Opyn Contracts Audit

FEBRUARY 10, 2020 | IN SECURITY AUDITS | BY OPENZEPPELIN SECURITY

Opyn is a generalized noncustodial options protocol for Decentralized Finance, or DeFi.

The team asked us to review and audit the system. We looked at the code and now publish our results.

The audited commit is `c34598565cba2bfcf824eb2da63d95c7f5dda4fa` and the contracts included in the scope were:

- `OptionsContract.sol`,
- `OptionsUtils.sol`, and
- `OptionsFactory.sol`.

The scope did not include the files inside the `lib` folder, `Migrations.sol`, and the `OptionsExchange.sol`.

All external code and contract dependencies were assumed to work correctly. Additionally, during this audit, we assumed that the administrators are available, honest, and not compromised.

System Overview

The core of the system is in the `OptionsContract` which defines how options work, and the `OptionFactory` contract which deploys `OptionContract` as needed.

OptionsContract

The `OptionsContract` defines an ERC20 compliant token called oToken as well as the functions for options. Parameters such as the expiration time, collateral, underlying asset, etc. are set at the deployment stage by the address that creates the option.

These parameters can be changed by the admin after deployment. Once an option is deployed, users will be able to provide collateral to their repo (vault) and issue oTokens in return. The system gets the asset prices from the Compound Oracle and requires over collateralization through `collateralizationRatio` to ensure a safety margin. When the ratio of collateral / underlying assets drops to an unsafe margin, the system allows a liquidation process to help itself stay collateralized. Liquidation allows any user to burn oTokens in exchange for the same value of collateral with a bonus, reducing the oTokens in circulation. Before the option expires, anyone with oTokens can exercise their option to get the promised collateral back by providing underlying assets. After an exercise event, the Repo owners, accounts that have collateral assets in the contract, will be able to collect the proportional remaining collateral along with the underlying assets transferred during exercise.

OptionsFactory

The `OptionsFactory` contract deploys `OptionContracts` as needed. It defines a white list of assets that can be used as collateral, while ETH and ERC20 tokens can be used as strike or underlying assets. Any user can call the factory contract with the right parameters to deploy a new option contract. The owner of all contracts is the deployer of the factory, and also the admin, which has special power like updating option parameters, whitelists, etc.

Next, our audit assessment and recommendations, in order of importance.

Update: *The Opyn team applied several fixes based on our recommendations and provided us with an updated fix commit* `3adfd9afa6d463869d9e0a78cc7f316ae34eb89e`*. We address the fixes introduced under each individual issue. Please note we only reviewed specific patches to the issues we reported. The code base underwent some other changes we have not audited, and can be reviewed in depth in a future round of auditing.*

Critical Severity

[C01][Fixed] Malicious users could steal from the OptionsContract contract

The require statement in Line 249 of OptionContract contract is not using the SafeMath library.

Let's assume that in the `OptionsFactory` the assets have been already added, that `now == 500`, and that UserA calls the `createOptionsContract` function to create a new type of Option. There, UserA passes to the function the following time parameters:
`_expiry = 1000`
`_windowSize = 1001`

Using those values, a new `OptionsContract` is created, and both `windowSize` and `expiry` are simply assigned without checks.

At this moment, UserB calls `openRepo`, then `addETHCollateral` or `addERC20Collateral`, and then `issueOTokens`. So far, the only time requirement in all those functions was:

```
require(block.timestamp < expiry);
```

Because `500 < 1000`, it does not `revert`.

UserC received the issued oTokens and let's supose that the underlying asset dropped its value, so UserC wants to `exercise`.

Here it is the first *and only* time that `windowSize` is actually used. The new time requirement is:

```
require(block.timestamp ≥ expiry - windowSize);
```

Because it is not using SafeMath, `expiry - windowSize == 1000 - 1001 == (2**256 - 1) - 1` so the statement `500 ≥ (2**256 - 1)` is `false`.

