShares
        );
        uint256 tokensRequired = (sharesPurchased.decimalMul(tokensPerShare))
            .div(DECIMAL);
```

This is mainly a readability and auditability suggestion. It is better to either remove the implicit DECIMALS in these functions and move them to where the formulas are implemented.

A.1.13 `payable` is used unnecessarily

In many cases an `address` is unnecessarily defined as `payable`.

Examples:

- `MainContracts/contracts/BondMaker.sol` *_issueNewBond()* minting tokens does not need payable address

```
address payable bondTokenAddress = _bondTokenAddress[bondID];
BondToken bondTokenContract = BondToken(bondTokenAddress);
bool success = bondTokenContract.mint(receiver, amount);
```

- `Fairswap_iDOLvsETH/contracts/BoxExchange.sol` *buyers* and *sellers* are defined as payable addresses in `ExchangeBox` struct, but also has been casted to `payable` address in many other code flows. such as:

```
// _executionOrders()
address payable[] storage buyers0 = Exchange[_nextExecuteBoxNumber]
    .buyers;
address payable[] storage buyers1 = Exchange[_nextExecuteBoxNumber]
    .buyersLimit;
address payable[] storage sellers0 = Exchange[_nextExecuteBoxNumber]
    .sellers;
address payable[] storage sellers1 = Exchange[_nextExecuteBoxNumber]
    .sellersLimit;

//which calls _payment on each array and has another payable tag there
function _payment(
uint256 _start,
uint256 _end,
uint256 _price,
uint256 _refundRate,
bool _isBuy,
address payable[] memory orderers,
mapping(address => uint256) storage orders
) private {
if (_end > 0) {
    for (uint256 i = _start; i < _end; i++) {
        address payable orderer = orderers[i];
```

Appendix 2 - Updated Commits

This section records changes to the codebase after the start of the engagement.

A.2.1 Updated Commits on Day 3

- MainContracts `a0a309aa8aca2c3aa58d5e22c3bada6fec66a5b2` (significant changes to solidity files) and then again:
- MainContracts `f031eaec16884900dd2033cbaa2ff69de042dad9` (adds package.json)
- FairSwap_iDOLvsImmortlOptions: `1d36cbf61df60f6daefb47c9126564d06eeb9527` (significant changes to solidity files)

A.2.2 Updated Commits on Day 3

- MainContracts `a0a309aa8aca2c3aa58d5e22c3bada6fec66a5b2` (significant changes to solidity files) and then again:
- MainContracts `f031eaec16884900dd2033cbaa2ff69de042dad9` (adds package.json)
- FairSwap_iDOLvsImmortlOptions: `1d36cbf61df60f6daefb47c9126564d06eeb9527` (significant changes to solidity files)

A.2.3 Updated Commits on Day 4

- FairSwap_iDOLvsImmortaloption: `fdbc8813400c05a20ee0c135eacc5d89fbecb3ba` (updates readme)
- FairSwap_iDOLvsLien: `92fea1cdebac4aeefe3f6cf9dd77f33e04f9de5a` (fixes tests, small changes to solidity files)

A.2.4 Updated Commits on Day 7

- MainContracts `79f9c4d2fa245ed124acd78fcf837f74818e44af` changed more than 3000 lines of code. Many of the changes were simply linting fixes, but modifications occured in many of the Solidity smart contracts as well. We did not shift to this version of the code.

Appendix 3 - Disclosure

ConsenSys Diligence (“CD”) typically receives compensation from one or more clients (the “Clients”) for performing the analysis contained in these reports (the “Reports”). The Reports may be distributed through other means, including via ConsenSys publications and other distributions.

The Reports are not an endorsement or indictment of any particular project or team, and the Reports do not guarantee the security of any particular project. This Report does not consider, and should not be interpreted as considering or having any bearing on, the potential economics of a token, token sale or any other product, service or other asset. Cryptographic tokens are emergent technologies and carry with them high levels of technical risk and uncertainty. No Report provides any warranty or representation to any Third-Party in any respect, including regarding the bugfree nature of code, the business model or proprietors of any such business model, and the legal compliance of any such business. No third party should rely on the Reports in any way, including for the purpose of making any decisions to buy or sell any token, product, service or other asset. Specifically, for the avoidance of doubt, this Report does not constitute investment advice, is not intended to be relied upon as investment advice, is not an endorsement of this project or team, and it is not a guarantee as to the absolute security of the project. CD owes no duty to any Third-Party by virtue of publishing these Reports.

PURPOSE OF REPORTS The Reports and the analysis described therein are created solely for Clients and published with their consent. The scope of our review is limited to a review of Solidity code and only the Solidity code we note as being within the scope of our review within this report. The Solidity language itself remains under development and is subject to unknown risks and flaws. The review does not extend to the compiler layer, or any other areas beyond Solidity that could present security risks. Cryptographic tokens are emergent technologies and carry with them high levels of technical risk and uncertainty.

CD makes the Reports available to parties other than the Clients (i.e., “third parties”) – on its website. CD hopes that by making these analyses publicly available, it can help the blockchain ecosystem develop technical best practices in this rapidly evolving area of innovation.

LINKS TO OTHER WEB SITES FROM THIS WEB SITE You may, through hypertext or other computer links, gain access to web sites operated by persons other than ConsenSys and CD. Such hyperlinks are provided for your reference and convenience only, and are the exclusive responsibility of such web sites’ owners. You agree that ConsenSys and CD are not responsible for the content or operation of such Web sites, and that ConsenSys and CD shall have no liability to you or any other person or entity for the use of third party Web sites. Except as described below, a hyperlink from this web Site to another web site does not imply or mean that ConsenSys and CD endorses the content on that Web site or the operator or operations of that site. You are solely responsible for determining the extent to which you may use any content at any other web sites to which you link from the Reports. ConsenSys and CD assumes no responsibility for the use of third party software on the Web Site and shall have no liability whatsoever to any person or entity for the accuracy or completeness of any outcome generated by such software.

TIMELINESS OF CONTENT The content contained in the Reports is current as of the date appearing on the Report and is subject to change without notice. Unless indicated otherwise, by ConsenSys and CD.