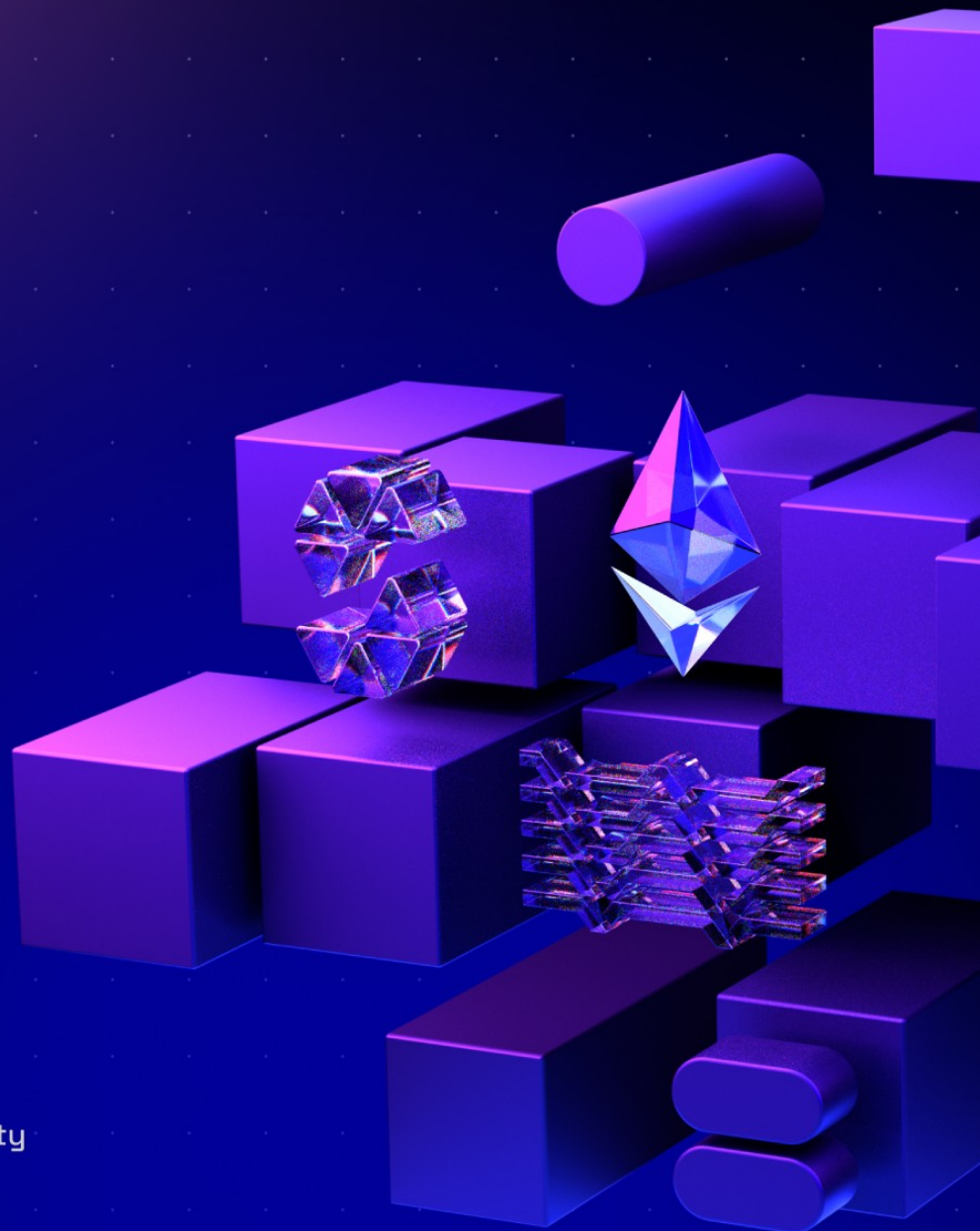


Cian

Yield Layer

14.1.2025



Contents

1. Document Revisions	3
2. Overview	4
2.1. Ackee Blockchain Security	4
2.2. Audit Methodology	5
2.3. Finding Classification	6
2.4. Review Team	8
2.5. Disclaimer	8
3. Executive Summary	9
Revision 1.0	9
Revision 1.1	10
4. Findings Summary	11
Report Revision 1.0	14
Revision Team	14
System Overview	14
Trust Model	14
Fuzzing	15
Findings	16
Report Revision 1.1	62
Revision Team	62
System Overview	62
Trust Model	62
Fuzzing	62
Appendix A: How to cite	63
Appendix B: Wake Findings	64
B.1. Fuzzing	64

1. Document Revisions

1.0-draft	Draft Report	11.12.2024
1.0	Final Report	16.12.2024
1.1	Fix Review	17.12.2024
1.1	Added client's responses	14.01.2025

2. Overview

This document presents our findings in reviewed contracts.