For that reason, the put holder cannot exercise his right and the contract will continue until `expiry` comes, where the `repo` owner will call `claimCollateral` and because its requirement is:

`require(block.timestamp ≥ expiry)`

UserB can withdraw the collateral, taking the premium without having any obligation to insure UserC with his underlying asset.

This problem comes from having a time checkpoint with a dynamic size, as was addressed in the issue **"[L06] Confusing time frame for actions"**, and for not using the SafeMath library when it is needed. Consider using the SafeMath methods for this operation and fixed time checkpoints along with inner checks during the set up.

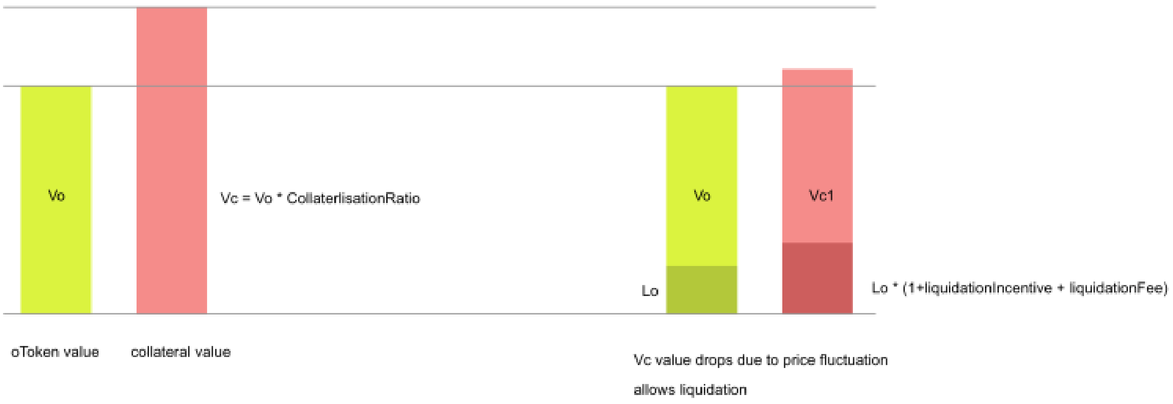
Update: *Fixed. The `require` statement now uses the SafeMath library.*

High Severity

[H01] Liquidation process could push the protocol into insolvency

Under certain asset price change conditions, the liquidation process could push the protocol into insolvency. The design of the current liquidation process incentivizes liquidators by providing them with a liquidation bonus. However, at times when the protocol is already under collateralization stress, offering a liquidation bonus plus, in Opyn's case, a protocol fee, will push further the particular Option into insolvency. Essentially these actions work against the original purpose of the liquidation function.

In the following chart we explore how the protocol's insolvency state is affected during a liquidation event, generated by the collateral to oToken price fluctuation, together with 3 variables: `collateralizationRatio`, `liquidationIncentive`, and `liquidationFee`.


Vo
Vc = Vo * CollaterlisationRatio
Vo
Vc1
Lo
Lo * (1+liquidationIncentive + liquidationFee)
oToken value
collateral value
Vc value drops due to price fluctuation
allows liquidation

On the left, we issue `Vo` value of oTokens with collateral value of `Vc`:
`Vc = Vo * CollateralizationRatio`

On the right, when total collateral value drops to `Vc1`, collateralization ratio drops, allowing someone to proceed with the liquidation. Assume that a user provides `Lo` value of oTokens for liquidation and that the `Repo` is under `liquidationFactor`. Then, the amount of collateral that will be deducted is:

`Lo * (1 + liquidationIncentive + liquidationFee)`

After this liquidation event, looking at the leftover oToken value and the leftover collateral value, if the leftover the collateral is less than the leftover the oToken value then the protocol is insolvent. Which means:

`Vc1 - Lo(1 + liquidationIncentive + liquidationFee) < Vo - Lo`

together with `Vc = Vo * collateralizationRatio` from the left part of the chart, we can get:

`Vc1 < Vc/collateralizationRatio + Lo(liquidationIncentive + liquidationFee)`

Which basically means that when the new value of collateral drops to this level, the liquidation process will push the protocol into insolvency. Plus from this moment on, and because the collateralization ratio is still low, further liquidation events are still allowed. Such events will further push the protocol deeper into insolvency.

