OpenZeppelin

Opyn Gamma Protocol Audit

OPENZEPPELIN SECURITY | NOVEMBER 10, 2020

Security Audits

The Opyn team asked us to review and audit version 1.0.0 of their new Gamma Protocol. We looked at the code and now publish our results.

The audited commit is `d151621b33134789b29dc78eb89dad2b557b25b9`, and the following files were in scope:

- `./AddressBook.sol`
- `./Controller.sol`
- `./MarginCalculator.sol`
- `./MarginPool.sol`
- `./Oracle.sol`
- `./Otoken.sol`
- `./OtokenFactory.sol`
- `./OtokenSpawner.sol`
- `./Whitelist.sol`
- `./external/proxies/PayableProxyController.sol`
- `./interfaces/AddressBookInterface.sol`
- `./interfaces/AggregatorInterface.sol`
- `./interfaces/CalleeInterface.sol`
- `./interfaces/CTokenInterface.sol`
- `./interfaces/ERC20Interface.sol`
- `./interfaces/MarginCalculatorInterface.sol`

- `./interfaces/OtokenInterface.sol`
- `./interfaces/WhitelistInterface.sol`
- `./libs/Actions.sol`
- `./libs/FixedPointInt256.sol`
- `./libs/MarginVault.sol`
- `./libs/SignedConverter.sol`
- `./packages/Spawn.sol`
- `./pricers/CompoundPricer.sol`
- `./pricers/ChainlinkPricer.sol`
- `./pricers/USDCPricer.sol`

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

Privileged roles

There are a number of privileged roles that exercise wide-ranging powers over the protocol. Users need to fully trust these roles to always behave honestly. All funds deposited into the protocol could be at risk if any of these accounts are compromised.

The owner of the `AddressBook` contract can:

- Update the address of the protocol's contract, such as the oToken implementation, the `OtokenFactory`, the `Whitelist`, the `Controller`, the `MarginPool`, the `MarginCalculator`, and the `Oracle`.
- Define the address of any other arbitrary key.
- Update the implementation of a certain contract.

The owner of the `Controller` contract can:

- Define who the full and partial pauser are.
- Restrict the usage of certain addresses to perform an arbitrary call.
- Manually refresh the configuration.

Pausers on the `Controller` contract can:

- Partially or fully pause the system at any stage, restricting most of the possible actions.

The owner of the `MarginPool` contract can:

- Define who the farmer is.

The farmer of the `MarginPool` contract can:

- Collect any excess of ERC20 compliant assets on top of the internal balance.

The owner of the `Oracle` contract can:

- Define the assets' pricer.
- Define the assets' locking and dispute periods.
- Define who the disputer is.

The disputer of the `Oracle` contract can:

- Dispute and override a pricer's price.
- Set an expiry price before the pricer has submitted their price.

The owner of the `Whitelist` contract can:

- Whitelist or blacklist: products, collaterals, oTokens, and `Callee` contracts.

Ecosystem dependencies

As the ecosystem becomes more interconnected, understanding the external dependencies and assumptions have become an increasingly crucial component of system security. To this end, we would like to discuss how the Gamma Protocol depends on the surrounding ecosystem.

- The protocol uses time-based logic to determine when options expire, which means it is dependent on the availability of the Ethereum network. If users are temporarily unable to submit transactions before the expiration of the options (for instance, during a period of high

Compound’s or Chainlink’s oracle as a trusted source of on-chain asset prices.

Update – Follow up new features audit

After the initial audit of the Gamma Protocol, the Opyn team asked us to audit a few extra features added to the code base. We further looked into these changes and the results are added to this report.

