Opyn Bull Strategy Contracts Audit

OPENZEPPELIN SECURITY | DECEMBER 19, 2022

Security Audits

December 13th, 2022

This security assessment was prepared by **OpenZeppelin**.

Table of Contents

Summary

Type
: DeFi

Timeline
: From 2022-11-14
: To 2022-12-13

Languages
: Solidity

Total Issues
: 20 (9 resolved, 4 partially resolved)

Critical Severity Issues
: 0 (0 resolved)

High Severity Issues
: 0 (0 resolved)

Medium Severity Issues
: 2 (0 resolved, 2 partially resolved)

Low Severity Issues
: 4 (2 resolved)

Notes & Additional Information

We audited the `opynfinance/squeeth-monorepo` repository at the `521203dc26e5a2c3e26c6d4ad02d513e7df63237` commit.

In scope were the following contracts:

- contracts/
  - BullStrategy.sol
  - AuctionBull.sol
  - EmergencyShutdown.sol
  - FlashBull.sol
  - LeverageBull.sol
  - interfaces/
    - IBullStrategy.sol
    - ICrabStrategyV2.sol
    - IEulerDToken.sol
    - IEulerEToken.sol
    - IEulerMarkets.sol

System overview

The system we audited is for an automated trading strategy that allows investors to execute a specific trade-flow and reap its returns. Different strategies have distinct objectives, exposing investors to different gains (or losses) based on their respective market paths. The audited strategy, named `Bull`, is comprised of a balance between two sub-strategies with the ultimate objective of allowing an investor (i.e. user) to benefit from a rise of the price of ETH, while collecting payments of ETH should its price remain flat.

The system relies heavily on a few critical components, which are out of scope for the current audit. Primarily, there's Squeeth which is a product through which users can trade *long* or *short* call options on a particular index that aims to track the value of `eth^2`. For a user to be able to trade long on Squeeth, they can buy `oSQTH`, the token representing Squeeth, in a liquidity pool in Uniswap v3. They must pay a premium for this position. Conversely, to trade short on Squeeth,

Secondly, there's the sub-strategy `Crab` which was not in scope for this audit. Crab's objective is to allow investors to profit from a low volatility environment. It does this by allowing a user to short Squeeth to collect the premiums from that position, and offset the risk of the price moving down by buying enough ETH to be "delta zero" (i.e. a small move in the price will have gains and losses of near zero). This strategy needs to be rebalanced periodically to have the correct positioning and this is triggered either by specific time periods or if the positions are out of balance enough (e.g. after a large price change). The Crab Strategy lets users deposit ETH to participate in it. The strategy itself must have at least a safe collateralization ratio of 150% in order to avoid liquidation. Users will receive crab tokens that represent their share of the strategy.

The contracts we have audited pertain to the Bull Strategy, which aims to maintain 100% exposure to ETH (i.e. delta of one) while profiting from the premiums generated by the Crab Strategy. This is achieved by combining a position in Crab with a leveraged position on Euler, where `WETH` is used as collateral and `USDC` is borrowed and sold in order to create the leverage. Since the Crab Strategy is delta neutral, the positive exposure to ETH is given by the leveraged position on Euler.

The core component of the strategy is the `BullStrategy` contract, which is an ERC20 contract on its own, representing users' positions in terms of Bull strategy tokens. It also extends from `LeverageBull`, which is the main component in charge of handling the leveraged position on Euler, having methods to deposit and borrow from it, plus some additional methods to handle proper amount conversions using Uniswap TWAP oracles. While extending from it, the `BullStrategy` adds methods on top of it to also handle Crab Strategy deposits and withdrawals. In this sense, the Bull Strategy must be seen as a single position in the Crab Strategy itself. The `BullStrategy` also defines a hard cap to limit the strategy itself and a shutdown contract that is used in shutdown operations.

Shutdown operations are managed by the `EmergencyShutdown` contract. Its sole purpose is to close the Crab and Euler positions in order to collect all the possibly recoverable ETH and then allow users to withdraw their share from the `BullStrategy` contract.

Another important piece of the protocol is the `FlashBull` contract, a wrapper around the `BullStrategy` deposit and withdraw functions that makes use of Uniswap flash swaps to

Lastly, the `AuctionBull` contract is the core logic for the rebalancing operations. It is owned by an `auctionManager` and is able to perform either a *full rebalance* – a type of rebalance where both the Crab and Euler positions are modified to make sure all ratios are properly set – or just a *leverage rebalance* where only the latter is tweaked. The full rebalance makes use of a set of orders sent by traders which have been submitted and signed off-chain, and then ordered and validated by a backend service. The goal is to allow external traders to bid on rebalances and use their bids to actually rebalance the system, avoiding a potentially large slippage if Uniswap pools were used instead.

