SMART CONTRACTS REVIEW

zokyo

October 20th 2023 | v. 1.0

Security Audit Score

PASS

Zokyo Security has concluded that these smart contracts passed a security audit.

ZOKYO AUDIT SCORING EMBER LABS

1. Severity of Issues:
    - Critical: Direct, immediate risks to funds or the integrity of the contract. Typically, these would have a very high weight.
    - High: Important issues that can compromise the contract in certain scenarios.
    - Medium: Issues that might not pose immediate threats but represent significant deviations from best practices.
    - Low: Smaller issues that might not pose security risks but are still noteworthy.
    - Informational: Generally, observations or suggestions that don't point to vulnerabilities but can be improvements or best practices.
2. Test Coverage: The percentage of the codebase that's covered by tests. High test coverage often suggests thorough testing practices and can increase the score.
3. Code Quality: This is more subjective, but contracts that follow best practices, are well-commented, and show good organization might receive higher scores.
4. Documentation: Comprehensive and clear documentation might improve the score, as it shows thoroughness.
5. Consistency: Consistency in coding patterns, naming, etc., can also factor into the score.
6. Response to Identified Issues: Some audits might consider how quickly and effectively the team responds to identified issues.

HYPOTHETICAL SCORING CALCULATION:

Let's assume each issue has a weight:
- Critical: -30 points
- High: -20 points
- Medium: -10 points
- Low: -5 points
- Informational: -1 point

Starting with a perfect score of 100:
- 0 Critical issues: 0 points deducted
- 0 High issues: 1 resolved = 0 points deducted
- Medium issues: 6 issues (4 resolved and 2 acknowledged) = - 8 points total.
- Low issues: 4 issues (3 resolved and 1 acknowledged) = -4 points total.- Informational issues: 15 issues (8 resolved and 7 acknowledged) = - 4 points total.
+5 points for Client Unit testing Client had tests written for the contract functionalities.

Thus, 100 - 8 - 4 - 4 +5 = 89

TECHNICAL SUMMARY

This document outlines the overall security of the Ember Labs smart contracts evaluated by the Zokyo Security team.

The scope of this audit was to analyze and document the Ember Labs smart contracts codebase for quality, security, and correctness.

Contract Status

There were 0 critical issues found during the audit. (See Complete Analysis)

It should be noted that this audit is not an endorsement of the reliability or effectiveness of the contracts but rather limited to an assessment of the logic and implementation. In order to ensure a secure contract that can withstand the Ethereum network's fast-paced and rapidly changing environment, we recommend that the Ember Labs team put in place a bug bounty program to encourage further active analysis of the smart contracts.

Table of Contents

AUDITING STRATEGY AND TECHNIQUES APPLIED

The source code of the smart contract was taken from the Ember Labs repository:
Repository - https://github.com/EmberFinance/LaaS

Last commit: - a77fdd1

Within the scope of this audit, the team of auditors reviewed the following contract(s):

- EmberVault.sol
- esEMBR.sol
- esEMBRRewardsDistributor.sol
- GenericERC20Token.sol
- Vester.sol

During the audit, Zokyo Security ensured that the contract:

- Implements and adheres to the existing standards appropriately and effectively;
- The documentation and code comments match the logic and behavior;
- Distributes tokens in a manner that matches calculations;
- Follows best practices, efficiently using resources without unnecessary waste;
- Uses methods safe from reentrance attacks;
- Is not affected by the most recent vulnerabilities;
- Meets best practices in code readability, etc.

Zokyo Security has followed best practices and industry-standard techniques to verify the implementation of Ember Labs smart contracts. To do so, the code was reviewed line by line by our smart contract developers, who documented even minor issues as they were discovered. In summary, our strategies consist largely of manual collaboration between multiple team members at each stage of the review:

01 Due diligence in assessing the overall code quality of the codebase.

02 Cross-comparison with other, similar smart contracts by industry leaders.

03 Thorough manual review of the codebase line by line.

Executive Summary

The Zokyo team uncovered a one high-severity issue and six medium-severity issues. Furthermore, there are low-severity and informational issues, but no critical concerns have been identified among them. Detailed information about these findings can be found in the "Complete Analysis" section.