2.1. Ackee Blockchain Security

Ackee Blockchain Security is an in-house team of security researchers performing security audits focusing on manual code reviews with extensive fuzz testing for Ethereum and Solana. Ackee is trusted by top-tier organizations in web3, securing protocols including Lido, Safe, and Axelar.

We develop open-source security and developer tooling [Wake](#) for Ethereum and [Trident](#) for Solana, supported by grants from Coinbase and the Solana Foundation. Wake and Trident help auditors in the manual review process to discover hardly recognizable edge-case vulnerabilities.

Our team teaches about blockchain security at the Czech Technical University in Prague, led by our co-founder and CEO, Josef Gattermayer, Ph.D. As the official educational partners of the Solana Foundation, we run the [School of Solana](#) and the [Solana Auditors Bootcamp](#).

Ackee's mission is to build a stronger blockchain community by sharing our knowledge.

Ackee Blockchain a.s.

Rohanske nabrezi 717/4

186 00 Prague, Czech Republic

<https://ackee.xyz>

hello@ackee.xyz

2.2. Audit Methodology

1. Verification of technical specification

The audit scope is confirmed with the client, and auditors are onboarded to the project. Provided documentation is reviewed and compared to the audited system.

2. Tool-based analysis

A deep check with Solidity static analysis tool [Wake](#) in companion with [Solidity \(Wake\)](#) extension is performed, flagging potential vulnerabilities for further analysis early in the process.

3. Manual code review

Auditors manually check the code line by line, identifying vulnerabilities and code quality issues. The main focus is on recognizing potential edge cases and project-specific risks.

4. Local deployment and hacking

Contracts are deployed in a local [Wake](#) environment, where targeted attempts to exploit vulnerabilities are made. The contracts' resilience against various attack vectors is evaluated.

5. Unit and fuzz testing

Unit tests are run to verify expected system behavior. Additional unit or fuzz tests may be written using [Wake](#) framework if any coverage gaps are identified. The goal is to verify the system's stability under real-world conditions and ensure robustness against both expected and unexpected inputs.

2.3. Finding Classification

A *Severity* rating of each finding is determined as a synthesis of two sub-ratings: *Impact* and *Likelihood*. It ranges from *Informational* to *Critical*.

If we have found a scenario in which an issue is exploitable, it will be assigned an impact rating of *High*, *Medium*, or *Low*, based on the direness of the consequences it has on the system. If we haven't found a way, or the issue is only exploitable given a change in *configuration* (system settings or parameters, such as deployment scripts, compiler configurations, using multi-signature wallets for owners, etc.) or given a change in the codebase, then it will be assigned an impact rating of *Warning* or *Info*.

Low to *High* impact issues also have a *Likelihood*, which measures the probability of exploitability during runtime.

The full definitions are as follows:

Severity

		<i>Likelihood</i>			
		High	Medium	Low	N/A
<i>Impact</i>	High	Critical	High	Medium	-
	Medium	High	Medium	Low	-
	Low	Medium	Low	Low	-
	Warning	-	-	-	Warning
	Info	-	-	-	Info

Table 1. Severity of findings

Impact

- **High** - Code that activates the issue will lead to undefined or catastrophic consequences for the system.
- **Medium** - Code that activates the issue will result in consequences of serious substance.
- **Low** - Code that activates the issue will have outcomes on the system that are either recoverable or don't jeopardize its regular functioning.
- **Warning** - The issue cannot be exploited given the current code and/or *configuration*, but could be a security vulnerability if these were to change slightly. If we haven't found a way to exploit the issue given the time constraints, it might be marked as a "Warning" or higher, based on our best estimate of whether it is currently exploitable.
- **Info** - The issue is on the borderline between code quality and security. Examples include insufficient logging for critical operations. Another example is that the issue would be security-related if code or *configuration* was to change.

Likelihood

- **High** - The issue is exploitable by virtually anyone under virtually any circumstance.
- **Medium** - Exploiting the issue currently requires non-trivial preconditions.
- **Low** - Exploiting the issue requires strict preconditions.

2.4. Review Team

The following table lists all contributors to this report. For authors of the specific revision, see the “Revision team” section in the respective “Report revision” chapter.

Member's Name	Position
Jan Kalivoda	Lead Auditor
Jan Převrátíl	Auditor
Martin Veselý	Auditor
Josef Gattermayer, Ph.D.	Audit Supervisor

2.5. Disclaimer

We've put our best effort to find all vulnerabilities in the system, however our findings shouldn't be considered as a complete list of all existing issues. The statements made in this document should not be interpreted as investment or legal advice, nor should its authors be held accountable for decisions made based on them.

