ACO Protocol Audit

AUGUST 6, 2020 | IN SECURITY AUDITS | BY OPENZEPPELIN SECURITY

Auctus is a company which produces decentralized financial (DeFi) applications. ACO is a decentralized options protocol created by Auctus. Users can mint call and put options for different markets at different strike prices, and sell tokens which represent those options to other users.

In this audit, we reviewed the smart contracts within the ACO protocol. The audited commit is `36bcab024c9781d799381f8f973e99fd5b0f4d2d`, and the scope includes all contracts within the `smart-contracts/contracts/core`, `smart-contracts/contracts/interfaces`, and `smart-contracts/contracts/periphery` directories, as well as `Strings.sol` and `UniswapV2Library.sol` from the `smart-contracts/contracts/libs` directory. The contracts within the `smart-contracts/contracts/only-for-tests` directory, as well as `Address.sol`, `BokkyPooBahsDateTimeLibrary.sol`, and `SafeMath.sol` from the `smart-contracts/contracts/libs` directory, were not included.

All external code and contract dependencies were assumed to work as documented.

Here we present our findings.

Summary

Overall, we were quite pleased with the design of the protocol. Use of established proxy patterns allows for easy upgrades to the system while leveraging existing knowledge and best practices of the Ethereum community to keep the protocol secure and keep different options pools isolated. Use of the `ERC20` standard for options makes the protocol robust and ready for further integration with other DeFi protocols.

Most issues identified were of relatively low severity, or have to do with edge-cases. These can be minimized by careful vetting of which markets are listed and by warning users of potentially unexpected behaviors ahead of time.

System overview

Users of the ACO protocol can access it via the user interface provided by Auctus. Options can be minted by sending a pre-determined collateral asset, such as `ETH` or `USDC`, to the `ACOToken` contract. Option minters can then offer their options (which themselves are `ERC20` tokens) for sale to others. If options are sold, they allow the buyer the choice to exercise the option any time before the designated expiration date of the option token. Upon exercise, the exercising account burns their option tokens and purchases the collateral asset locked by the option minter at the designated strike price. The proceeds of this sale are transferred to the original option minter. Since the options are `ERC20` tokens and are completely fungible, the exerciser may not receive collateral from the user that they purchased options from. Exercisers have the option to receive collateral from either a queue of option minters, or from a list of chosen accounts.

If options are not exercised before the expiration date, the option token becomes worthless, and can no longer be exercised. Option minters can then redeem their accounts, receiving back their locked collateral, and keeping any profits from the sale of their option tokens.

All available option markets correspond to their own `ERC20` token, with an unchangeable strike price and expiration date. New options are created by the `FactoryAdmin` role, which is currently controlled by Auctus.

Privileged roles

There are two privileged roles which have somewhat limited abilities. Users will need to trust these roles and should only utilize the protocol if they do.

- The owner of the `factoryAdmin` address of the `ACOFactory` contract can create new `ACOToken` contract instances, change the fee charged by the ACO protocol, and change the underlying code behind the `ACOToken`s (but only before they are deployed).

- The owner of the `admin` address of the `ACOProxy` contract can change the code used as `ACOFactory`. This means that it can change nearly everything about how `ACOToken`s are deployed and the code which defines them, and that it can override any powers the `factoryAdmin` has.

Currently, both roles are managed by Auctus.

Ecosystem dependencies

The `ACOFlashExercise` contract depends on UniswapV2 and Wrapped Ether (`WETH`) to conduct a “flash exercise” of `ACOToken`s. This allows users to exercise their `ACOToken`s without having the needed funds initially, by utilizing Flash Swaps from Uniswap. Notably, this functionality is built on top of the `ACOToken` code, so any checks that apply to `ACOToken`s to retain solvency are also done here.

Since `ACOToken`s cannot be exercised after they have expired, there is a time dependency for users who purchase them. During times of high ETH congestion, it may be challenging to have an “exercise” transaction mined in time. Users should plan to exercise long enough before the expiry time that their transactions are mined, or risk losing the ability to exercise.