Consider setting up an offline observation mechanism to ensure liquidation events happen as fast as they can before the price gets closer to the mentioned value.

Update: *The Opyn team is implementing a liquidator bot to help solve this issue.*

[H02][Partially Fixed] Malicious Admin can steal from the protocol

Currently an EOA (*Event Oversight Administrator*) has a lot of privileges over the control of the whole protocol, from setting up initial variables to updating critical values like Oracles and market parameters. For instance, `liquidationIncentive`, `liquidationFactor`, `liquidationFee`, `transactionFee`, and `collateralizationRatio`.

These privileges render the admin with exceptional power over general users, where it could override parameters set up in the deployment. This design puts the whole protocol in a vulnerable state if the admin account is hacked or an internal admin becomes malicious. For example, a malicious admin could easily steal from the protocol by setting a high `liquidationFee`.

Consider the use of an multi-sig account and time-locks to improve the safety of the contract against the powers of a malicious admin.

Update: *Partially Fixed in the follow-up commit `3adfd9afa6d463869d9e0a78cc7f316ae34eb89e`. The team has put some restrictions on the parameter update function which restricts admin power when assigning values. The team has also indicated they are working on a multi-sig solution to further protect the admin account.*

Medium Severity

[M01][Fixed] Potential race condition with Repo ownership transfer

Currently a `Repo` owner can transfer the ownership by calling `transferRepoOwnership`. A malicious original owner could front run this transaction with another one that puts the `Repo` in a worse collateralization status, For example, mint more oTokens and remove collateral. This could potentially harm the new owner.

Depending on how the arrangement for the ownership transfer has been done and how important this function is and the risk it presents, we recommend the team to consider solutions accordingly such as: implementing a time lock and allow the proposed new owner to accept the ownership, state the risk clearly to users in documentation or remove this function all together.

Update: *Fixed in the follow-up commit `3adfd9afa6d463869d9e0a78cc7f316ae34eb89e` where this function is removed.*

[M02][Fixed] Lack of event emissions after sensitive changes

It is beneficial for critical functions to trigger events for purposes like record keeping and filtering. However, functions like `updateParameters`, `transferFee`, and `transferCollateral` will not emit an event through inherited ERC20 functions, if the collateral is ETH.

Consider double checking all critical functions to see if those trigger events properly.

Update: *Fixed in the follow-up commit `3adfd9afa6d463869d9e0a78cc7f316ae34eb89e` where events were added to critical functions.*

Low Severity

[L01][Fixed] Not following check-effect-interaction pattern

In the `issueOTokens` function of the `OptionsContract` contract a `Repo` owner can `mint` new oTokens and send them to a third party.

Once the check confirms that the new amount of oTokens is safe, it mints them and transfers them to the destinatary.

There is an issue with the order of the operations: First the oTokens are minted, and then the put balance of the `Repo` is updated .

Although a reentrancy cannot happen in this case, to preserve a safer *check-effect-interaction* pattern, consider inverting the order of those operations.

Update: *Fixed in the follow-up commit `3adfd9afa6d463869d9e0a78cc7f316ae34eb89e` where `_mint` function is only called after `vault.oTokensIssued` has been updated.*

[L02] Different behavior between ETH and Tokens collateral

The project allows the usage of ETH or ERC20 tokens as collateral. Because ETH is a coin and not a token, it does not have a contract which keeps the account balances nor an address defined for that asset.

The project solves this by pretending that ETH is an ERC20 compliant token in the zero address.

This is a type of semantic overload over the zero address, which is used for two purposes. 1. to represent 'ETH contract address'. 2. to show if a token asset is supported or not by checking if the desired asset has changed its address in the `tokens` mapping from the default zero address.