Privileged roles

Within the Bull Strategy system, there are mainly two special roles to take into account, the `owner` and the `auctionManager`. Both roles have the following responsibilities and associated trust assumptions.

The `owner` of the `BullStrategy` contract can:

- Call the `farm` function to sweep tokens stuck in the contract (except `WETH`, `USDC`, `dToken`, `eToken` and `oSQTH`)
- Arbitrarily set a strategy cap, which is enforced as the maximum `WETH` collateral than can be owned within the leverage component in Euler
- Set the address of the `EmergencyShutdown` contract
- Set the address of the `AuctionBull` contract
- Transfer ownership to any other address

The `owner` of the `AuctionBull` contract can:

- Set the address of the auction manager
- Set clearing prices tolerance for full rebalance and for WETH specifically
- Set collateral ratio lower/upper limits (notice that there is no requirement for these bounds to be aligned with the liquidation threshold)
- Set delta to ETH price lower/upper limits (notice that there is no requirement for these bounds to be in line with the strategy parameters)

- Trigger the `redeemShortShutdown` function to initiate a shutdown in the Bull Strategy. This `owner` will need to avoid improper shutdown management, such as not shutting down Squeeth controller before the Crab and Bull strategies. This could potentially cause the `BullStrategy` contract to be drained if there is a partial redemption without the Squeeth controller already being shut down. Specifically, executing a partial redemption (`shareToUnwind < 1`) without the Squeeth controller being already shut down would convert the strategy funds into ETH sitting in the contract without triggering the `hasRedeemedInShutdown` flag, thus allowing any user to `withdraw` an arbitrary amount of shares and resulting in them receiving the entire ETH balance of `BullStrategy`
- Transfer ownership to any other address

We assume that these owners will set appropriate values for each one of those parameters and will correctly handle calls to the `farm` and `redeemShortShutdown` functions.

The `auctionManager` role of the `AuctionBull` contract can:

- Trigger a full rebalance
- Trigger a leverage rebalance

We assume that this account will run healthy auctions with a healthy orders list, and that it will not run empty auctions with the side-effects of exclusively moving funds. Specifically, the Bull Strategy has infinite approval over `AuctionBull` funds.

Update: *With the changes introduced in PR #782, the excess ETH deposited will be refunded by transferring the entire contract balance to the depositor. This means that the trust assumption about an improper shutdown protocol extends to any user calling `deposit` within `BullStrategy` with an arbitrary amount, not only to existing depositors calling `withdraw`.*

Trust assumptions

- **UniFlash**: Abstract dependency that is meant to provide the necessary infrastructure to perform flashswaps against Uniswap V3 pools.
- **Euler component**: We assume that Euler Finance will operate with no downtime, and no security bugs or exploits. Note that Euler benefits from infinite `WETH` and `USDC` approval from the `BullStrategy` contract.
- **ETH/USDC and oSQTH/ETH Uniswap v3 pools** are heavily relied upon for both depositing, withdrawing and rebalancing the strategy. It is assumed there will always be enough liquidity to carry out normal operations.
- **Uniswap v3 TWAP Oracle** is assumed to work as expected, with no downtimes, hacks or bugs. Note that there is no fallback mechanism in place in order to mitigate potential price manipulation attacks.
- **Crab Strategy** is a core component of the Bull Strategy, and it is expected to work as intended, with no security vulnerabilities.
- **Squeeth controller** is another core component of the Bull Strategy, and it is also expected to work as intended.
- **`EmergencyShutdown owner`** is expected to diligently perform the shutdown protocol, by first shutting down the Squeeth controller, then shutting down the Crab Strategy and finally the Bull Strategy. As mentioned before, mistakes in this protocol can potentially end up draining user funds.
- **The off-chain auction management** system is expected to perform multiple checks on all trader bids before submitting them to the contract, calculating sensitive parameters accurately in order to always keep user interests at heart. Notice that broad parameter settings might have a large impact on rebalancing costs, which are absorbed by all depositors.