Critical severity

None. 🙂

High severity

[H01] Users can exercise small amounts without sending collateral

When exercising option tokens, the `_exercise` function calls `_validateAndBurn` to validate that the exercising account is able to transfer in the required amount of the `exerciseAsset`. This is calculated by `getExerciseData(tokenAmount)`, which uses the amount of option token units transferred in as `tokenAmount`. In the case of a `CALL` option, `getExerciseData` will call `_getTokenStrikePriceRelation`. If `tokenAmount` is small enough, and `strikePrice` is smaller than `underlyingPrecision`, it is possible for `_getTokenStrikePriceRelation` to return `0`. This results in the `expectedAmount` (which the exerciser is required to transfer in) being `0` as well. So, the user can choose a `tokenAmount` which is nonzero, that allows them to transfer in no tokens to conduct an exercise.

After `_validateAndBurn`, the `_exerciseOwners` function is called. Within this function, the `_exerciseAccount` function calls `getExerciseData` to determine the amount to be transferred to the accounts being exercised. In the case of a `CALL` option, it will again utilize `_getTokenStrikePriceRelation(tokenAmount)`, where `tokenAmount` is the amount of collateral tokens being exercised. This results in `_exerciseAccount` also transferring `0` units of the asset to the account being exercised. This should be the option minter's reward for selling their collateral, but we see that the minter receives nothing and thus is effectively robbed.

Finally, the `_exercise` function calls `getCollateralOnExercise`, which will simply return `tokenAmount` with a fee taken out for the `CALL` case. The following call to `_transferCollateral` will then transfer this amount back to `msg.sender`.

As can be seen from the above situation, it is possible for a user to exercise another "for free". In the case of a `CALL` option for `ETH/USDC`, with the strike price being `200 USDC`, any amount under `5e9` option token units will result in the exerciser having to pay nothing. This, notably, can be extended to users receiving more collateral than deserved by always exercising an amount of token units that is one less than a number evenly divisible by `5e9`. Thus, although redeeming `4.99...e9` tokens is disinentivized by gas costs, there is little disincentive to modify a larger token amount when exercising if the exercise was already planned anyway.

Here we present two potential solutions. First, consider modifiying the `_getTokenStrikePriceRelation` function such that instead of rounding down by default, it rounds up upon any division which results in a fraction. However, this may cause problems when exercising multiple accounts, as each account's received amount within `_exerciseAccount` is determined via a call to `getExerciseData`, which may also round. So, the amount transferred in by the user to exercise may be calculated as `x+1` units due to rounding, but when exercising `n` accounts, the total amount transferred to the accounts may be as high as `x+n` due to rounding on *each separate account's transfer*. This means that the transfers within `_exerciseAccount` may attempt to transfer more than the exercising account originally paid within `_validateAndBurn`. To solve this, the user could be required to transfer in the required amount of units plus `n_max`, where `n_max` is the highest possible number of accounts that could be exercised. Any extra funds can be returned to the user at the end of the `_exercise` function. Alternatively, it can be left in the contract if it is deemed to not be worth as much as the potential gas savings from avoiding the transfer.

The second potential solution is to refactor all calculation functions to take the less precise asset of any asset pair, and determine option token amounts based on this. Then, the corresponding amount of the higher precision asset can be derived via multiplication. This way, fractional units will never occur, as all mathematical operations will be based on multiplication. However it may require more complex logic to handle different combinations of `CALL` vs. `PUT` and underlying assets having more decimals vs. less decimals than the stike assets.

Consider implementing one of the described solutions. In general, any rounding errors should favor protocol solvency, followed by the users minting tokens, since they are taking a risk. If there is any loss of units of assets, it should be done in such a way that the protocol never has, and option minters never receive, less assets than expected. As always, when making large changes to the codebase, thorough passing test coverage should be achieved before moving to deployment.