STRUCTURE AND ORGANIZATION OF THE DOCUMENT

For the ease of navigation, the following sections are arranged from the most to the least critical ones. Issues are tagged as “Resolved” or “Unresolved” or “Acknowledged” depending on whether they have been fixed or addressed. Acknowledged means that the issue was sent to the Ember Labs team and the Ember Labs team is aware of it, but they have chosen to not solve it. The issues that are tagged as “Verified” contain unclear or suspicious functionality that either needs explanation from the Client or remains disregarded by the Client. Furthermore, the severity of each issue is written as assessed by the risk of exploitation or other unexpected or otherwise unsafe behavior:

Critical

The issue affects the contract in such a way that funds may be lost, allocated incorrectly, or otherwise result in a significant loss.

High

The issue affects the ability of the contract to compile or operate in a significant way.

Medium

The issue affects the ability of the contract to operate in a way that doesn’t significantly hinder its behavior.

Low

The issue has minimal impact on the contract’s ability to operate.

Informational

The issue has no impact on the contract’s ability to operate.

COMPLETE ANALYSIS

FINDINGS SUMMARY

#	Title	Risk	Status
1	Vesting Contract Can Be Overridden	High	Resolved
2	Possible Sandwich Attack	Medium	Resolved
3	Tokens Can Be Stuck	Medium	Acknowledged
4	Constructor Parameter Validation	Medium	Resolved
5	Eth rewards might be stuck in contract	Medium	Acknowledged
6	Shares Are Not Updated During Burn	Medium	Resolved
7	APR Can Be Higher Than Intended	Medium	Resolved
8	Potential Unfair Liquidation Mechanic	Low	Resolved
9	Missing Zero Address Check	Low	Resolved
10	Not Using SafeERC20	Low	Acknowledged
11	Input Validation for Vesting Contract	Low	Resolved
12	Arithmetic operations	Informational	Resolved
13	Centralization Risk	Informational	Acknowledged
14	Multiplier Can Leave a Shortfall of Funds	Informational	Resolved
15	Unreasonable Tax For Buy and Sell	Informational	Acknowledged
16	Price Manipulation of `sell_threshold`	Informational	Resolved
17	Uniformed Variable Typing	Informational	Resolved
18	Unchecked Return Value of External Call	Informational	Resolved
19	Custom Errors	Informational	Acknowledged

#	Title	Risk	Status
20	Floating Pragma	Informational	Resolved
21	Empty Catch Statement	Informational	Acknowledged
22	Leftover Test Code	Informational	Acknowledged
23	Unknown Usage of State Variable	Informational	Resolved
24	Wrong value of precision	Informational	Resolved
25	Lack of events in the implementation of privileged functions	Informational	Acknowledged
26	Claimable Can Initially Return Zero	Informational	Acknowledged

HIGH-1 | RESOLVED

Vesting Contract Can Be Overridden

In the contract EsEMBR.sol, the function addVester() can cause catastrophic failure of users' funds. If a contract is added with the same timeframe, then all of the users funds being calculated in the previous contract will be permanently lost as there is no way to track the funds on the new Vester contract.

Recommendation:
It is recommended that there be a check when adding a Vester contract a validation check is done to ensure that a vesting contract cannot be overridden while assets are being vested.

Comment: The client fixed the issue in commit:
c1e1fba2e1958f6a41d9b573ec6133a684260892

MEDIUM-1 | RESOLVED

Possible Sandwich Attack

In the contract EmberVault.sol, the function liquidateToken() can be sandwich attacked due to having unlimited slippage and deadline.

```
// Wrap in a try catch to prevent owner from rugging by setting an invalid router, although
try IUniswapV2Router01(swap_router).swapExactTokensForETHSupportingFeeOnTransferTokens(
    contract_token_balance,
    0,
    path,
    address(this),
    type(uint).max
) { } catch {
    // Ignore
}
```

The code snippet above has zero and the max value of uint256 for slippage and deadline. This means that when liquidating a token, the token will take any amount against it. An MEV bot might pick this up and flashloan a trade against it to extract as much value as possible. This hurts the users of the protocol by extracting unnecessary value.

Recommendation:
Put a reasonable amount of slippage on the trade and do not allow for an infinite deadline.

Comment: Client resolved the issue in commit:
609d6bff264c180b21fabd6a75e7658f6cb885f6

MEDIUM-2 | ACKNOWLEDGED

Tokens Can Be Stuck

While calling *liquidateToken()*, the funds are sent from the token contract into the vault contract. When tokens are sent to be swapped at the router, the contract does not check if this is successful and optimistically assumes that it will work. If the operation somehow reverts, then the funds will not revert back to the token contract and will be stuck inside the Vault.