Another problem is that if in the future the project needs to support only ERC20 token collaterals, ETH cannot be removed from the supported assets when the `deleteAsset` function is called.

Consider treating ETH as a different collateral type instead of adapting it to a ERC20 compliant token or add the necessary functionalities to keep up with the token based ones.

Update: *The Opyn team explained they always plan on supporting ETH as a collateral asset hence didn't remove it.*

[L03] Cannot update exponent after deployment

The `OptionsContract` allows the admin to update parameters such as `liquidationIncentive`, `liquidationFactor`, `liquidationFee`, `transactionFee`, and `collateralizationRatio`, by calling the `updateParameters` function.

Nevertheless, only the value of those variables can be changed. The `exponent`s of those `Number` variables cannot be changed.

Consider letting the function `updateParameters` to update also the `exponent`s used in the project.

Update: *The Opyn team explained that the exponent cap is there on purpose. They don't anticipate taking a fee lower than 0.01% so the extra precision is unnecessary*

[L04][Fixed] Miscalculated maxCollateralLiquidatable in liquidate function

In the `OptionsContract`, the `liquidation` function checks if the liquidator is not liquidating more than the `Repo`'s allowance by calculating `maxCollateralLiquidatable` with the `liquidationFactor` on the `Repo`'s. However there is a math mistake during the calculation in line 518: if the `liquidationFactor.exponent > 0`, the `maxCollateralLiquidatable` is calculated as

`maxCollateralLiquidatable.div(10 ** uint32(liquidationFactor.exponent))`

but this should be

`maxCollateralLiquidatable.mul(10 ** uint32(liquidationFactor.exponent))`

instead.

This bug is not really exploitable because `liquidationFactor` should be always ≤ 1 , which means `liquidation.exponent` should be always < 0.

Consider fixing the math issue, or simply remove it from the condition `liquidationFactor.exponent > 0` since it should never happen.

Update: *Fixed in line 630 of the follow-up commit* `3adfd9afa6d463869d9e0a78cc7f316ae34eb89e`.

[L05][Partially Fixed] Not using SafeMath

Besides the critical vulnerability found in **"[C01] Malicious users could steal with and from the OptionsContract contract"**, in `OptionsContract` there are more places where it is not used or badly used, such as:

- Line 509: `uint256 amtCollateralToPay = amtCollateral + amtIncentive`
- Line 582: `collateralizationRatio.exponent + strikePrice.exponent`

Although the first one is unlikely to cause an overflow or underflow, consider using the `SafeMath` library to eliminate any potential risks. In the last situation, because the variable type is `int32`, SafeMath cannot be used there. Consider instead adding extra checks to ensure the operation is not performing any underflow or overflow.

Update: *Partially Fixed in the follow-up commit* `3adfd9afa6d463869d9e0a78cc7f316ae34eb89e` *where first issue is fixed with SafeMath but second issue remains unfixed.*

[L06] Confusing time frame for actions

In an option there is a period in which certain actions such as creating `repo`s, adding collateral, liquidating a `repo`, or burning oTokens, have to be executed prior the expiration time (`expiry`) of the contract.

There is alaso an `exercise` action that can be done only during a specific time window defined by the `windowSize` variable (which closes at `expiry`).

Instead of subtracting the `windowSize` from `expiry` to know the start point where `exercise` can be executed, it is clearer to define 2 variables: `startsExercisability` and `endsExercisability`.

Consider changing the logic to have a start time where these actions can be executed and an end time where the contract expires.

Note: this issue is related to **"[C01] Malicious users could steal with and from the OptionsContract contract"** and any mitigation should consider both simultaneously.