Update: *Fixed in pull request #12. They are implementing the first solution presented. We want to highlight the fact that the exercising account doesn't receive back the amount of extra tokens left after the exercise. While this can be acceptable if the gas* *cost to send them back is higher than the value they represent, it can start to be a problem if a single token unit is highly valuable.*

Medium severity

[M01] `ERC20` transfers can misbehave

The `_transferFromERC20` function is used throughout `ACOToken.sol` to handle transferring funds into the contract from a user. It is called within `mint`, within `mintTo`, and within `_validateAndBurn`. In each case, the destination is the `ACOToken` contract.

Such transfers may behave unexpectedly if the token contract charges fees. As an example, the popular USDT token does not presently charge any fees upon transfer, but it has the potential to do so. In this case the amount received would be less than the amount sent. Such tokens have the potential to lead to protocol insolvency when they are used to mint new `ACOToken`s.

Transfers may also behave unexpectedly when they don't `throw` upon an execution's failure. Remember that the `ERC20` standard allows for such tokens to still be considered `ERC20` compliant. In this case, the `require` on line 1073 or line 1061 may not cause a `revert`, since `success` will be true.

In the case of `_transferERC20`, similar issues can occur, and could cause users to receive less than expected when collateral is transferred or when exercise assets are transferred.

Consider thoroughly vetting each token used within an ACO options pair, ensuring that failing `transferFrom` and `transfer` calls will cause reverts within `ACOToken.sol`. Additionally, consider implementing some sort of sanity check which enforces that the balance of the `ACOToken` contract increases by the desired amount when calling `_transferFromERC20`. See related issue **Warning about listing tokens**.

Update: *Ackowledged. Auctus' statement for this issue:*

> The team is aware and will always check if new token pairs are supported, paying attention to the points mentioned in [L07]

[M02] It is possible to mint `0` tokens

The various minting functions (`mintPayable`, `mintToPayable`, `mint`, and `mintTo`) all call `_mintToken`. This function calculates the number of tokens minted via `getTokenAmount` based on the `collateralAmount` sent in by the user. In the case of a `PUT` option, where `underlyingPrecision < strikePrice`, it is possible for `getTokenAmount` to return `0`, even with a non-zero `collateralAmount`. This will mint `0` tokens to the user or targeted address. Notably, if an account has `0` tokens, the check on line 568 will pass and the account will be added to the `_collateralOwners` array. This can be repeated, causing the same account to be added to the array. This will cause execution of `_exerciseOwners` to require more gas, since any calls to `_exerciseAccount` on an account with `0` balance will do nothing when the `available` amount for that account is `0`. If abused, it could lead to a freezing of this function. However, it should also be noted that `_exerciseAccounts` exists and could be used to work around this. Consider adding a check after line 573 that requires `tokenAmount != 0`. This will help keep the `_collateralOwners` array clean and minimize user error.

Update: *Fixed in pull request #6.*

[M03] Collateral owners can skip being exercised

When a user calls `exercise()` or `exerciseFrom()`, an eventual call to `exerciseOwners()` will be made. Within `exerciseOwners()`, a loop will call `_exerciseAccount(_collateralOwners[i], tokenAmount, exerciseAccount)` with decreasing `i` after every iteration. Within `_exerciseAccount`, the collateral owner in question (`_collateralOwners[i]`) will only be exercised if `_getAssignableAmount` for that account evaluates to `> 0`. If the collateral owner in question has an account balance greater than or equal to their `tokenData[account].amount` value, `_getAssignableAmount` will return `0`. This will cause the account to not be exercised upon.

A user, or coalition of users, may do this by having two accounts which own collateral, with one near index `0` of the `_collateralOwners` array and one near the final index of the same array. If the user or users suspect that the account closer to the final index of the array may be exercised, they can transfer tokens from the account closer to index `0` such that the balance of the account further from index `0` equals the `tokenData[account].amount` value for that account. This can be done by front-running an `exercise` transaction, or proactively before an `exercise` transaction takes place.