Given the sensitive external dependencies (Uniswap, Euler, Crab Strategy and Squeeth) consider whether the emergency shutdown procedure should be extended so that, in case of an emergency specific to Bull Strategy, it can redeem its crab and leveraged positions in a secure way, without the Squeeth controller necessarily being shutdown. Alternatively, consider adding an emergency pause/unpause on deposits and withdrawals, exclusively for the Bull Strategy. While doing so, consider using the OpenZeppelin contracts library and OpenZeppelin Defender to promptly react upon unforeseen scenarios.

During the audit, the client reported two issues within the `AuctionBull` when executing a `fullRebalance`:

1) When `isDepositingInCrab == true` and `wethTargetInEuler <= current WETH collateral in Euler`, there is no execution flow to determine whether the excess collateral should be sold for USDC in order to repay some Euler debt, or whether it should be used along with further borrowing in order to deposit more WETH in Crab. The current implementation always borrows more from Euler in order to deposit a larger amount in Crab, causing some rebalances to revert.

2) When `isDepositingInCrab == false` and `wethTargetInEuler > current WETH collateral in Euler + current WETH balance`, there is no check to see if the actual Euler collateral is beyond the target or not. However, it is always assumed that it is, because it is always combining the current `WETH` balance with Euler's collateral. This will always cause withdrawals from Euler, and therefore unwanted reverts when Euler's collateral is not higher than the target.

Update: *The Opyn team resolved both scenarios in PR #746.*

Findings

Medium Severity

Potential DoS in rebalance feature

The `BullStrategy` relies on a rebalancing mechanism coded into the `AuctionBull` contract to rebalance the strategy's ETH exposure against the Crab strategy and/or Euler's loan. The rebalancing mechanism works with a special EOA, namely the `auctionManager` that calls either the `fullRebalance` or `leverageRebalance` functions in the contract. In the case of a `fullRebalance`, both the Crab strategy and Euler's leveraged position are updated.

are actually processed and included in a block there could be many ways in which individual `orders` fail during a rebalance. Some of them are:

- Lack of allowance in tokens transfer: the user might have provided a valid approval first, but then revoked it by front running the rebalance transaction.
- Lack of funds: the user might have no funds when submitting the order, or might have transferred them away after right before the rebalance transaction takes places.
- Wrong signature (the ECDSA check might fail).
- Wrong order on-chain (orders are not finally ordered from best to worst).
- Very tight expiration set: the order might be expired by the time it's confirmed in high gas price scenarios.
- Sending a certain amount of `WETH` to the `AuctionBull` contract when depositing into Crab might cause the formula that calculates how much `WETH` is needed from Uniswap to revert, since it can become a negative number, thus forcing the rebalance to revert. This can be fixed by keeping an internal accounting of how much `WETH` the contract got from the traders' orders instead of checking the entire contract balance.
- Signing a lot of valid `orders` with a competitive price will force the rebalance transaction to use up a lot of gas since there are many safety checks involved and transfers of funds. An attacker can craft enough valid orders from different wallets and IPs in order to build up a list of orders so large that it will use more than the maximum gas per block (30M gas units at the time of writing). Consider ordering `Orders` not only by price, but also keeping in mind the order quantity.
- There is no check of whether all the funds pulled from traders are enough to cover the entire rebalance. It is assumed so, and it eventually reverts at the very end when not enough funds can be pulled into the proper contract. Consider checking if a partial rebalance can be achieved with the pulled orders while still keeping acceptable CR and delta values.
- If anyone forces `ETH` into the `BullStrategy` contract, it will be converted into `WETH` and transferred into `AuctionBull` when performing a full rebalance that needs to withdraw from Crab. Additionally, if Euler's current collateral is larger than the target collateral, it will sell the entire `WETH` balance for `USDC` in order to repay some debt. The same effect can be achieved by sending `USDC` to the `AuctionBull` contract directly. If it's higher than Euler's debt, repayments will never be possible on full rebalances when withdrawing from Crab and repaying Euler's debt. It doesn't need to be that large of an

- The borrow operation from Euler can fail if there's not enough USDC available for borrowing. A third party can temporarily borrow enough from the lending pool prior to the rebalance in order to make sure there will never be enough USDC to rebalance the strategy. There's no way to recover from this, apart from waiting / depositing more collateral from another account in order to increase available USDC to borrow. A similar approach was used recently by an attacker on Aave.

Given the different possibilities of how a rebalance can be forced to revert, consider taking all of them into account for the off-chain orders selection and ordering algorithms, but also when it comes to doing safety checks and operations on-chain.

Also consider always using a private transaction relay service (such as FlashBots) to broadcast the rebalance transaction in order to avoid these potential attacks.