3. Executive Summary

Cian is a protocol that allows to earn yield on deposited assets.

Revision 1.0

Cian engaged Ackee Blockchain Security to perform a security review of the Cian protocol with a total time donation of 15 engineering days, including 4 days dedicated to fuzzing, in a period between November 27 and December 11, 2024, with Jan Kalivoda as the lead auditor.

The audit was performed on the commit [54e953^{\[1\]}](#) and the scope included all contracts, excluding strategies.

We began our review using static analysis tools, including [Wake](#). We then took a deep dive into the logic of the contracts. For testing and fuzzing, we utilized the [Wake](#) testing framework. For more information about the fuzzing, see [Fuzzing](#) section. During the review, we paid special attention to:

- ensuring the arithmetic operations and accounting of the system were correct;
- detecting possible reentrancies and unprotected calls in the code;
- ensuring access controls are not too relaxed or too strict;
- looking for common issues such as data validation.

Our review resulted in 26 findings, ranging from Info to Medium severity. The protocol demonstrated centralization, making its correct functioning highly dependent on the protocol owners (see [M3](#)). Additionally, we identified arithmetic and data validation issues that could lead to incorrect protocol accounting (see [M1](#)).

Ackee Blockchain Security recommends Cian:

- write a documentation;
- create a comprehensive test suite;
- focus on intermediary divisions and precision during the calculations;
- address all other reported issues.

See [Report Revision 1.0](#) for the system overview and trust model.

Revision 1.1

The fix review was performed on the given commit `06f333`^[2]. The scope was the fixes of the findings from the previous revision. All issues were addressed and were either fixed, partially fixed, or acknowledged with comments explaining the reasons.

See [Findings Summary](#) for updated statuses of the findings and [Report Revision 1.1](#) for more information about the new revision.

[1] full commit hash: `54e9538f04743db9e1d996baf8d8099daf34d98d`

[2] full commit hash: `06f3331d332c8ffee67e487f8ec17baf39d13849`

4. Findings Summary

The following section summarizes findings we identified during our review. Unless overridden for purposes of readability, each finding contains:

- *Description*
- *Exploit scenario* (if severity is low or higher)
- *Recommendation*
- *Fix* (if applicable).

Summary of findings:

Critical	High	Medium	Low	Warning	Info	Total
0	0	3	4	9	10	26

Table 2. Findings Count by Severity

Findings in detail:

Finding title	Severity	Reported	Status
M1: Invalid calculations due to intermediary division	Medium	1.0	Partially fixed
M2: Pool state variables have insufficient data validation	Medium	1.0	Fixed
M3: Users have almost no control over their deposited funds	Medium	1.0	Acknowledged
L1: Double-entryptoint initialize functions	Low	1.0	Fixed

Finding title	Severity	Reported	Status
L2: Using <code>transfer</code> instead of <code>call</code>	Low	1.0	Acknowledged
L3: Missing initializers on constructors	Low	1.0	Acknowledged
L4: Strategy Position Limit Calculation Inaccuracy	Low	1.0	Acknowledged
W1: Strict equality check for balances	Warning	1.0	Acknowledged
W2: Potential depeg of ETH-based assets	Warning	1.0	Acknowledged
W3: Vault is not ERC4626 compliant	Warning	1.0	Acknowledged
W4: Protocol owner can set arbitrary exchange price to pools	Warning	1.0	Partially fixed
W5: Pitfalls of the Ownable contract	Warning	1.0	Acknowledged
W6: Protocol owner can artificially mint Vault shares	Warning	1.0	Partially fixed
W7: Underflow can cause DoS in <code>confirmWithdrawal</code>	Warning	1.0	Acknowledged
W8: Users are not able to request more than one withdrawal	Warning	1.0	Acknowledged
W9: Potential issues with retrieving borrow and supply caps	Warning	1.0	Partially fixed

Finding title	Severity	Reported	Status
I1: The function can be declared as a view function	Info	1.0	Acknowledged
I2: Missing documentation	Info	1.0	Fixed
I3: Typos and incorrect NatSpec comments	Info	1.0	Fixed
I4: Missing underscore in internal function's name	Info	1.0	Acknowledged
I5: Modifier consistency on access controls	Info	1.0	Acknowledged
I6: Unused variable	Info	1.0	Fixed
I7: Unused using-for directives	Info	1.0	Fixed
I8: Unused imports	Info	1.0	Fixed
I9: Unused events	Info	1.0	Fixed
I10: Unchecked return value for OFT receipt	Info	1.0	Fixed

Table 3. Table of Findings

Report Revision 1.0

Revision Team

Member's Name	Position
Jan Kalivoda	Lead Auditor
Jan Převrátíl	Auditor
Martin Veselý	Auditor
Josef Gattermayer, Ph.D.	Audit Supervisor

Sytem Overview

The protocol allows users to deposit assets into the Vault contract and earn yield on them. The yield is generated by multiple strategies. Proxies for these strategies are deployed from the Vault contract. Users cannot directly withdraw funds from the protocol but can request a withdrawal from the Vault, which registers the request in the RedeemOperator. The RedeemOperator can then process the withdrawal requests. The protocol also features cross-chain interoperability with pools on other chains that can hold a representation of the Vault token, which can be exchanged for allowed deposit tokens on the given chain.