Recommendation:
Use the catch statement to ensure errors do not occur. If they do occur, then revert the transaction to ensure funds are not lost.

Comment from a client: It doesn't matter if the swap fails, the tokens are not used afterwards so it's fine if they're stuck. The subsequent calls do not use those tokens and the balance isn't used as well.

MEDIUM-3 | RESOLVED

Constructor Parameter Validation

In contract GenericERC20Token.sol, is highly recommended to perform a sanity check when setting the UniV2SwapRouter and UniV2Factory in this contract's constructor.

Recommendation:

Ensure that the addresses of UniV2SwapRouter and UniV2Factory are not set to zero (address(0)). A zero address can lead to contract functionality issues and unexpected behavior.

Comment: The client fixed the issue in commit: c1e1fba2e1958f6a41d9b573ec6133a684260892 by updating the vault contract to never allow a zero address to be passed.

MEDIUM-4 | ACKNOWLEDGED

Eth rewards might be stuck in contract

In EsEMBR.sol - Contract can receive ETH in a scenario in which no token is minted `(i.e. totalSupply = 0)`. ETH sent as a msg.value contributes to the state of `totalEthPerEsembr` which in turn affects the rewards to be sent out. In case of zero `totalSupply`, `totalEthPerEsembr` remains unaffected hence no rewards for that sent ETH is retrievable making it effectively stuck.

Recommendation:
function `receive()` reverts if there is no `totalSupply`. Or keep track of the `msg.value` being sent in that scenario so it might be retrievable.

MEDIUM-5 | RESOLVED

Shares Are Not Updated During Burn

EsEMBR.sol - During demo testing phase client discovered an issue on calling `vest(uint256 timeframe, uint256 amount)`. As this function burns the ERC20 of this contract which leads to a change of balance of the user calling it. Rewards are not updated after changing the balance this leads to user receiving rewards calculated based on old balance.

The impact of this finding though is mitigated by calling `claim()` before the vesting. Client resolved that issue once they discovered it by adding `_updateRevShareForUser(0, msg.sender);` before burning.

MEDIUM-6 | RESOLVED

APR Can Be Higher Than Intended

EsEMBR.sol - During testing phase client discovered an issue with the payOffWeeks function. The issue occurred with the APR calculation and would increase in certain scenarios such as paying off attempting to pay off multiple weeks but not having enough funds.

The client mediated the issue by adding the debt increase only if there was enough to pay the following week. The added code was added into the if statement `if(newPending >= nextWeeksDebt)..... newDebt = newDebt2.`

LOW-1 | RESOLVED

Potential Unfair Liquidation Mechanic

In the contract EmberVault.sol, as the *payup()* is less than *pending debt* the team's token can have the liquidity pulled and the token be set to *defaulted*. Because of the high APY, a token can be liquidated within the first week and not have enough time to grow. This might be an intended mechanic, but it should be made clear in the documentation that developers need to be aware of their payment otherwise their token will be immediately taken down.

Recommendation:
Clear documentation describing how the debt payment system works. Additionally, logic could be added to have different health scores for each token based on their risk score.

Comment: The client confirmed this was intended and will update documentation to assist users.

LOW-2 | RESOLVED

Missing Zero Address Check

In the contact GenericERC20Token.sol, the following functions are missing zero address checks that could cause undesired effects:

- *setRouter()*
- *setTaxReceiver()*
- *setExcludedFromLimits()*
- *setExcludedFromFees()*

Recommendation:
Add a custom error or require statement to ensure the zero address cannot be passed.

Comment: The client fixed the issue in commit:
c1e1fba2e1958f6a41d9b573ec6133a684260892

LOW-3 | ACKNOWLEDGED

Not Using SafeERC20

Contracts include several occurrences of unsafe transfer of ERC20 tokens. This might lead to undesirable results if the transfer is not successful while the transaction is.

Recommendation:
Utilize SafeERC20 for token transfers.

LOW-4 | RESOLVED

Input Validation for Vesting Contract

In the contract Vester.sol, there should be a default minimum time a contract will allow for vesting a token. When a user vests a token, the token is usually locked up for over one year. However in the current implementation, a vesting contract could be launched with zero timeframe and instantly unlock all tokens.