Update: *Partially resolved in PR #789 by implementing a* `farm` *function that allows the contract* `owner` *to retrieve any asset balance from* `AuctionBull`*, and expanding the pre-existing* `farm` *function within* `BullStrategy` *so* `ETH`*,* `WETH` *and* `USDC` *can also be recovered by the owner if necessary.*

Artificial asset balance inflation

All the contracts in scope are susceptible to an asset balance inflation attack, where a user might send ETH or any ERC20 token directly to the protocol contracts.

For ETH specifically, the only way to do so is by making use of an intermediate contract that uses `selfdestruct` to force funds directly into any protocol contract, bypassing the `receive` require statements in place.

The `farm` function can recover stray ERC20 token balances (except the excluded ones), but there is no general way to withdraw ETH. One interesting side effect from the rebalancing mechanism is that `AuctionBull` is capable of wrapping the entire ETH balance into `WETH` and then transferring it out of `BullStrategy` during certain flows of a full rebalance. However, this does not solve the issue entirely. Moreover, the `farm` function is not present everywhere. For example, it is absent in `AuctionBull`.

risk of an inflation attack.

Consider whether it is safe to leave the doors open for such scenarios and whether it is relevant to include some mitigations, such as implementing a less strict `farm` function to be used across all contracts, or having a restricted function to retrieve stuck ETH.

Update: *Partially resolved in PR #789 by implementing a* `farm` *function that allows the contract* `owner` *to retrieve any asset balance from* `AuctionBull`*, and expanding the pre-existing* `farm` *function within* `BullStrategy` *so* `ETH`*,* `WETH` *and* `USDC` *can also be recovered by the owner if necessary.*

However, the `farm` *function within* `BullStrategy` *should be restricted when attempting to recover* `ETH` *if the* `Squeeth` *controller is shut down, since all user funds will be sitting in* `ETH` *during that time.*

Low Severity

Rebalances are subject to sandwich attacks

Rebalancing transactions are very meaningful events within the strategy lifecycle. Potentially large amounts of `WETH` and `USDC` might be involved, and given the necessary use of Uniswap V3 pools in any scenario (apart from OTC auctions) there are two potential scenarios in which placing a transaction before and after the rebalancing might be beneficial for a third party:

- In the case of a third party willing to profit off the protocol, a `wethLimitPrice` is specified in both the `leverageRebalance` and `fullRebalance`. This value determines the maximum slippage for each swap, so special care should be placed on the calculation of this value and the tolerance parameters in order to avoid being sandwiched out of this slippage.
- A rebalancing event inevitably carries some costs for all the strategy depositors due to slippage, price impact and spread. This cost will always be a percentage of the entire strategy portfolio value. Any investor within the strategy can sandwich the rebalance transaction in order to exit before it happens, and re-enter it immediately afterwards. The incurred cost of doing so might be much lower than the rebalancing costs, so if executed correctly, they would avoid that extra cost at the expense of the rest of the depositors.

In both cases, consider if these are relevant risks and always relay the rebalancing transaction through a private relay like FlashBots in order to avoid undesired reverts due to DoS attacks, and unexpected losses in slippage due to sandwich attacks.

Duplicated code

There are instances of duplicated code within the codebase. This can lead to issues later on in the development lifecycle, and leaves the project more prone to the introduction of errors. Errors can inadvertently be introduced when functionality changes are not replicated across all instances of code that should be identical. One example of duplicated code is the function `_calcWsqueethToMintAndFee` which is defined both in the `FlashBull` and `AuctionBull` contracts.

Instead of duplicating code, consider having just one contract or library containing the duplicated code and using it whenever the duplicated functionality is required.

Missing docstrings

Throughout the codebase there are several parts that do not have docstrings. Some examples are:

- Line 408 of the `AuctionBull` contract
- Line 96 of the `LeverageBull` contract
- Line 105 of the `LeverageBull` contract
- Line 121 of the `LeverageBull` contract

Additionally, the `AuctionBull` NatSpec documentation is incorrect, since it is referencing the `FlashBull` contract instead.

Consider thoroughly documenting all functions (and their parameters) that are part of the contracts' public API. Functions implementing sensitive functionality, even if not public, should be clearly documented as well. When writing docstrings, consider following the Ethereum Natural Specification Format (NatSpec).

Update: *Resolved in PR #775, PR #776, PR #818 and PR #822.*

Missing error messages in require statements

- The `require` statement on line 202 of `AuctionBull.sol`
- The `require` statement on line 126 of `FlashBull.sol`

Contrast this with the `BullStrategy`'s identical line 90 require statement, which includes an error message.