The exact scope of this new features audit is outlined in these PRs
https://github.com/opynfinance/GammaProtocol/pull/264
https://github.com/opynfinance/GammaProtocol/commit/2507ae989f292b9878e73a5679f02060a1858f09
https://github.com/opynfinance/GammaProtocol/pull/269
https://github.com/opynfinance/GammaProtocol/pull/274
https://github.com/opynfinance/GammaProtocol/pull/282
https://github.com/opynfinance/GammaProtocol/pull/266
https://github.com/opynfinance/GammaProtocol/pull/306
https://github.com/opynfinance/GammaProtocol/pull/309
https://github.com/opynfinance/GammaProtocol/pull/307
https://github.com/opynfinance/GammaProtocol/pull/311
https://github.com/opynfinance/GammaProtocol/pull/314/files
https://github.com/opynfinance/GammaProtocol/pull/318/files
https://github.com/opynfinance/GammaProtocol/pull/279/files#diff-2e744714ccb24c7cc1ca07cae31558edc260f000765875b4e28fc3545beab41e
https://github.com/opynfinance/GammaProtocol/pull/328/files#diff-d1f71ac07d568b461161e32a3acbe378e4cb1e1ccfc3772aee08369e04692fb9
https://github.com/opynfinance/GammaProtocol/pull/355

Critical severity

[C01][Fixed] An attacker can steal all the collateral assets

The `Controller` contract allows users to interact with the majority of the platform, being able to open a vault, deposit collateral assets, minting oTokens, or redeem their oTokens. All of these

Once one oToken expires, if an oToken holder wants to exercise their right and collect the payout for their oTokens, the user must send a `Redeem` action type which would end up in a call to the `_redeem` internal function.

However, in this action a malicious user can steal all the collateral assets following the attack vector below:

1.The attacker deploys a malicious oToken contract which has the same assets and structure of an existing legit oToken, especially an oToken with the most amount of collateral asset.

2.The attacker then calls the `operate` function and passes on the argument as follows:

```
[{
actionType : Redeem,
owner : attacker address,
secondAddress : attacker address,
asset : malicious oToken,
vaultId : anything,
amount : the max amount of oToken redeemable,
index: anything,
data : anything
}]
```

The `Controller` contract will first call the `_runActions` function with these arguments, `vaultUpdated` will not be set to true because `actionType == Actions.ActionType.Redeem` and will skip a conditional statement, and then the internal function will be called for the redeem action with these arguments:

```
_redeem(attacker address, malicious oToken address, amount)
```

3.The `_redeem` function does not check if given oToken is a real one or not, it only checks if:

- `now > otoken.expiryTimstamp`

redeem process.
The `Controller` contract will then burn the attacker's malicious oTokens and pay out the collateral asset from the pool, resulting in the attacker stealing all the collateral assets from the platform.

Consider validating if the oToken passed in the arguments correspond to a real oToken from the platform.

Update: *Fixed in PR#301.*

High severity

[H01][Fixed] ETH could get trapped in the protocol

The `Controller` contract allows users to send arbitrary actions such as possible flash loans through the `_call` internal function.

Among other features, it allows sending ETH with the action to then perform a call to a `CalleeInterface` type of contract.

To do so, it saves the original `msg.value` sent with the `operate` function call in the `ethLeft` variable and it updates the remaining ETH left after each one of those calls to revert in case that it is not enough.

Nevertheless, if the user sends more than the necessary ETH for the batch of actions, the remaining ETH (stored in the `ethLeft` variable after the last iteration) will not be returned to the user and will be locked in the contract due to the lack of a `withdrawEth` function.

Consider either returning all the remaining ETH to the user or creating a function that allows the user to collect the remaining ETH after performing a `Call` action type, taking into account that sending ETH with a push method may trigger the fallback function on the caller's address.

Update: *Fixed in PR#304 where the `payable` property is removed from the `operate` function. However this change also means it is impossible to do outbound calls which require ETH through the `operate` function.*

The Protocol uses 2 time-based prices to value all the actions related to assets: live and expired prices.

The live prices are used by the `getExcessCollateral` function from the `MarginCalculator` contract to calculate the margin of a vault before the oToken expires. After the oToken expires, the expired price is then used to calculate the vault margin.

Because the collateral asset may not be the same as the strike asset or the underlying asset, it is not guaranteed that the collateral's expiration price will be higher than its price in any point in the past, when the oToken has not expired yet, resulting in a possible undercollateralized situation for vaults.

Even though collateral assets must be whitelisted to be able to be used in the platform, with the caveat described in the issue ***oToken can be created with a non-whitelisted collateral asset***, assets such as cTokens from Compound that are supposed to gain value over time may suffer a drop in their values due to market fluctuations or events as the one in early 2020, which produced a drop in value for cDai token. When the value of all collateral is worth less than the value of all borrowed assets, we say a market is insolvent. In case the platform allows the usage of non-monotonically-increasing price assets, the insolvency may be caused by a simple market price fluctuation.