It is also important to note that this could cause the `_exerciseOwners` function to always run out of gas due to the loop, in the case of too many users having no exercisable collateral. This would effectively disable the `exercise` and `exerciseFrom` functions.

Consider adding some mechanism which allows users to be exercised once they have transferred minted tokens out of their account, even if they transfer them back.
This could be accomplished with a new data field in `tokenData`, perhaps called something like `transferredTokens`. With any functionality changes to the code, thorough test coverage is a must.

Update: *Fixed. The Auctus team implemented a different solution in pull request #25 which mitigates the possibility of disabled `exercise` and `exerciseFrom` functions. Auctus' statement for this issue:*

> We are proposing an alternative solution, because the proposed solution would make it impossible to avoid being exercised even if the option is bought back. In traditional markets, most traders buy it back to avoid being exercised. Our proposed solution is adding a salt argument that will put randomness on the start index to the array of accounts to exercise.

Low severity

[L01] Using `.call` for `init` is not flexible

When deploying another `ACOToken` using `ACOFactory.sol`, after a minimal proxy is deployed, the low-level `call` function is used to call `init` for the proxy. The `_acoTokenInitializeError` function is used to return any error message from the proxy.

While such a method of calling `init` makes sense when `ACOToken.sol`'s `init` function may be changed, we noticed the presence of the `_getAcoTokenInitData` function, which is used to format data for passing into the `init` function in the `createAcoToken` function. This is the only method which calls `_deployAcoToken`, and is therefore the only way to deploy a new `ACOToken` proxy instance.

If any future changes are made to the parameters passed into the `init` function of `ACOToken.sol`, a modified version of `ACOFactory.sol` will have to be deployed, as the `_getAcoTokenInitData` function will no longer be valid. So, the use of the low-level `call` function is unfortunately not any more useful than a direct call to the `init` function, since the flexibility of passing arbitrary data is lost. However, by implementing extra code to format the `init` input data and handle any errors, a larger surface for error is created.

Consider removing `_getAcoTokenInitData` and `_acoTokenInitializeError`, and replacing the low-level `call` with a direct call to the proxy's `init` function. This will require importing `IACOToken.sol` and adding the `init` function to the interface. All parameters for `init` can then be passed directly to `_deployAcoToken`. By doing so, the code's readability and auditability will be greatly increased, and the surface for error will be made significantly smaller.

Update: *Fixed in pull request #13.*

[L02] `ACOToken.sol` reference implementation is publicly initializable

The `init` function of `ACOToken.sol` is `public` and allows anyone to call it, setting key parameters for the contract. `ACOToken` contracts are deployed to new addresses via the minimal proxy pattern, so a reference implementation must be deployed beforehand, exposing the `init` function. A malicious user can access the `init` function and set the contract parameters as if it was a reputable `ACOToken` from Auctus. Even though this will not affect any other token contract, it should never be treated as a valid token, given that it may appear to be so to an uninformed user.

Consider initializing the `ACOToken.sol` reference implementation with "garbage" data, such as invalid asset addresses and an expiry time that has already passed, to make this clearer to users and prevent other calls to `init`. Additionally, consider establishing a developer process for doing this every time a new version of `ACOToken.sol` is deployed to the network. Consider informing users to only access `ACOToken`s through the official user interface.

Update: *Acknowledged. Auctus' statement for this issue:*

> The team will establish a process to initialize the ACOToken.sol reference implementation with "garbage" data after every new deployment.

[L03] Invalid data can be returned after expiration

Comments on lines 252, 263, 283, and 298 of `ACOToken.sol` state that the functions `unassignableCollateral`, `assignableCollateral`, `unassignableTokens` and `assignableTokens` are only valid when the `ACOToken` is not "expired". These functions are all `view` and are not called by other functions within the `ACOToken` contract. There also exists a `notExpired()` modifier, which is applied to many other functions within `ACOToken.sol` and causes reverts after the `ACOToken` is expired. Additionally, `unassignableCollateral` calls `unassignableTokens` and `assignableCollateral` calls `assignableTokens`.