Recommendation:
A reasonable time frame should be enforced to ensure unintended timeframes are set during the deployment of the Vester contract.

Comment: The client fixed the issue in commit: c1e1fba2e1958f6a41d9b573ec6133a684260892 by requiring the time frame to be greater than zero.

INFORMATIONAL-1 | RESOLVED

Arithmetic operations

The SafeMath library is used for arithmetic operations in this contract. However, as of Solidity version 0.8 and later, SafeMath is no longer required for preventing arithmetic overflow/underflow issues. Solidity 0.8 introduced built-in overflow checks, making SafeMath redundant.

Recommendation:
It is recommended to remove the 'using SafeMath for uint256;' statement to simplify the code and reduce gas costs. You can rely on the built-in overflow checks provided by Solidity for arithmetic operations.

Centralization Risk

In the contract(s) GenericERC20Token.sol, EsEmber.sol, EsEMBRRewardsDistributor.sol, and EmberVault.sol, the owners of the contracts have excessive authority over the functionality of the smart contracts. The excessive authority granted to the admin poses a vulnerability in the contracts. The central issue is that the admin has unrestricted access to critical functions and general setters/mutators. The owner of the GenericERC20Token has potential excessive control over tokens deposited within the contract. If a malicious actor were to get control of the GenericERC20Token contract after it has paid off the debt, then excessive minting could occur. This concentration of power raises concerns, especially when certain functions carry higher risks than others. Depending on the project's objectives, it is advisable to implement governance mechanisms or utilize multisig wallets to distribute control and mitigate potential risks associated with a single wallet's authority.

The owner of the contracts have control over the following functions:

EsEMBRRewardsDistributor.sol:
setEmissionPerSecondEmbr() - Change the reward rate of Ember for the vault contract.
setEmissionPerSecondEth() - Change the reward rate of Ether for the vault contract.
setEsEMBR() - change the esEmber address.
EsEmber.sol:
addVester() - Add a new vesting contract with a new timeframe.
setRevShareSource() - toggle addresses ability to send ether to the contract.

GenericERC20Token:
addLiquidity() - Add liquidity once at the start of the contract. This is covered by the vault contract after the token is created.
mint() - Mint an unlimited amount of tokens after the debt is paid off up to the maximum supply set during deployment.
withdrawTokens() - Withdraw all tokens that were sent to the contract but only tokens created by the contract itself. Third party tokens cannot be saved.
withdrawEth() - Withdraw all the ether sent to the contract.
disableTransfers() - Set the status of the contract into default. This disables all transfers and limits functionality of the contract.
transferOwnershipToRealOwner() - Transfers the ownership of the contract after the debt is paid off.

setInitialLiquidityPool() - Sets the initial liquidity pool for the token.
disableMinting() - Permanently disables the ability to mint tokens and sets the max supply to equal the current supply.
setLP() - Sets the address of the LP address and toggles a boolean to enable or enable an address from being an LP.
setExcludedFromFees() - Removes the buy and sell fees from an address.
setExcludedFromLimits() - Removes the transaction limit from an address.
setTaxReceiver() - Sets the address that receives the buy and sell taxes from users.
setRouter() - Changes the router to an address the developer chooses.
setTaxes() - Changes the buy and sell taxes up to 25.2% for each. Buy tax can be 25.2% and the sell tax can be 25.2%.

Recommendation:
It is important that some functionality is locked behind a time lock or time lock implementation so users funds are not at stake and cannot be manipulated. Additionally adding multi-sig, governance, and/or a timelock. Lowering control over tokens deposited into the contract would lower risk severity.

INFORMATIONAL-2 | RESOLVED

Multiplier Can Leave a Shortfall of Funds

In the contract Vester.sol, a *multiplier* is set during deployment and is used to calculate the return value for a claimable amount. The local variable *claimable_amount* can equal the entire amount of the vested tokens. However, the calculation multiplying *claimable_amount* by the *multiplier* and dividing by `10000` could leave a shortfall in the contract if the multiplier is set egregiously higher than intended.

Recommendation:
Set a parameter or require statement limiting the size of the multiplier to ensure that a shortfall of Ember cannot occur.