Update: *The Opyn team explained the exercise window allows for the existence of both American and European Options. American Options can be exercised at any time until expiry, European Options can only be exercised on the last day or so.*

[L07][Fixed] Repo owner could lose collateral if leftover oTokens are not burnt before the option expires

Currently, `Repo` owners are allowed to freely mint oTokens by providing collateral. However, there is no way for the `Repo` owner to redeem the corresponding collateral for any unsold oTokens after the option expires. The `Repo` owners are supposed to burn all unsold oTokens before `expiry` to avoid losing the corresponding collateral.

While this design works and makes sense, it is quite risky for the `Repo` owners and it is unclear that `Repo` owners are bearing risks of being stuck with their own oTokens.

Consider adding more documentation and warnings in the code to further advice `Repo` owners, or only allow issuing oTokens when a trade occurs.

Update: *The team confirmed this is an expected behavior, comments are added in line 442 of the follow up commit to ensure users are only issuing oTokens when a trade occurs.*

[L08][Fixed] Factorize Repo ownership into modifier

In several functions of the `OptionsContract`, such as `issueOTokens`, the function is marked as public but further restricts the operation to only the `Repo.owner` inside the code.

Since this pattern appears in several functions, consider implementing a `isRepoOwner()` `modifier` to write less error-prone code instead of copying the code to check repo ownership in each function independently.

Update: *Fixed in the follow-up commit `3adfd9afa6d463869d9e0a78cc7f316ae34eb89e` where `repos` was replaced by newly introduced `vaults`. The function locates targeted vaults through `vaults[msg.sender]` and runs a check on if current `msg.sender` has a vault using `hasVault(msg.sender)`.*

[L09][Fixed] Unbalanced ETH operations

In `OptionsContract` contract, there is a `payable` fallback function that allows ETH to be sent to the contract but there is no way to withdraw.

Consider adding a `withdraw` function to the contract or do not accept ETH in the fallback function.

Update: *Fixed in the follow-up commit `3adfd9afa6d463869d9e0a78cc7f316ae34eb89e` where the payable fallback function was removed.*

[L10] Unbounded loops

The `Repo`s and the deployed `OptionContract`s are registered by pushing the `Repo` structs and the contract addresses into the `repos` and `optionsContracts` arrays respectively. When more `Repo`s and `OptionsContract`s are created, these arrays grow in length.

Because there is no function that decreases the length of those arrays once those `Repo`s and `OptionsContract`s are not needed, it is dangerous to loop through these arrays when they are too big.

In the contract `OptionsContract` the `length` of the `repos` array is used twice as a boundary for a `for` loop.

Because anyone can call the `public` `openRepo` function, someone could start creating `Repo`s until the `length` of `repos` is too big to be processed in a block. At that moment, the function could not be called anymore.

Because the only function that uses the `length` of the `repos` array is `getReposByOwner`, and no other function calls `getReposByOwner`, the issue will not freeze the contract's logic. Nevertheless, consider removing unbounded loops from the contract logic in case a child contract uses a function implementing one of the loops.

Notes

[N01] No way to check liquidation allowance

Currently in `OptionsContract` contract, there is no way to check the maximum liquidation allowance. During a liquidation situation, liquidators have to *blindly* request an oToken amount they want to liquidate when calling `liquidate`.

Although it is within liquidator's interest to liquidate as many oTokens as possible, there is no easy way to find out the maximum limit. The `liquidate` function will revert if liquidators try to liquidate too many oTokens. This design is not very user-friendly for liquidators, and it might cause liquidation events to fail multiple times due to the wrong balance being requested.

Consider adding a function to allow liquidators to get the maximum liquidation allowance for a certain `Repo`, or just allow liquidators to liquidate the maximum amount if their requests are over the maximum limit.

[N02][Fixed] Usage of SafeMath in non-uint256 variables

In the `OptionsContract` contract, the function `liquidate` uses the `SafeMath` library with non-`uint256` variables in L518 and 520 where the value is casted from `int32` to `uint32`. Consider casting to `uint256` instead.

Later, in the `isSafe` function (L592 and 595) `uint32` values are used along with the `SafeMath.mul` method. Consider casting the operation to `uint256`.