Trust Model

The users have to trust protocol owners to act honestly since, once they deposit funds into the Vault, they cannot withdraw without external confirmation. The withdrawn amount is also decided by the entity that confirms the withdrawal requests. Several findings were raised related to the trust model (see [M3](#), [W6](#), [W4](#)).

Client's response

As an advanced yield strategy protocol, its nature determines strategy parameters need to be updated based on various market conditions. The restriction of Solidity based smart contract, the immaturity of DeFi market, and the inaccessibility to some on-chain data in a decentralized way make it impractical to rely purely on smart contracts for parameter updates. As an example, the protocol allocates users' funds across multiple decentralized yield strategies for yield optimization. The allocation percentage is coded in the smart contract. The adjustment of the allocation scheme cannot and should not be automated by the smart contract and requires multi-sig that reflects community's consensus.

— Cian Team

Fuzzing

A manually-guided differential stateful fuzz test was developed during the review to test the correctness and robustness of the system. The fuzz test employs fork testing technique for tokens such as stETH, wstETH, and WETH, to test the system with external contracts exactly as they are deployed in the deployment environment. The cross-chain architecture was needed to be mocked and to these external components we were acting as to a black box. Also tokens on the other chains were mocked from the performance reasons, to allow execute several flows over the mainnet deployment while still performing cross-chain operations. Finally, strategies for the Vault were out of scope so we developed a mock for the strategy interface and yield was simulated with an artificial minting.

The differential fuzz test keeps its own Python state according to the system's specification. Assertions are used to verify the Python state against

the on-chain state in contracts.

The list of all implemented execution flows and invariants is available in [Appendix B](#).

Findings

The following section presents the list of findings discovered in this revision.

M1: Invalid calculations due to intermediary division

Medium severity issue

Impact:	High	Likelihood:	Low
Target:	RedeemOperator.sol	Type:	Arithmetics

Description

Two `mulDiv` functions are used to calculate the `cutPercentage_` value.

Listing 1. Excerpt from RedeemOperator

```
186 if (exchangePrice_ < lastExchangePrice) {
187     uint256 diff_ = (lastExchangePrice - exchangePrice_).mulDiv(
188         (IERC20(vault).totalSupply() - totalShares_), PRECISION,
189         Math.Rounding.Ceil
190     );
191     cutPercentage_ = diff_.mulDiv(PRECISION * PRECISION, totalShares_ *
192     exchangePrice_, Math.Rounding.Ceil);
191 }
```

Splitting the calculation into two `mulDiv` functions leads to invalid calculations and loss of precision.

The two formulas used for the calculation can be reduced to one formula using a single `mulDiv` function, thus avoiding intermediary division.

While both implementations work similarly with high decimal place numbers, precision is lost when handling smaller numbers.

Assuming the following example of Solidity code:

```
function test(uint256 totalShares, uint256 withdrawShares, uint256
exchangePrice, uint256 lastExchangePrice) public pure returns (uint256) {
    require(exchangePrice < lastExchangePrice, "exchangePrice must be less than
```

```

lastExchangePrice");

    uint256 diff_ = (lastExchangePrice - exchangePrice).mulDiv(
        (totalShares - withdrawShares), PRECISION, Math.Rounding.Ceil
    );
    uint256 cutPercentage_ = diff_.mulDiv(PRECISION * PRECISION, withdrawShares
* exchangePrice, Math.Rounding.Ceil);
    return cutPercentage_;
}

function test2(uint256 totalShares, uint256 withdrawShares, uint256
exchangePrice, uint256 lastExchangePrice) public pure returns (uint256) {
    return PRECISION.mulDiv(((lastExchangePrice - exchangePrice)*(totalShares-
withdrawShares)),
        (withdrawShares*exchangePrice), Math.Rounding.Ceil);
}

```

and the Python test file for it:

```

from wake.testing import *
from pytypes.tests.mocks.Sandbox import Sandbox

@chain.connect()
def test_sandbox():
    sandbox = Sandbox.deploy()
    print(f"Two mulDivs: {sandbox.test(50,10,30,40)}")
    print(f"One mulDiv: {sandbox.test2(50,10,30,40)}")