Comment: The client fixed the issue in commit:
c1e1fba2e1958f6a41d9b573ec6133a684260892

INFORMATIONAL-4 | ACKNOWLEDGED

Unreasonable Tax For Buy and Sell

In the contract GenericERC20Token.sol, the constructor and function `setTaxes()` allows a user to set the buy and sell tax up to 25.2% for either tax. The buy and sell taxes should be set to a reasonable level where users will not lose a majority of their assets when buying and selling.

Recommendation:
Choose a reasonable threshold value to protect users assets.

INFORMATIONAL-5 | RESOLVED

Price Manipulation of `sell_threshold`

In the contract GenericERC20Token.sol, the `sell_threshold` can be gamified. Because `sell_threshold` is public data, bots can be set up to extract value from the pool when the threshold is about to be hit. The larger the threshold, the larger the gamification and potential swing in price might be.

For example: Price is trading at 1.25 USD and the liquidity is 200,000.

The sell_threshold is at 5,000 USD when this is sold against the liquidity it will push the price down in terms of Tokens/ETH. A bot can be set up to sell right before each threshold and gain more tokens over time. Eventually the user will dump a large amount of tokens on the supply.

Recommendation:
Choosing an optimal number to reduce gamification and extraction of value from users' funds.

INFORMATIONAL-6 | RESOLVED

Uniformed Variable Typing

Throughout the codebase many variables are initialized as uint256 and uint. Under the hood, all `uint` is initialized as uint256. However, to remain consistent throughout the codebase it is better practice to list all intended uint256 variables to be initialized as such.

Recommendation:
It is recommended to change all uint variables to uint256.

Comment: The client fixed the issue in commit:
c1e1fba2e1958f6a41d9b573ec6133a684260892

INFORMATIONAL-7 | RESOLVED

Unchecked Return Value of External Call

In the contract EsEMBR.sol, the function *vest()* does not handle the return value for an external contract call to the Vester contract.

Recommendation:
All external calls that return a value should be handled by default.

Comment: The client fixed the issue in commit:
c1e1fba2e1958f6a41d9b573ec6133a684260892

INFORMATIONAL-8 | ACKNOWLEDGED

Custom Errors

Throughout the codebase, `require` statements are used instead of custom errors. Custom errors are available from solidity version ^0.8.4. The contracts are on solidity version ^0.8.0 but are using a form of customized errors by importing `Errors.sol`. Custom errors can roughly save 50 gas per call. This gas is saved because of not having to allocate the string and store the string through the revert. Throughout the contracts in the codebase, `require` statements are used instead of custom errors. Custom errors save gas on deployment as well.

Recommendation:
Convert `require` statements to custom errors to save gas on each function call and deployment. If this is unnecessary, this finding will be deleted.

INFORMATIONAL-9 | RESOLVED

Floating Pragma

Throughout the codebase, the contracts that are unlocked at version ^0.8.19, and it should always be deployed with the same compiler version. By locking the pragma to a specific version, contracts are not accidentally getting deployed by using an outdated version that can introduce unintended consequences.

Comment: The client fixed the issue in commit:
c1e1fba2e1958f6a41d9b573ec6133a684260892

INFORMATIONAL-10 | ACKNOWLEDGED

Empty Catch Statement

The empty catch statement could handle errors from a fail swap and not revert silently. By using the catch users will be able to correctly identify why their transaction might have reverted.

Recommendation:
Use the catch statement to report errors that can happen during the external call during the swap.

Comment: Client resolved the issue in EmberVault.sol. There are occurrences of that issue in FreeERC20.sol and GenericERC20Token.sol which were not addressed.

INFORMATIONAL-11 | ACKNOWLEDGED

Leftover Test Code

In the contract EmberVault.sol at line 276 has leftover test code that is commented out. All testing code should be removed before being pushed to the production environment.

Recommendation:
It is recommended to remove any testing code before deploying the contracts.

Comment: Client partly resolved the issue in commit:
609d6bff264c180b21fabd6a75e7658f6cb885f6

INFORMATIONAL-12 | RESOLVED

Unknown Usage of State Variable

In the contract GenericERC20Token.sol the state variable *uint[] Extra* is not used within the contract itself. As the use case for this is unknown, it would be beneficial to understand what the intended use case for *Extra* is.

Recommendation:
Implement *uint[] Extra* or remove it from the contract.