Here are other examples of things that could cause a market to become insolvent:

- The price of the underlying (or borrowed) asset makes a big, quick move during a time of high network congestion.
- The price oracle temporarily goes offline during a time of high market volatility. This could result in the oracle not updating the asset prices until after the market has become insolvent.
- The admin or oracle steals enough collateral that the market becomes insolvent.
- Administrators list an ERC20 token with a later-discovered bug that allows minting of arbitrarily many tokens. This bad token is used as collateral to borrow funds that it never intends to repay.

In any case, the effects of an insolvent market could be disastrous. It may result in a "run on the bank" situation, with the last suppliers out losing their money. It is important to know that this risk currently does not have any liquidation process, any vault is susceptible to becoming insolvent during the payout of the options right after expiration.

Consider adding a liquidation process to prevent insolvent vaults, carefully selecting the whitelisted assets for the protocol, adding more test units, and running a testnet version to understand how other assets may cause an undercollateralized scenario.

Update: *Fixed in PR355, the Opyn team restricted all oTokens issued on the protocol to be collateralized with the exact payout asset depending on the oToken type(strike asset for PUT options and underlying asset for CALL options). Although this update restricted what Options the Opyn protocol can issue, it eliminated the risk of protocol become insolvent. We suggest a liquidation component to be ready in the future should the Opyn Team decide to remove PR 355 restriction, and get it thoroughly audited.*

Medium severity

[M01][Fixed] oToken can be created with a non-whitelisted collateral asset

A product consists of a set of assets and an option type. Each product has to be whitelisted by the admin using the `whitelistProduct` function from the `Whitelist` contract.

Then, a user can call the `createOtoken` function from the `OtokenFactory` with the same assets and option type, and because the product is whitelisted, the requirement on line 70 will succeed.

However, although the product has been whitelisted, the collateral itself may not be approved. This is because the `whitelistProduct` function does not check against the `isWhitelistedCollateral` function if that collateral is allowed in the platform or not. Therefore, the first engagement with the collateral will appear on line 613 from `Controller.sol`, where the transaction will revert when the user wants to deposit some collateral in their vault.

Consider validating if the assets involved in a product have been already whitelisted before allowing the creation of oTokens.

[M02] ERC20 compliant assets may not be used

A new oToken can be created by calling the `createOtoken` function from the `OtokenFactory` contract and passing the whitelisted assets, among other parameters.

During the initialization of the new oToken, the code calls the `_getNameAndSymbol` function which retrieves the standardized symbol and name for that oToken.

Nevertheless, the same function calls every single asset involved in the oToken to get their symbol, but because the `symbol` and `name` function from the ERC20 standard are optionals, these external calls may fail if those are not implemented and the oToken will not be created.

Similarly, the `decimals` function is optional and ERC20 compliant assets may not include such function. Although in this scenario an oToken could be created using those assets, any action that would trigger the `_verifyFinalState` function from the `Controller` contract will revert either in line 90 or 101 from `MarginCalculator.sol`.

Consider being as general as possible and assuming that these functions may not be implemented in the whitelisted assets.

Update: *The Opyn team explained their views regarding this issue in a follow-up discussion. They decided not to implement any fixes and, instead, they will rely on the admin process of whitelisting underlying ERC20 tokens that conform to the requirements of their system. In the future, if the need arises, they intend to upgrade the system to support a broader range of ERC20 tokens.*

[M03][Fixed] Actions are undefined at the exact time of oToken expiry

The `oToken` contract defines an expiration time in which certain operations will no longer be active due to the expiration of the option. The option style used for the Protocol is the european option in which the exercise of the option comes at maturity.

Some actions that can be performed before the expiration time are depositing long oTokens or minting new oTokens; while after the expiration time it can be redeemed or settled a vault.

Because it is based on a European option, actions that happen after maturity should be able to be called at the exact time of expiry. Consider including and defining the expiration time to either of the 2 conditions so transactions will not revert.

Update: *Fixed in PR#291*.