Consider including specific, informative error messages in all `require` statements to improve overall code clarity and facilitate troubleshooting whenever a requirement is not satisfied. This should also improve consistency across several contracts.

Update: *Resolved in PR #776.*

Notes & Additional Information

Lack of indexed parameters

Throughout the codebase, events do not have their parameters indexed. Some examples are:

- line 139 of the `AuctionBull` contract
- line 54 of the `BullStrategy` contract
- line 101 of the `FlashBull` contract
- line 59 of the `LeverageBull` contract

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

Update: *Resolved in PR #748.*

Outdated solidity versions

Throughout the codebase there are `pragma` statements that use an outdated version of Solidity. For instance:

- The `pragma` statement on line 2 of the `AuctionBull` contract
- The `pragma` statement on line 2 of the `BullStrategy` contract
- The `pragma` statement on line 2 of the `EmergencyShutdown` contract

Consider taking advantage of the latest Solidity version to improve the overall readability and security of the codebase.

Unused function parameter

Within the `AuctionBull` contract at the `_pushFundsFromOrders` function, the `_isDepositingInCrab` parameter is not used.

To improve the overall clarity, intentionality, and readability of the codebase, consider removing any unused function parameters.

Update: *Resolved in PR #779.*

Unused imports

Throughout the codebase, imports on the following lines are unused and could be removed:

- Line 10 of the `EmergencyShutdown.sol` contract
- Line 10 of the `FlashBull.sol` contract
- Line 5 of the `IBullStrategy.sol` contract (eventually consider if the defined interface should extend from the imported one)

Consider removing unused imports to avoid confusion that could reduce the overall clarity and readability of the codebase.

Update: *Resolved in commit ab8226c.*

Inconsistent event emission parameters

Throughout the codebase, some inconsistencies were found regarding event emission parameters.

Events presenting an incorrect parameter ordering include:

- The SetCap event has its old and new value parameters in a different order than the SetShutdownContract event. This causes the setShutdownContract function to emit the event with the parameters in the wrong order.

Consider emitting both events with the same parameter ordering for consistency, and to avoid hindering the task of off-chain event analysis.

Additionally, events missing important parameters include:

- When using the `flashDeposit` and `flashWithdraw` functions, both the `Deposit` and `Withdraw` events are fired specifying `msg.sender` as the depositor, which will always resolve to the `FlashBull` contract address.
- The specific events `FlashDeposit` and `FlashWithdraw` do not contain any information about the actual user performing the flash deposit / withdrawal.

Consider refactoring the event emission logic for deposits and withdrawals so that information is complete and accurate.

Update: *Partially resolved in PR #785. The Opyn team updated the specified setter events to follow a consistent parameter ordering, and added a parameter to both the `FlashDeposit` and `FlashWithdraw` events to indicate the user who performed the action. However, both the `Deposit` and `Withdraw` events still do not contain information about the original user, since they will always emit the `FlashBull` contract address as the `from` and `to` parameters respectively.*

Not checking ERC20 transfers return parameter

In the `LeverageBull` contract, there are several calls to the EIP20 standard `transfer` and `transferFrom`. These calls return a `bool` parameter that is not checked. In the scenarios that the team explored, this is not a security issue since every token involved provides reverts in the code for failing transfers.

However, as new tokens may be integrated in the future, usage of this pattern could cause issues because new tokens could lack this reverting behavior.

Consider always using the `SafeERC20` contract from OpenZeppelin to wrap for such calls, or evaluating the return parameter to be `true`.

Unnamed return parameters

- In the _calcWsqueethToMintAndFee, _calcSharesToMint and _getCrabVaultDetails functions of the `FlashBull` contract
- In the _getCurrentDeltaAndCollatRatio, getCurrentDeltaAndCollatRatio, _calcWPowerPerpAmountFromCrab and _calcWsqueethToMintAndFee functions of the `AuctionBull` contract

Consider adding and using named return parameters to improve explicitness and readability.

Lack of event emission for sensitive actions

In contrast to the Crab Strategy, Bull Strategy withdrawals performed when Squeeth contracts have been shutdown do not emit events.

Consider emitting descriptive events in order to properly track these sensitive actions.

Update: *Resolved in PR #748.*

Misleading parameter passing

When depositing into the strategy directly with `Crab` tokens, there is a mixed use of the variables `_crabAmount` and `bullToMint`. The first one refers to the amount of `Crab` tokens being deposited, while the latter represents the amount of `Bull` tokens to be minted.