Comment: The client fixed the issue in commit by removing it from the codebase: c1e1fba2e1958f6a41d9b573ec6133a684260892

INFORMATIONAL-13 | RESOLVED

Wrong value of precision

Vester.sol - `precision` value is set to
`uint256 public precision = 10e9`

But according to function `vest()`, the error message in the first line states that `precision` should be `1e9`.

```
// This function is only called by esEMBR contract. esEMBR also calls
claim for this user.
function vest(address user, uint256 amount) onlyEsEMBR external returns
(uint) {
    require(amount >= precision, "Vester: Amount cant be smaller than
1,000,000,000");
    ...
}
```

Recommendation:
Correction to precision value is needed.

Comment: Client resolved the issue in commit: 609d6bff264c180b21fabd6a75e7658f6cb885f6

Lack events in implementation of privileged functions

Some functions that carry out important changes to contracts state emit no events on changing significant parameters of the contracts by admin or other. This is taking place in a number of external state changing functions in the following files below.

In `EmberVault.sol`

function setRewardSettings(uint _base, uint _max, uint _rate) onlyOwner external
function setEsEMBR(address payable _esEmbr) onlyOwner external
function addPackage(Package calldata _package) external onlyOwner
function setPackageEnabled(uint package_id, uint8 _status) external onlyOwner
function unstakeEth(uint amount, address unstaker) onlyEsEMBR external
In EsEMBRRewardsDistributor.sol
function setEsEMBR(address payable _esEmbr) external onlyOwner
function setEmissionPerSecondEth(uint256 amount) external onlyOwner
function setEmissionPerSecondEmbr(uint256 amount) external onlyOwner

In GenericERC20Token.sol

function addLiquidity(uint token_amount) external payable onlyOwner returns(address)
function withdrawEth() external onlyOwner returns (uint)
function disableTransfers() external onlyOwner
function setInitialLiquidityPool(address _addy) public onlyOwner
function disableMinting() public onlyOwner
function setLP(address _lp, bool _bool) onlyOwner external
function setExcludedFromFees(address _address, bool _bool) onlyOwner external
function setExcludedFromLimits(address _address, bool _bool) onlyOwner external
function setTaxReceiver(address _tax_receiver) onlyOwner external
function setRouter(address router) onlyOwner external
function setTaxes(uint8 _buyTax, uint8 _sellTax) onlyOwner external

In EsEMBR.sol
function addVester(uint timeframe, IVester vester) onlyOwner external
function setRevShareSource(address _source, bool _enabled) onlyOwner external

Recommendation:
Emit relevant events to announce the changes.

Comment: The client updated some events to be emitted for *payup*, *stakeEmbr*, and *unstakeEmbr* in commit: c1e1fba2e1958f6a41d9b573ec6133a684260892.

INFORMATIONAL-15 | ACKNOWLEDGED

Claimable Can Initially Return Zero

In the contract EsEmbr.sol, the function *claimable* can cause confusion to external users. The intended functionality is to return the EsEmbr thats is available to be claimed. If a user has not called a function that changes the state of *mapping(address ⇒ uint256) claimableRewards*, then this value can return zero even though the true value is not zero.

Recommendation:
Simulate the return value of claim without changing state if *claimableRewards* is equal to zero to mitigate confusion of users as . However, the front end could solve this problem by creating a static call and showing the return value of *to_claim*. The downside to the alternative front end approach is that checking the value on tools such as Etherscan would still return zero.

	EmberVault.sol esEMBR.sol esEMBRRewardsDistributor.sol GenericERC20Token.sol Vester.sol
Re-entrancy	Pass
Access Management Hierarchy	Pass
Arithmetic Over/Under Flows	Pass
Unexpected Ether	Pass
Delegatecall	Pass
Default Public Visibility	Pass
Hidden Malicious Code	Pass
Entropy Illusion (Lack of Randomness)	Pass
External Contract Referencing	Pass
Short Address/ Parameter Attack	Pass
Unchecked CALL Return Values	Pass
Race Conditions / Front Running	Pass
General Denial Of Service (DOS)	Pass
Uninitialized Storage Pointers	Pass
Floating Points and Precision	Pass
Tx.Origin Authentication	Pass
Signatures Replay	Pass
Pool Asset Security (backdoors in the underlying ERC-20)	Pass

We are grateful for the opportunity to work with the Ember Labs team.

The statements made in this document should not be interpreted as an investment or legal advice, nor should its authors be held accountable for the decisions made based on them.

Zokyo Security recommends the Ember Labs team put in place a bug bounty program to encourage further analysis of the smart contract by third parties.

zokyo