Consider applying the `notExpired()` modifier to `unassignableTokens` and `assignableTokens`. This will cause calls to `unassignableCollateral`, `assignableCollateral`, `unassignableTokens` and `assignableTokens` to revert once the `ACOToken` is expired. This will not only improve the code's readability and auditability, but it will also protect future developers, both internal and external, who may have missed the comments mentioned above. When being called after expiration, these functions will revert rather than return erroneous data. Alternatively, consider implementing code such that the correct values are returned both before and after expiration.

Update: *Fixed in pull request #7. Correct values are returned now, depending on the expiration status.*

[L04] Allowance requiring functions can be front-run

There are many functions within `ACOToken.sol` which require an allowance of one user to another. For example, `burnFrom`, `redeemFrom`, and `exerciseFrom` allow one account `A` to perform actions on behalf of account `B` when account `B` has given `A` an allowance of `ACOToken`s.

If `B` gives `A` an allowance of `ACOToken`s, but front-runs `A`'s calls to these functions to change the allowance to `0`, it will cause a `revert`. For example, in `exerciseFrom`, front-running by `B` can cause `A`'s call to fail within `_burnFrom`.

The implicit mitigation for issues is that a trust relationship between `A` and `B` should exist for an allowance to be given in the first place. However, in certain cases, such as other DeFi protocols building off of ACO, or users selling allowances for their `ACOToken`s, front-running can be abused to cause losses or prevent functionality of external protocols.

We report this issue to better inform external developers and users. Both should be aware that allowances can be changed arbitrarily. Consider informing users of the risks of using functions which rely on allowances.

Update: *Acknowledged. Auctus's statement for this issue:*

> We will inform developers and external users using our channels and when we publish a development documentation.

[L05] `ACOToken`s become non-transferrable after expiration

Although the `ACOToken` contract inherits `ERC20.sol`, which has a standard `transfer` function, the `_transfer` function of `ERC20.sol` is overridden by the `_transfer` function in `ACOToken.sol`. Although it does call `_transferAction` just like the `ERC20.sol` version, one notable difference is that the version in `ACOToken.sol` has the `notExpired()` modifier on it, which means that calls to `transfer` will revert after the `ACOToken` expires.

Users should be aware that not only will their tokens be worthless after expiry, but any `transfer` attempts will fail. So, if users place their tokens in liquidity pools, such as UniswapV1, or within custodial exchanges, any attempts to transfer out their tokens will not succeed after expiration. This is especially noteworthy in the Uniswap case, as removing liquidity will always fail if one of the assets is an expired `ACOToken`, meaning that all other liquidity in the pair will also be locked.

Consider informing users of the risks of trading and transferring their tokens after expiration, especially in the case of using Uniswap.

Update: *Fixed in pull request #14 and pull request #22. Tokens are now transferrable and approvable after expiration.*

[L06] Use of `/` operator can cause unexplained reverts

In the `UniswapV2Library` library the `/` operator is used. If a division by `0` occurs, the transaction would revert with no explanation.

The `/` operator used in line 57 of `UniswapV2Library.sol` would revert with no error message if a division by `0` occurs.

Note that the `/` operator used in lines 38 and 48 of the same file would revert if the divisor is `0` before it is able to reach the actual division expression. This is due to the checks in line 37 and line 44.

Consider implementing a check before calling `getAmountIn` to ensure that `reserveOut != amountOut`, preventing a division by 0. Alternatively, consider using `div` from the `SafeMath` library, which checks if the denominator is 0 and reverts with an error message.