There are a few occurrences around the code where these two variables are used indistinctly:

- When total `Bull` supply is zero, the variable passed to the `_mint` function is `_crabAmount`, when it should be `bullToMint`.
- When calling `_leverageDeposit`, the documentation states that the second parameter should be the amount of Crab tokens being deposited, but the passed parameter is `bullToMint`.

Even though these variables might have the exact same value, using them indistinctly hinders readability and is error-prone. Consider using the appropriate variable each time.

Update: *Resolved in PR #778.*

the `quoteCurrency` for the debt component within Euler. Its address is fetched via calling the `quoteCurrency` function within the controller, but the decimals difference between WETH and USDC is hardcoded to be 12 as a constant.

Within the `Controller` contract, it is not assumed nor stated that the quote currency is `USDC`.

Consider dynamically fetching the number of decimals of the quote currency and calculating the decimals difference against `WETH`.

Inconsistent coding style

The codebase presents some code style inconsistencies.

For example, within `AuctionBull`,
the `_checkFullRebalanceClearingPrice` function is using `mul` and `div` functions,
instead of `wmul` as the `_checkRebalanceLimitPrice` function.

Consider using the same math functions for both situations.

Update: *Resolved in PR #780.*

Enum `null` values are used with a meaningful purpose

Throughout the codebase, every `enum` definition matches the value `0` to some meaningful state.

Some examples can be found within the `AuctionBull` and `FlashBull` contracts.

When a variable using those `enums` is not set, they will default to the null value, potentially causing confusion over whether it is an actual value or a null state.

Consider using `NULL` as the first `enum` state in order to avoid using a meaningful state on null values.

Gas optimizations

Notable gas cost improvements were found throughout the codebase. Some examples include:

from `BullStrategy`, which in turn approves `CrabStrategy` to pull them. Finally, `CrabStrategy` ends up executing a `transferFrom` to get them. Trader funds could be pulled directly into `CrabStrategy` to get significant gas savings on auctions.

- When performing a `leverageRebalance`, given `isSellingUsdc = true`, `WETH` tokens are transferred from the Uniswap pool into the `AuctionBull` contract, only to have them pulled from `BullStrategy` so they can be deposited within Euler.

Consider refactoring the code, so that intermediate `transfers` and `approvals` can be removed in order to get significant gas savings.

Update: *Partially resolved in PR #744 by granting infinite approval for the relevant tokens on both `FlashBull` and `BullStrategy` constructors instead of on every individual deposit and withdrawal. Additionally, a small gas optimization was added in PR #743 to avoid repeating the same function call several times. Notice that each individual point above still applies. Moreover, having infinite allowances must be weighted against potential bugs introduced in the future that might take advantage of such approvals.*

Typo in event parameter names

In both the TransferToOrder and TransferFromOrder events, the second parameter should be named “quantity” instead of “quanity”.

Update: *Resolved in PR #748.*

Conclusions

No high or critical vulnerabilities have been found. Even though the system presents complex integrations and a non-trivial design, we are happy to see robustness and prevention of small edge cases and scenarios. Given the overall complexity and the out-of-scope parts of the project, we wanted to explicitly highlight the sensitive out-of-scope parts of the system that need special attention. Finally, we appreciated that the project came with a comprehensive test suite, and that the team provided detailed and specific documentation.

properly tested and that overall coverage is not affected.

Appendix

Monitoring Recommendations

`BullStrategy` is a complex trading strategy that needs periodic rebalancing. Since these rebalancing events are critical for the strategy to work as expected, we recommend the following sensitive actions to be monitored:

- Monitor all restricted functions (functions only called by the multisig owner) to ensure that all actions are authorized by the team and that the values they set are in line with their expectations (especially those related to the emergency shutdown protocol).
- Monitor the functions `flashDeposit`, `deposit`, `flashWithdraw` and `withdraw` for large deposits / withdrawals.

This can also be extended by all privileged roles' actions, so that the community can always check for unexpected special actions.

The liquidity pools used by the project are normal Uniswap v3 pools, and the TWAP oracle is used to retrieve price data from them. Monitoring pool activities for abnormal behaviours might be useful for the team in order to prevent unwanted scenarios in high volatility or manipulation attempts.

Related Posts

Beefy
Zap Audit

BRUSHFAM
OpenBrush Contracts Library Security Review

Linea
Bridge Audit

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