[M04][Fixed] Use of transfer might render ETH impossible to withdraw

When withdrawing ETH deposits, the `PayableProxyController` contract uses Solidity's `transfer` function. This has some notable shortcomings when the withdrawer is a smart contract, which can render ETH deposits impossible to withdraw. Specifically, the withdrawal will inevitably fail when:

- The withdrawer smart contract does not implement a payable fallback function.
- The withdrawer smart contract implements a payable fallback function which uses more than 2300 gas units.
- The withdrawer smart contract implements a payable fallback function which needs less than 2300 gas units but is called through a proxy that raises the call's gas usage above 2300.

To prevent unexpected behavior and potential loss of funds, consider explicitly warning end-users about the mentioned shortcomings to raise awareness before they deposit Ether into the protocol. Additionally, note that the `sendValue` function available in OpenZeppelin Contract's `Address` library can be used to transfer the withdrawn Ether without being limited to 2300 gas units. Risks of reentrancy stemming from the use of this function can be mitigated by tightly following the "Check-effects-interactions" pattern and using OpenZeppelin Contract's `ReentrancyGuard` contract. For further reference on why using Solidity's `transfer` is no longer recommended, refer to these articles:

- Stop using Solidity's transfer now
- Reentrancy after Istanbul

Update: *Fixed in PR#305*.

[M05] User can force the Controller contract to perform an undesired external call

Besides those actions, the contract allows the execution of a more general transaction to be used as a flash loan in other projects by formatting the call with the `CallArgs` struct format.

By doing this, if the user submits that action in the `operate` function, the call would jump into the `_call` function to then perform an external call to the `callFunction` payable function in the `callee` address.

However, if the `callee` address is not a `CalleeInterface` based contract but it has either a fallback or payable fallback function in it, and it is not restricted the whitelisted addresses, the call coming from the `Controller` contract will end up executing any code under the fallback function on behalf of the `Controller` contract's address. This other address could be either an asset that may be part of the whitelisted assets in the protocol or a future contract of the project that allows the execution of sensitive actions by the `Controller` contract.

Consider preventing an external call on behalf of the `Controller` contract when the destination address is not a `CalleeInterface` type of contract and it is not a whitelisted address.

Low severity

[L01] [Fixed] Disputer can dispute a price before a pricer has submitted a price

The `Oracle` contract is the one in charge of providing the price of assets used in the platform. To do so, it uses different pricers to submit time-specific prices for different assets.

Disputers are allowed to challenge and override those prices. However, this should only be allowed after the specific price has been submitted by the pricer and before the dispute window is over.

Nevertheless, the disputer can call the `disputeExpiryPrice` function before a price and timestamp are submitted by the pricer. During this call, the `isDisputePeriodOver` function will return false due to default timestamp == 0, causing the check in line 221 from `Oracle.sol` to pass and allow the disputer to successfully dispute a non-existing price.

Moreover, this process will not update the timestamp even though the `ExpiryPriceDisputed` event log makes it look like the timestamp has been updated.

Update: *Fixed in PR#292.*

[L02][Fixed] Constants are not being declared explicitly

There are several occurrences of literal values with unexplained meaning in the Gamma Protocol's contracts. For example, line 64 in `OtokenFactory.sol`. Literal values in the codebase without an explained meaning make the code harder to read, understand and maintain, thus hindering the experience of developers, auditors and external contributors alike.

Developers should define a constant variable for every magic value used (including booleans), giving it a clear and self-explanatory name. Additionally, for complex values, inline comments explaining how they were calculated or why they were chosen are highly recommended. Following Solidity's style guide, constants should be named in `UPPER_CASE_WITH_UNDERSCORES` format, and specific public getters should be defined to read each one of them.

Update: *Fixed in PR#285.*

[L03][Fixed] Operations are not explicitly casted

The current lack of explicit casting when handling unsigned integer variables in the codebase, considering the various different types of unsigned integers in use (e.g. uint256, uint64 and uint32), hinders the code's readability, making it more error-prone and hard to maintain.

Some examples of this issue can be found at:

- In line 101 from `MarginCalculator.sol`, where an `uint8` is copied to an `uint256` variable. It should be done as in line 62.
- In line 18 from `OtokenSpawner.sol`, where a zero value is implicitly casted into a `bytes32` variable.