Update: *Fixed in pull request #23. The Auctus team acknowledged the problem and decided to remove the library and to use directly the `IUniswapV2Router02` interface. Notably, the Uniswap implementation of such interface is still using the same library presenting the highlighted issue.*

[L07] Warning about listing tokens

When taking into consideration the list of supported token pairs, special attention should be paid to:

- Tokens which do not revert on failing `transfer`s or `transferFrom`s.
- Tokens that extract fees (e.g. `USDT`) which may transfer less tokens than intended.
- Tokens with whitelists (`ACOToken` proxies may not be allowed to transfer in those cases).
- Tokens with `ERC777`-compliant hooks that can execute code.

The first two points are addressed in the issue **`ERC20` transfers can misbehave**.

Update: *Acknowledged. Auctus' statement for this issue:*

> The team is aware and will always check if new token pairs are supported, paying attention to the points mentioned.

[L08] Lack of indexed parameters

None of the parameters in the `SetFactoryAdmin`, `SetAcoTokenImplementation`, `SetAcoFee` and `SetAcoFeeDestination` events in the `ACOFactory` are indexed. The same is true for the `ProxyAdminUpdated` and `SetImplementation` events of the `ACOProxy` contract.

Indexing parameters in these events allows the timeline of sensitive changes to be more easily tracked. Consider indexing event parameters to avoid hindering the task of off-chain services searching and filtering for specific events.

Update: *Partially fixed. The Auctus team decided to index event parameters in the `ACOFactory` contract in pull request #11, but not in `ACOProxy` as shown in pull request #15. Auctus' statement for this issue:*

> We indexed parameters in the ACOFactory. About the proxy parameters, we don't expect significant amount of data to justify changing our current proxy address.

[L09] An infinite loop may exist

Within `ACOToken.sol` there is an infinite loop on line 665. The continuation condition for this loop is that `i >= 0`. Since `i` is a `uint` type and since it is decreased at each iteration, it will end up passing from `0` and then start again from the maximum positive value of an `uint` variable. So the continuation condition will always be true no matter what the value for `i` is.

Considering the check following the loop (which checks that `tokenAmount == 0`) and the `break` condition for the loop which checks the same thing, it appears that the intention was for this loop to be exitable without using the `break` condition.

Consider changing the definition of `i` to `int` instead of `uint`.

Update: *Fixed. They refactored the mentioned loop in different pull requests. Now it iterates over a maximum number of accounts that can be exercised for each call, shown in pull request #9.*

[L10] `underlyingPrecision` can overflow

`underlyingPrecision` is defined within the `ACOToken.init` function as `10 ** uint256(underlyingDecimals)`. When `underlyingDecimals` is set, there are no checks afterwards to ensure that `underlyingDecimals < 78`. The maximum possible value of `uint256` is roughly `1e77`, so any value above `77` for `underlyingDecimals` will cause `underlyingPrecision` to overflow. Consider implementing a check on `underlyingDecimals` after setting it.

Update: *Fixed in pull request #10.*

Notes & Additional Information

[N01] Superfluous code in `_getFormattedStrikePrice`

On line 987 of `ACOToken.sol`, the code can be changed to `representativeAt = 0`, since `digits` is initialized to `0` and cannot be otherwise changed until after `representativeAt` has been changed. Additionally, since this change can only happen when `representativeAt == -1`, it follows that `representativeAt` can only be either `-1` or `0`. Because of this, the condition on line 1014 must always be true, since `i` is always `0` or greater. So, this `if` branch will always execute and lines 1014 and 1016 can be removed. Consider making the changes indicated above. Doing so will make this portion of the code clearer for future developers.

Update: *Fixed in pull request #16.*

[N02] Superfluous check within `_validateAndBurn`

The check on line 738 of `ACOToken.sol` is superfluous. It checks that `balanceOf(account) > tokenData[account].amount`. However, in the next line, `sub` is called with `balanceOf(account)` and `tokenData[account].amount` as its parameters. A check within `sub` requires that `tokenData[account].amount <= balanceOf(account)`. While this does not cover the case of `tokenData[account].amount == balanceOf(account)` (which should revert with the current code), calling `sub` in this case would result in a value of `0`. Since `tokenAmount` is required to be `> 0` and the result of `sub` is required to be `>= tokenAmount`, there is no way for the case of `balanceOf(account) == tokenData[account].amount` to not cause a revert, even with the check on line 738 removed.