Update: *Fixed in the follow-up commit* `3adfd9afa6d463869d9e0a78cc7f316ae34eb89e` *where all variables mentioned in the issue are casted to uint256.*

[N03][Fixed] oTokens can still be burnt after the contract expires

In `OptionsContract`, the function `burnOTokens` should be limited to before the contract expires. There is no need to call this function after it expires, similar to the counterpart actions like `issueoTokens`, or add and remove collateral.

Consider adding a time check in the `burnOTokens` function.

Update: *Fixed in the follow-up commit* `3adfd9afa6d463869d9e0a78cc7f316ae34eb89e` *where a* `notExpired` *modifier is added to the function.*

[N04][Fixed] Change uint to uint256

Throughout the code base, some variables are declared as `uint`. To favor explicitness, consider changing all instances of `uint` to `uint256`. For example,, the following lines in `OptionsContract`:

- Line 200 `returns (uint)` should be `returns (uint256)`
- Line 315 `returns (uint[] memory)` should be `returns (uint256[] memory)`
- Line 316 `uint[]` should be `uint256[]`
- Line 318 `uint index = 0` should be `uint256 index = 0`
- Line 327 `uint[](count)` should be `uint256[](count)`

Update: *Fixed in the follow-up commit* `3adfd9afa6d463869d9e0a78cc7f316ae34eb89e`*, all issues above are either fixed or removed due to code update.*

[N05] External contracts addresses should be constants

The `OptionsUtils` contract provides basic functionalities regarding the available exchanges, the oracle, and how to tell if an asset is ETH.

Both the *Compound* Oracle and the *Uniswap* Factory addresses are stored in 2 public variables called `COMPOUND_ORACLE` and `UNISWAP_FACTORY` and those are assigned to the current addresses from the beginning.

During the `constructor` function those variables are overridden using the parameters provided during the deployment, making the original assignments unnecessary.

On the other hand, `OptionsContract` and `OptionsExchange` are the contracts that inherit `OptionsUtils`, however, a new `OptionsContract` will copy the values set in the current `OptionsExchange` to its `COMPOUND_ORACLE` and `UNISWAP_FACTORY` variables. Since there is no way to change these values in `OptionsExchange` after deployment plus the `optionsExchange` variable from `OptionsFactory` cannot be updated, the entire project would need to be re-deployed in order to change those variables. Instead of assigning these values twice, it is good enough to set the `COMPOUND_ORACLE` and `UNISWAP_FACTORY` addresses directly in `OptionsUtils` as `constant`.

Consider declaring the variables as `constant`.

[N06][Fixed] Default visibility

Some state variables in `OptionsContract` are using the default public visibility. Consider declaring explicitly the visibility of all the variables for better readability.

These are some of them: `liquidationIncentive`, `transactionFee`, `liquidationFactor`, `liquidationFee`, `windowSize`, `totalExercised`, `totalFee`, `totalUnderlying`, `totalCollateral`, and `collateralExp`.

Update: *Fixed in the follow-up commit* `3adfd9afa6d463869d9e0a78cc7f316ae34eb89e` *where state variable visibilities are clearly stated.*

[N07][Fixed]Empty lines

There are some empty lines inside `OptionsContract` such as L39, L98, and L421.

To favor readability, consider using a linter such as Solhint.

Update: *Fixed in the follow-up commit* `3adfd9afa6d463869d9e0a78cc7f316ae34eb89e` *where empty lines are removed*

[N08][Fixed] Lack of checks

Certain functions in this project will execute even if the given parameters are not actually updating any state variable values.

For instance,

- passing zero as the `amtToRemove` parameter for `removeCollateral` will cause the function to execute and trigger the `RemoveCollateral` event.
- Calling `transferRepoOwnership` from the `OptionsContract` with the same address as previous owner will trigger the `TransferRepoOwnership` event but the owner of the `Repo` has not changed.
- Calling `transferRepoOwnership` with zero address.