Consider explicitly casting all integer values to their expected type when sending them as parameters of functions and events. It is advisable to review the entire codebase and apply this recommendation to all segments of code where the issue is found.

Update: *Fixed in PR#288.*

of them are:

- The `OtokenCreated` event from the `OtokenFactory` contract does not index the assets involved.
- The `PriceUpdated` event from the `Oracle` contract does not index both asset and new pricer.

Consider indexing event parameters to avoid hindering the task of off-chain services searching and filtering for specific events.

Update: *Fixed in PR#286.*

[L05] [Fixed] Lack of input validation could pollute the event logs

The function `setOperator` of the `Controller` contract allows vault owners to nominate an account as their operator. However, the function does not check if the nominated account is already an operator, resulting in the unnecessary `AccountOperatorUpdated` event emission.

Throughout the code base, similar cases were found. Some of them are the `setSystemPartiallyPaused`, the `setSystemFullyPaused`, the `setFullPauser`, the `setPartialPauser`, and the `setCallRestriction` functions.

Consider adding proper input validation before updating the values in storage for these functions to improve the readability of the emitted logs.

Update: *Fixed in PR#289.*

[L06] Lack of input validation

In the `AddressBook` contract:

- The `setOtokenImpl` function does not check if the given address is zero. Although this function is currently restricted to onlyOwner, future updates might allow other permitted roles to call this function.

- The `disputeExpiryPrice` function allows the disputer to send a zero price value, and because it would be the final number after the disputing period is over, a zero price could revert a division calculation in other contracts that use that value.

In the `Whitelist` contract:

- The `whitelistProduct`, `whitelistCollateral`, `whitelistOtoken`, and `whitelisteCallee` functions do not check if the input is already whitelisted.
- The `blacklistProduct`, `blacklistCollateral`, `blacklistOtoken`, and `blacklistCallee` functions do not check if the input is already blacklisted.

In the `Otoken` contract:

- The `_slice` function does not check if the `_end` is before the `_start` and the operation is made without the protection of the `SafeMath` library. However, the function is only used once and both values cannot be less than 1, case that would end up with an empty string.

In the `MarginVault` library:

- The `removeShort`, `removeLong`, and `removeCollateral` functions do not check if the given amount is zero as the counter functions to add assets do.

Even though this issue does not pose a security risk, the lack of validation on user-controlled parameters may result in erroneous transactions considering that some clients may default to sending null parameters if none are specified. To favor explicitness and avoid unexpected behaviors, consider implementing require statements where appropriate to validate these parameters.

[L07][Fixed] Mismatches between contracts and interfaces

Interfaces define the exposed functionality of the implemented contracts. However, in several interfaces there are functions from the counterpart contracts that are not defined, as listed below.

The [illegible] interface is missing the [illegible] getter, the `blacklistProduct`, `blacklistCollateral`, `blacklistOtoken`, `whitelisteCallee`, and `blacklistCallee` functions.
- The `OtokenInterface` interface is missing the `addressBook` getter.
- The `MarginPoolInterface` interface is missing the `addressBook` and `farmer` getters, and the `getStoredBalance`, `farm`, and `setFarmer` functions. Also, it has a mismatch between the name of the implemented `batchTransferToPool` and `batchTransferToUser` functions and the ones in the interface.
- The `MarginCalculatorInterface` interface should use `calldata` instead of `memory` due to the explicit use of the `external` visibility in interfaces, and it is missing the `addressBook` getter.

Consider applying the necessary changes in the mentioned interfaces and contracts so that definitions and implementations fully match.

Update: *Fixed in PR#287.*

[L08] Actions not executed atomically might lead to inconsistent state

The `setAssetPricer`, `setLockingPeriod`, and `setDisputePeriod` functions of the `Oracle` contract execute actions that are always expected to be performed atomically. Failing to do so can lead to inconsistent states in the system.

In particular, after the owner calls the `setAssetPricer` function, both the dispute and locked periods are still in zero. The pricer could backrun the `setAssetPricer` function call and set a price using the `setExpiryPrice` function, skipping the locked period. Then, if the dispute period is not adjusted correctly, the expiration price will remain.

Consider implementing an additional function that calls the `setAssetPricer`, `setLockingPeriod`, and `setDisputePeriod` functions, so that these actions can be executed atomically in a single transaction.