```

The execution produces the following output:

```

Two mulDivs: 33333333333333333333333333333334
One mulDiv: 13333333333333333334

```

Increasing the input parameters for the functions makes the difference between the two results smaller.

```

sandbox.test(50_000_000,10_000_000,30_000_000,40_000_000)
sandbox.test2(50_000_000,10_000_000,30_000_000,40_000_000)

```

```
Two mulDivs: 33333333333333333334
One mulDiv: 133333333333333334
```

Increasing the input parameters by another 3 decimals, both functions produce the same result.

```
Two mulDivs: 1333333333333333334
One mulDiv: 133333333333333334
```

Two `mulDiv` functions are used a second time in the same function in this part of the code:

Listing 2. Excerpt from RedeemOperator

```
196 uint256 assetPerShare_ = tokenBalanceGet_.mulDiv(PRECISION, totalShares_,
    Math.Rounding.Floor);
197
198 address thisUser_;
199 uint256 thisUserGet_;
200 uint256 gasPerUser_ = _totalGasLimit * tx.gasprice / _users.length;
201 uint256[] memory amounts_ = new uint256[](_users.length);
202 for (uint256 i = 0; i < _users.length; ++i) {
203     thisUser_ = _users[i];
204     thisUserGet_ = _withdrawalRequest[thisUser_].mulDiv(assetPerShare_,
        PRECISION, Math.Rounding.Floor);
```

The `assetPerShare_` value is calculated using a `mulDiv` function call. Then, for each user, the `thisUserGet_` value is calculated using the `assetPerShare_` value. This calculation uses the second `mulDiv` function. The reduced formula was checked with differential fuzzing, and it was discovered that using two `mulDiv` functions causes a loss of precision with more decimal places.

Exploit scenario

The `confirmWithdrawal` function is called, and the `cutPercentage_` variable loses precision during calculation. As a result, Alice, who has a pending withdrawal

request, receives fewer assets than expected.

Recommendation

Avoid using division before multiplication. Simplify the equations and perform one `mulDiv` operation when possible to avoid precision loss or miscalculations.

Partial solution 1.1

The intermediate division is removed for the `cutPercentage_` calculation. However, the `thisUserGet_` calculation still uses two `mulDiv` functions.

[Go back to Findings Summary](#)

M2: Pool state variables have insufficient data validation

Medium severity issue

Impact:	High	Likelihood:	Low
Target:	Pool.sol, PoolArb.sol, PoolOp.sol	Type:	Data validation

Description

The initialize function and setter functions for the pool contracts have insufficient data validation. While the `VaultYieldBasic` contract has checks for zero values and allowed ranges for state variables, the pool contracts do not have any checks for their state variables. As a result, fees can be set unreasonably high, and the exchange price can be set to zero, which would break the logic due to divisions by zero.

Listing 3. Excerpt from Pool

```
146 shares_ = _amount * PRECISION / getStorage().exchangePrice;
```

All passed addresses should be checked for the zero address. For withdrawal and deposit fees, there should be checks for the allowed range.

Exploit scenario

The incorrect deposit fee value is passed, as a result, a user will receive on deposit less assets than expected.

Recommendation

Add data validation for pool contracts.

Fix 1.1

Data validation has been added to the pool contracts. The exchange price is now restricted to not allow more than 5% change per day (cumulatively). During initialization, fees can be set arbitrarily; however, setter functions are now restricted with a maximum fee value, thus mitigating the high impact of this finding.

[Go back to Findings Summary](#)

M3: Users have almost no control over their deposited funds

Medium severity issue

Impact:	High	Likelihood:	Low
Target:	*	Type:	Trust model

Description

There are no guarantees for users to claim their funds back when they are once deposited into the Vault contract. Users can only request to redeem their funds, but they can't know which amount they will receive back because there are several mechanisms that can potentially reduce the amount of funds they will receive back. The most important during withdrawals is the `cutPercentage_` variable that is calculated during calling the withdrawal process.

The privileged account is calling the `updateExchangePrice` to update the exchange ratio between shares and assets. When the exchange price is up-to-date then there is no cut percentage.