- Creating a new repo by calling `createOptionsContract` with an `expiry` in the past or `windowSize` value larger that the expiry time.
- Calling the `updateParameters` function with the same existing or wrong value for `liquidationIncentive`, `liquidationFee`, `transactionFee`, `collateralizationRatio`, and `liquidationFactor`.

These scenarios could emit useless events or compromise the functionality of the project. Consider adding parameter checks to these functions.

Update: *Fixed in the follow-up commit `3adfd9afa6d463869d9e0a78cc7f316ae34eb89e` where proper validations and checks are added.*

[N9][Partially Fixed] Misleading variable names

- In `OptionsContract`, the variable `amtCollateralToPayNum` is actually the amount of collateral to pay in ETH, not in collateral. Consider renaming it to `amtCollateralToPayInEthNum` to avoid confusion.
- In `OptionsContract`, the `isSafe` function creates a variable called with the same name as the function. Consider changing the name to `stillSafe`.
- In `OptionsContract`, the `getPrice` function gets the price of the asset in ETH, but the name where those values are saved suggest the opposite. Consider changing them to `collateralToEthPrice` and `StrikeToEthPrice`.

Update: *Partially Fixed in the follow-up commit `3adfd9afa6d463869d9e0a78cc7f316ae34eb89e` where some of the variable names have been updated.*

[N10][Fixed] Unknown use of multiple Repos per account

The `OptionsContract` allows any user to create a new `Repo` by calling the `openRepo` function.

However at the moment a single address can create as many `Repo`s as they want but without getting a explicit benefit.

Consider adding to the documentation what are the benefits of doing this, or removing the possibility of having multiple `Repo`s per account and allowing only one.

Update: *Fixed in the follow-up commit `3adfd9afa6d463869d9e0a78cc7f316ae34eb89e` where repos are replaced by vaults and only one vault is allowed per user.*

[N11][Fixed] Inverse order of operations

In the `OptionsUtils` contract, the `getExchange` function creates a pointer to the `exchange` instance for the queried `_token` by using the `getExchange` function from the Uniswap factory contract, and then it checks if the address retrieved from Uniswap's `getExchange` is zero.

Consider first checking if the address returned by `UNISWAP_FACTORY.getExchange(_token)` in L26 is zero before linking the queried exchange instance into the local variable `exchange`.

Update: *Fixed in line 29 of the follow-up commit `3adfd9afa6d463869d9e0a78cc7f316ae34eb89e`.*

[N12][Fixed] Mismatch case for payable address

The `TransferRepoOwnership` event takes the `Repo` index, the old owner, and the new one. The event is only triggered when the `transferRepoOwnership` function is called, a function that allows the old owner to change the ownership of the `Repo`. There is an issue with the way parameters are defined: the new owner is defined as `address payable`, however the old owner is only an address. Because the function updates the `owner` variable in the `repos` array, which is defined as `address payable`, consider changing the parameter definition of the event.

There is a similar issue in the `liquidate` function: The `Liquidate` event is defined using an address parameter, however when it is used in the `liquidate` function, `msg.sender` can be `payable` if the collateral is ETH. Consider updating the variable type in the event.

Update: *Fixed in the follow-up commit* `3adfd9afa6d463869d9e0a78cc7f316ae34eb89e`.

[N13][Fixed] Repo owners might not know their collateral balance

The project team mentioned that they are implementing new code into the `getRepoByIndex` function to return the collateral balance of a `Repo` owner. However, it might be confusing for a `Repo` owner to see their collateral value drops due to other users calling the `exercise` function.
This issue was pointed out to us by the project team and asked for our recommendation.

The fact is when the collateral drops, the underlying contained in the contract increases with each `exercise` call. It might be beneficial to return both the collateral and underlying balances when a `Repo` owner checks their balance. It is self-explanatory that when other users call `exercise`, the collateral balance drops while the underlying balance increases. Consider retrieving both balances to the `Repo` owner to prevent confusions about their balances.