Consider removing the check on line 738.

Update: *Fixed in pull request #18.*

[N03] Check for updates to copied code

Many contracts, such as `Strings`, `BokkyPooBahsDateTimeLibrary`, and `UniswapV2Library` come from other sources. While these "imported" libraries can be useful to leverage the development efforts of other teams, a process should exist to check these contracts for updates.

Consider establishing a cadence for regularly checking these contracts for updates. By doing so, any functionality or security improvements can be quickly integrated into the ACO repository.

Update: *Acknowledged. Auctus' statement for this issue:*

> We will establish a process to regularly check updates

[N04] Lack of comments in `_getFormattedStrikePrice`

In the `_getFormattedStrikePrice` function of the `ACOToken` contract, there is not enough information in the docstrings about how the function should behave and what is doing.

Given the intrinsic complexity of the function, consider giving a short explanation of how the function works, either through NatSpec comments or inline comments, so that readers can have a clearer understanding of the code.

Update: *Fixed in pull request #17.*

[N05] Inaccurate docstrings

There are some inaccuracies in the docstrings describing the functions in the codebase.

- Line 168 of the `ACOToken` contract should say "first `require`" instead of "`assert`".
- Line 610 of the `ACOToken` contract should say "transferred" instead of "redeemed".

Update: *Fixed in pull request #19.*

[N06] Inaccurate error message

The error message on line 459 of `ACOToken.sol` indicates that there is no allowance for `msg.sender` from `account`. A more accurate error message would be `ACOToken::redeemFrom: Allowance too low`.

Error messages are intended to notify users about failing conditions, and should provide enough information so that the appropriate corrections needed to interact with the system can be applied. Uninformative error messages greatly damage the overall user experience, thus lowering the system's quality. Therefore, consider not only fixing the specific issue mentioned, but also reviewing the entire codebase to make sure every error message is informative and user-friendly enough.

Update: *Fixed in pull request #20.*

[N07] Typos in code

In the code, the following typos were found:

- On line 11 of ACOToken.sol, the comment should say "compliant" instead of "compliance".
- On line 156 of ACOToken.sol, the comment should say "prevent" instead of "prevents".
- On line 965 of ACOToken.sol, the function should say "Characters" instead of "Caracters". This change should also be made in the function calls on lines 903, 907, and 908.
- On line 147 of ACOFlashExercise.sol, the comment should say "to be called" instead of "to called".

Update: *Fixed in pull request #21.*

[N08] `IUniswapV2Router01.sol` is unused

The `IUniswapV2Router01` interface is not used within the contracts and can be safely removed.

Consider removing the file from the repository. If it is intended to be used, consider implementing it or documenting the reason for keeping it.

Update: *Fixed in pull request #23. The file has been renamed to `IUniswapV2Router02.sol` and it has been modified to add a new interface that now is being used.*

[N09] Expiration definition mismatches the actual behaviour

The `expiryTime` passed as input parameter in the `ACOToken` `init` function is then used in the `_notExpired` internal function. There, `now == expiryTime` is accepted as not expired, so the actual definition of the `expiryTime` should be "the last not-expired time".

Consider adding the proper comments in the docstrings and in the documentation or providing a better name for such variable in order to clarify its purpose and make easier the task of understanding the code properly.

Update: *Fixed in pull request #24. `_notExpired` now returns `false` when `now == expiryTime`.*

Conclusion

No critical and one high severity issue was found. Some changes were proposed to improve the usability of the code and the overall user experience.

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

OpenZeppelin

Products

Contracts

Defender

Security

Security Audits

Learn

Docs

Forum

Ethernaut

Company

Website

About

Jobs

Logo Kit

t
S