Listing 4. Excerpt from RedeemOperator

```
175 function confirmWithdrawal(address[] memory _users, uint256 _totalGasLimit)
    external onlyOperator {
176     uint256 totalShares_;
177     for (uint256 i = 0; i < _users.length; ++i) {
178         if (!_pendingWithdrawers.contains(_users[i])) revert
            Errors.InvalidWithdrawalUser();
179         totalShares_ += _withdrawalRequest[_users[i]];
180     }
181     uint256 exchangePrice_ = IVault(vault).exchangePrice();
182     uint256 lastExchangePrice = IVault(vault).lastExchangePrice();
183     if (lastExchangePrice == 0) revert Errors.UnSupportedOperation();
184
185     uint256 cutPercentage_;
```

```

186     if (exchangePrice_ < lastExchangePrice) {
187         uint256 diff_ = (lastExchangePrice - exchangePrice_).mulDiv(
188             (IERC20(vault).totalSupply() - totalShares_), PRECISION,
189             Math.Rounding.Ceil
190             );
191         cutPercentage_ = diff_.mulDiv(PRECISION * PRECISION, totalShares_ *
            exchangePrice_, Math.Rounding.Ceil);
191     }

```

Essentially, it can be up-to-date all the time, if it will be called twice before the withdrawal and not only once. When it is called once and there is a bigger difference between the last exchange price and the current price, then the cut percentage will be higher. The cut percentage can be as high as 100% and even more to cause an underflow error and revert (which is better than the execution on high cut percentage for the users).

When the cut percentage is close to 100% ($10^{18} == \text{PRECISION}$), then from the following snippet can be seen that amount which is redeemed to be distributed to the users can be significantly reduced or even zeroed.

Listing 5. Excerpt from VaultYieldBasic

```

495 uint256 assets_ = previewRedeem(_shares * (PRECISION - _cutPercentage) /
    PRECISION);
496 _burn(_owner, _shares);

```

Then from the redeemed amount (which can be zero) is calculated asset per share and it is evenly distributed to the users based on their shares to redeem.

Exploit scenario

The `confirmWithdrawal` function is called with a high cut percentage. As a result, Alice, who has an active withdrawal request, receives fewer assets than expected.

Recommendation

Reconsider the design decision of the withdrawal process or at least **implement checks directly in to the code to avoid such huge losses.**

Transaction should be reverted if the losses are too high. Also inform adequately the users about the all subtracted amounts during withdrawals and risks.

Acknowledgment 1.1

To optimize the yield, the protocol

- 1. allocates users' funds across multiple decentralized yield strategies;*
- 2. updates the strategy parameters through a multi-sig controlled by CIAN and other related protocols collectively;*
- 3. determines the exit strategy based on market conditions.*

The domain expertise and operational complexities involved in advanced DeFi yield strategies decide that there are a number of operations that the protocol has to execute on users' behalf.

— Cian Team

[Go back to Findings Summary](#)

L1: Double-entripoint initialize functions

Low severity issue

Impact:	Medium	Likelihood:	Low
Target:	LzBridgeArb.sol, LzBridgeOp.sol	Type:	Logic error

Description

The `LzBridgeArb` (and `LzBridgeOp`, respectively) contract has two initialize functions. The first one is defined correctly in `LzBridgeArb.sol` in the contract itself. However, the contract also inherits the second initialize function from the `LayerZeroBridgeHelper` contract. As a result, it is possible to initialize the `LzBridgeArb` (and `LzBridgeOp`, respectively) contract with both functions.

Initialization through the inherited function results in an unset Arbitrum Outbox parameter.

Listing 6. Excerpt from LzBridgeArb

```
22 function initialize(bytes calldata _initBytes) public initializer {
23     (address admin_, address operator_, address vault_, address
      mintAuthority_, address oftWrapper_, address outbox_) =
      abi.decode(_initBytes, (address, address, address, address, address,
      address));
24     __BridgeHelper_init(admin_, vault_, mintAuthority_, operator_, new
      address[](0));
25     __LayerZeroBridgeHelperInit(oftWrapper_);
26     getArbBridgeStorage().outbox = outbox_;
27 }
```

Listing 7. Excerpt from LayerZeroBridgeHelper

```
26 function initialize(address _oftWrapper, address _owner, address _vault,
      address _mintAuthority, address _operator) public virtual initializer {
27     __BridgeHelper_init(_owner, _vault, _mintAuthority, _operator, new
      address[](0));
```

```
28     __LayerZeroBridgeHelperInit(_oftWrapper);  
29 }
```

Exploit scenario

Alice, the deployer, deploys the contract and initializes it using the inherited initialization function. The contract becomes partially initialized with an unset Arbitrum Outbox parameter, requiring contract redeployment for full functionality.

Recommendation

Allow only the one correct initialization function to be used.

Fix 1.1

The `initialize` function from the `LayerZeroBridgeHelper` contract is removed.

[Go back to Findings Summary](#)

L2: Using `transfer` instead of `call`

Low severity issue

Impact:	Low	Likelihood:	Low
Target:	Pool.sol, BridgeHelper.sol	Type:	Data validation

Description

The `transfer` function in Solidity imposes a 2,300 gas limit on the recipient contract. This limitation creates potential transaction failures when the recipient contract requires more gas for execution. The `call` function provides a more reliable alternative by allowing flexible gas limits and enabling proper return value validation.