[L09] [Fixed] oToken symbols might not be unique

Nevertheless, the symbol may not be unique due to the lack of the collateral asset information in its string, leading to a confusing state where different oTokens are mixed and displayed as the same one to end-users or other projects. For instance, the following example shows the current symbol for one oToken but it does not reflect the collateral asset's type used:

```
tokenSymbol (ex: oETHUSDC-05SEP20-200P)
```

Consider adding the collateral asset's info into the tokenSymbol string.

Update: *Fixed in PR#297.*

[L10][Fixed] String is used as hash key

The `AddressBook` contract keeps a record of all the current contracts in the Protocol so other contracts can make use of them.

The `AddressBook` contract defines several `bytes32` constants for each one of the contracts, and then it uses those constants as key for the `addresses` mapping, which goes from `bytes32` to the respective contract's address.

However, these constants are set directly as a string with an implicit casting into a `bytes32` variable type. Furthermore, although the length of these strings is short enough to fit in a `bytes32` variable, in the future a new part of the Protocol could be added and its name may not fit under such type.

It is frequent to use the hashed version of the string as the key instead of the string itself because its length will always fit under a `bytes32` variable.

Consider using the hashed version of the string instead of the direct string as the key for the `addresses` mapping.

Update: *Fixed in PR#295.*

[L11] Chainlink pricer is using a deprecated API

working if Chainlink stop supporting deprecated APIs.

Consider refactoring these to use the latest Chainlink API.

[L12] Stable asset price could be stuck if mistakenly set with setAssetPricer()

The update introduced a `stablePrice[_asset]` mapping to track stable asset prices, while the original `assetPricer[]` tracks all other asset prices. However if any stable asset price was mistakenly set with `setAssetPricer()` into `assetPricer[]`, there is no way to delete this record as `setAssetPricer()` does not allow setting records to `address(0)`, this will jam the `setStablePrice()` for this asset because it requires `assetPricer[asset] == address(0)`.

Although this is only possible if the admin made a mistake first, the consequence is quite severe if it does happen. Consider allowing deletion of `assetPricer[]` record in `setAssetPricer()` in case the system gets stuck.

Notes & Additional Information

[N01][Partially Fixed] Functions default to a particular case

Throughout the codebase, there have been cases where functions default any other case into a particular defined outcome. Some of these are:

- In the `_getMonth` function from the `Otoken` contract, any `_month` greater or equal than 12 will return the short and long string for December.
- In the `_runActions` function from the `Controller` contract, the function to execute is based on the action passed in the arguments. However, the case for a `Call` action is left as the default case instead of defining a particular case as it was done with the rest of the actions.

Although these cases do not represent a security risk, consider defining one outcome to each one of the cases and use the default case for those situations that do not match with any other particular case to reduce the attack surface and improve the readability of the code.

[N02] Disputer may not be able to dispute

The `Oracle` contract allows a disputer to dispute the price after the respective asset's pricer has submitted a price.

This can be done by calling the `disputeExpiryPrice` function and defining the new price, as long as the disputer submits their price before the disputing windows is over.

Nevertheless, the admin can set this period to zero, by calling the `setDisputePeriod` function, which would prevent the disputer from disputing a price.

However, because the disputer is also defined by the admin, this should not be a risk unless the admin is compromised.

Consider either restricting the dispute period to a value greater than zero or documenting the reason of having a zero-size window to dispute a price.

[N03][Fixed] Refactoring opportunities

There are a number of functions in the codebase that can benefit from a code refactor. In particular:

- In line 204 from `AddressBook.sol`, the `params` variable is only used when the `proxyAddress` is zero. It could be moved inside of the `if` statement.
- In line 175 from `OtokenFactory.sol`, the address of the oToken implementation is retrieved from the `AddressBook` contract to then be used in the `_computeAddress` function. However, that address was already retrieved in line 164 of the same function.
- In lines 145 and 146 from `MarginCalculator.sol`, the `_getMarginRequired` function checks if the vault has long or short oTokens. However, the `getExcessCollateral` function which calls the `_getMarginRequired` function, also checks the same thing on lines 79 and 80.
- In lines 239 and 240 from `Controller.sol`, the `initialize` function calls the `__Context_init_unchained` and `__Ownable_init_unchained` functions from the `OwnableUpgradeSafe` contract can be replaced by the `__Ownable_init` function from the same contract.