Update: *Fixed in the follow-up commit* `3adfd9afa6d463869d9e0a78cc7f316ae34eb89e` *where the newly added* `getVault()` *function returns both the collateral and underlying balances*.

[N14][Partially Fixed] Misleading comments, variable names, and documentation typos

Since the purpose of the Ethereum Natural Specification (NatSpec) is to describe the code to end users, misleading statements should be considered a violation of the public API.

In the audited contracts, several functions have incomplete, or non-existent, docstrings which need to be completed.

In `OptionsContract`:

- s `trikePrice` should be `strikePrice`.
- There are ToDo comments along the code which shows that part of the functionalities are not ready yet.
- In L36, L40, and L43 a method is used to convert an integer to percentage (using the net percentage without adding the 100%), but in L47 and L162 it used a different method, which is not congruent to the previous one. For example, for the L40 definition `1054 == 105.4%` but for the L47 definition, `1054 == 5.4%`. Consider using only one method to calculate the percentage.
- `collatera` should be `collateral`.
- `indecies` should be `indices`.
- In the `exercise` function a docstring implies that L270 transfers oTokens to the contract, but actually it burns them.
- Functions that move oTokens from the user account, such as `addERC20Collateral` and `exercise`, do not say that the user has to allow the contract to handle their oTokens on his behalf before these functions are called.
- The docstrings of the `addETHCollateral` function do not say what the `return` value is.
- The revert message in the `exercise` function mentions the use of pTokens instead of the current oTokens.
- There are places, such as L14, L15, L20, L26, L32, L34, L36, and L40, where it should be explained with more detail what the variable/contract does.
- For the `Number` variables, such as the ones in L36, L40, L43, L47, and L51, it would be ideal add those examples to the actual `Number` structure in L20 and explicitly explain the formula that links the `value` with the `exponent`.

In `OptionsExchange`:

- The functions `sellPTokens` and `buyPTokens`, along with their `_pTokens` parameters, implies the use of pTokens instead of oTokens. If these are a different type of tokens, then it should be explained in the documentation.

In the whitepaper:

- "cryptoassets" in the section 1.1 line 2 should be "crypto assets".
- "marketplaces marketplace" in the section 3.3.1 paragraph 2 line 4-5 should be "marketplaces".

Update: *Partially Fixed in the follow-up commit* `3adfd9afa6d463869d9e0a78cc7f316ae34eb89e`. *Some of these issues are fixed in the new commit.*

[N15] Funds cannot be withdrawn under certain conditions

After the `OptionsContract` has expired, the forgotten funds in collateral or underlying will remain in the contract forever.

Consider adding an extra time checkpoint, such as `terminated`, when it would be possible to withdraw the forgotten funds in the contract after this period is reached.

[N16] Unnecessary code

There are some uncessary code through out the project, for example

- There is an unnecessary `return` operation in the `issueOTokens` function of the `OptionsContract` contract, the function definition does not specify any returned value.
- The `OptionsContract` defines the same `isEth` function that was inherited from the `OptionsUtils` contract.
- In the `OptionsFactory` it is imported the `OptionsUtils` contract to it but it is never used.

Consider removing all unnecessary code.

[N17][Fixed] Unneeded assignment

In the `OptionsContract` contract, the variable `oTokenExchangeRate` is defined and assigned but then, during the `constructor`, it is again assigned with the parameter value `_oTokenExchangeExp`.

Because the first assignment will not matter when the contract is deployed, consider removing it during the declaration of the variable.

Update: *Fixed in the follow-up commit* `3adfd9afa6d463869d9e0a78cc7f316ae34eb89e` *where the* `oTokenExchangeRate` *value is only assigned once in the Constructor.*

Conclusion

One critical and two high severity issues were found. Some changes were proposed to follow best practices and reduce potential attack surface.

Update: *The Opyn team has fixed the critical issue and implemented partial fixes and monitoring solutions for all high issues in their follow up commit.*

Security Audits

- If you are 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