Exploit scenario

Alice calls the `transferToAuthority` or `recoverETH` function to send ETH to a recipient smart contract. The recipient contract requires more than 2,300 gas units to process the incoming ETH transfer. Due to the `transfer` function's gas limitation, the transaction fails, preventing the intended ETH transfer.

Recommendation

Replace the `transfer` function with `call` in the `Pool` and `BridgeHelper` contracts.

[Go back to Findings Summary](#)

L3: Missing initializers on constructors

Low severity issue

Impact:	Low	Likelihood:	Low
Target:	*	Type:	Front-running

Description

All upgradeable contracts are missing the `initializer` modifier or other mechanisms to prevent undesired initialization. One scenario occurs when the proxy of the logic contract is not created atomically with the `initialize` function call of the logic contract. In this case, an attacker can front-run the transaction to gain ownership of the proxy after its creation, forcing the deployer to redeploy the proxy.

Another scenario allows any user to call the `initialize` function on the logic contract and gain ownership of it. While no direct threat was identified from this case, there is no reason to allow unrestricted access to this functionality.

Exploit scenario

Alice observes the new contract deployment and front-runs the upcoming initialization transaction by calling the `initialize` function first, gaining unauthorized ownership of the contract.

Recommendation

Always deploy proxies atomically by calling the `initialize` function with the deployment and use the `initializer` modifier (or other mechanism) on the logic contract's constructor to prevent undesired initialization.

Acknowledgment 1.1

*We're deploying proxy using OpenZeppelin's V5
TransparentUpgradeableProxy, which ensure the contract
creation and initialization is done in the same tx.*

— Cian Team

[Go back to Findings Summary](#)

L4: Strategy Position Limit Calculation Inaccuracy

Low severity issue

Impact:	Low	Likelihood:	Low
Target:	VaultYieldBasic	Type:	Overflow/Underflow

Description

The `VaultYieldBasic.transferToStrategy` function may incorrectly trigger an `Errors.InvalidLimit` error when the strategy has not reached its specified limit. This occurs due to the usage of the `totalAssets()` function on Line 326:

Listing 8. Excerpt from VaultYieldBasic

```
326 if ((nowAssets_ + transferAsset_) > (totalAssets() * positionLimit_ / 1e4))
    revert Errors.InvalidLimit();
```

The `totalAssets()` function is implemented as follows:

Listing 9. Excerpt from VaultYieldBasic

```
381 /**
382  * @dev Retrieve the amount of assets in the strategy pool.
383  * @return The total assets in the strategy pool.
384  */
385 function totalAssets() public view override returns (uint256) {
386     if (block.timestamp - vaultState.lastUpdatePriceTime >
387         vaultParams.maxPriceUpdatePeriod) {
387         revert Errors.PriceNotUpdated();
388     }
389
390     return vaultState.exchangePrice * totalSupply() / PRECISION;
391 }
```

When the price remains fresh but `exchangePrice` increases due to an increase

of underlying assets, the transfer fails if the amount reaches exactly the `positionLimit_`. Since the `totalAssets()` function uses the `exchangePrice` to calculate the total value of assets, the increase in assets causes the difference between `totalAssets()` and `underlyingTvl()` to be greater than 0. This difference causes the transfer reaching exactly the limit to fail even if the strategy has not reached its specified limit.

Conversely, when the price decreases while remaining fresh, the transfer may exceed the specified limit by the same calculation.

Exploit scenario

Alice, the Operator, manages a vault with the following configuration:

1. The vault contains Strategy A, which has a 40% limit, and Strategy B, which has a 60% limit.
2. Strategy A holds 390 ULT (underlying token), Strategy B holds 595 ULT, and the vault contains 14.7 ULT.
3. The price of 1 share equals 1.0 ULT, resulting in a total value of 999.7 for both assets and shares.
4. Due to yield generation, the balance of strategy A increases to 390.1 ULT, and the balance of strategy B to 595.2 ULT.
5. The vault's total assets reach 1000 ULT while maintaining 999.7 shares. The new price of 1 share would be ~1.0003 ULT, but the price is not yet updated since the old price is still valid.
6. Alice attempts to transfer 9.9 ULT to Strategy A, which should succeed as it would result in 400 ULT, within the 40% limit of 1000 ULT.
7. The transfer of 9.9 ULT to strategy A will fail with an `InvalidLimit` error. That is because `totalAssets()` will return 999.7 ULT (999.7 shares * 1.0 price), from which 40% are 399.88 ULT, making the 400 ULT over the limit.

Recommendation

Ensure there is always a small reserve within the limit when transferring assets to a strategy, or set the `position limit` during strategy creation to a slightly higher value than the intended limit.