Update: *Fixed in PR#299.*

[N04] Inconsistent coding style

The codebase is not always following a consistent coding style. Some examples are:

- The usage of an implicit returned value, such as on line 61 from `OtokenSpawner.sol`, while other functions from the same contract do explicitly return the value, such as on line 35.
- The usage of named return variables, such as on line 205 from `Otoken.sol`, but unnamed returns in other functions of the same contract, such as on line 191.
- Different method to interact with a variable, such as the `vaults` mapping on lines 636 and 646 from `Controller.sol`.

Consider always following the same style to improve the project's readability. As a reference, consider always following the style proposed in Solidity's Style Guide. Taking into consideration how much value a consistent coding style adds to the project's readability, enforcing a standard coding style with help of linter tools such as Solhint is recommended.

[N05][Fixed] Repeated value in event

The `PricerUpdated` event from the `Oracle` contract is triggered when the admin sets or changes the pricer for a particular asset using the `setAssetPricer` function.

The event is defined to log both the asset's address and the new pricer address. However, during the `setAssetPricer` function execution, the event is logging the asset's address twice instead of the new pricer.

Consider logging both the asset and the new pricer in the event.

Update: *Fixed in PR#283.*

[N06][Fixed] Typographical errors and misleading comments

Several docstrings, inline comments, and variable names throughout the codebase were found to be erroneous and/or incomplete and should be fixed. In particular:

a price during the dispute period, by setting a new one" but actually the disputer is the only one allowed.
- In line 231 from `Oracle.sol`, the inline documentation states "submits the expiry price to the oracle, can only be set from the pricer" but the disputer is also allowed to call that function.
- In lines 90 and 91 from `MarginCalculator.sol`, `colllateralDecimals` should say `collateralDecimals`.
- In line 141 from `MarginVault.sol`, the revert message states "MarginVault: long otoken address mismatch" but it also could be made by an out-of-bound-index related issue.
- In line 215 from `Whitelist.sol`, `whitelisteCallee` should be named `whitelistCallee`.
- In line 197 from `AddressBook.sol`, the inline documentation states that the function is internal but actually the function has public visibility.
- In line 47 of PayableProxyController.sol in PR279, "to wrap WETH and the beginning" should be "to wrap WETH at the beginning"

Clear docstrings are fundamental to outline the intentions of the code. Mismatches between them and the implementation can lead to serious misconceptions about how the system is expected to behave. Therefore, consider fixing these errors to avoid confusions in developers, users, auditors alike.

Update: *Fixed in PR#302.*

[N07][Fixed] Potential fake payableProxyController.sol

The payable Proxy Controller works in between the caller and the controller contract to handle ETH payment, wrapping and unwrapping. There isn't any built-in mechanism in the smart contract to verify this proxy address is the official one which means a hacker could deploy their own proxy with fraudulent `weth` address to steal all user deposit if they can somehow convince users to use it or make it look like the official proxy.

This might not be an issue depending on how security is handled from the front end, which is out of scope for this audit. We just want to gently remind the team to ensure safe communication with *this attack including a close sourced front end, a document listing official smart contract addresses.*

Conclusions

1 critical and 2 high severity issues were found. Some changes were proposed to follow best practices, reduce the potential attack surface, and enhance overall code quality.

Related Posts

Beefy Zap Audit

BeefyZapRouter serves as a versatile intermediary designed to execute users' orders through routes...

Security Audits

OpenBrush Contracts Library Security Review

OpenBrush is an open-source smart contract library written in the Rust programming language and the...

Security Audits

Linea Bridge Audit

Linea is a ZK-rollup deployed on top of Ethereum. It is designed to be EVM-compatible and aims to...

Security Audits

OpenZeppelin

OpenZeppelin

Defender Platform

Secure Code & Audit
Secure Deploy
Threat Monitoring
Incident Response
Operation and Automation

Services

Smart Contract Security Audit
Incident Response
Zero Knowledge Proof Practice

Learn

Docs
Ethernaut CTF
Blog

Company

About us
Jobs
Blog

Contracts Library

Docs