Acknowledgment 1.1

The strategy's position limit actually works as a cap for strategy's deposit, and it's usually not used up to cap, so we decided to not change this.

— Cian Team

[Go back to Findings Summary](#)

W1: Strict equality check for balances

Impact:	Warning	Likelihood:	N/A
Target:	OneInchCallerV6.sol	Type:	Denial of service

Description

The `OneInchCallerV6` contract uses a strict equality check after swapping for ETH balance. This can cause issues when the balance is not exactly as expected. For example, when the contract receives additional ETH during the call, the transaction will revert.

Listing 10. Excerpt from OneInchCallerV6

```
108 uint256 ethBal_ = address(this).balance;
109 returnData_ = Address.functionCallWithValue(ONEINCH_ROUTER, _swapData,
    _amount);
110 spentAmount_ = ethBal_ - address(this).balance;
111 returnAmount_ = IERC20(_dstToken).balanceOf(address(this)) - tokenBefore_;
112 if (spentAmount_ != _amount) revert Errors.OneInchUnexpectedSpentAmount();
```

Recommendation

Revert the transaction when the value of `spentAmount` is bigger than `amount`.

Acknowledgment 1.1

It's unlikely under our usage condition.

— Cian Team

[Go back to Findings Summary](#)

W2: Potential depeg of ETH-based assets

Impact:	Warning	Likelihood:	N/A
Target:	*	Type:	N/A

Description

While the current implementation is using ETH, WETH, and stETH, it can be considered reasonably safe against depeg. However, once tokens are deposited into pool, the ratio between them is not maintained. Thus, if some specific ETH-based asset will be used within the Vault and gets affected by a significant price deviation, then it can harm the protocol.

Recommendation

Be aware of potential depegging when implementing other ETH-based assets in the future, and adjust the logic as needed.

Acknowledgment 1.1

The price calculation of this vault is based on steth, the eth deposit function is only for user's convience, and the only asset allowed to withdraw is steth.

— Cian Team

[Go back to Findings Summary](#)

W3: Vault is not ERC4626 compliant

Impact:	Warning	Likelihood:	N/A
Target:	VaultYieldBasic.sol	Type:	Standards violation

Description

The vault is not fully compliant with the ERC-4626 standard. For example, the `previewWithdraw` and `previewRedeem` functions are not fee-inclusive, which can potentially confuse users. Moreover, the `withdraw` and `redeem` functions are not available.

Recommendation

Add proper documentation for these deviations from the standard to inform the protocol users.

Acknowledgment 1.1

We'll notice user on the deposit page and docs of these contracts.

— Cian Team

[Go back to Findings Summary](#)

W4: Protocol owner can set arbitrary exchange price to pools

Impact:	Warning	Likelihood:	N/A
Target:	*	Type:	Trust model

Description

The exchange price of the pools is set by the pool owner. This can be either abused by the pool owner or provide incorrect amounts for exchanges, allowing someone to exploit it.

Recommendation

Reconsider the design of exchange price settlement in the pools to prevent misuse. Ensure that the exchange price is updated promptly to prevent unwanted exploitation when withdrawals are allowed.

Partial solution 1.1

Together with the [M2](#) issue, this finding is partially remediated by the introduction of restrictions for values for the exchange price. These values are now checked to not allow more than 5% change per day (cumulatively). If price is not changed for more days, it means that the price can be changed at maximum: current price * days without change * 5%.

Client's response

CIAN's yield layer is a chain-agnostic solution that allows users to deposit funds into the same yield layer vault from all supported networks and enjoy the same yield. Out of security consideration, it was decided that the receipt token is only minted on Ethereum mainnet and so is its bookkeeping. Therefore, when the yield layer smart contract on another

network receives a user's deposits and needs to decide the amount of receipt tokens this user should receive, it has to refer to the "real-time" price of the receipt tokens maintained by the smart contract on the Ethereum mainnet. This process can only be decentralized when a high-frequency oracle on the destination network supports this receipt token. However, it's impractical for all receipt tokens of increasing number of yield layers to be supported by Oracle service providers. Therefore, we have to feed the "real-time" price of the receipt tokens in a centralized way to other networks to guarantee the accurate amount of receipt tokens users should receive.

— Cian Team

[Go back to Findings Summary](#)

W5: Pitfalls of the Ownable contract

Impact:	Warning	Likelihood:	N/A
Target:	*	Type:	Data validation

Description

The contracts in the codebase use the Ownable contract. The Ownable contract is a simple contract that allows the owner to transfer ownership of the contract to a new address. This can be done accidentally or intentionally to an invalid address, thus causing the contract to lose access to the owner's functions.

Additionally, the Ownable contract has the `renounceOwnership` function that can be used to renounce ownership of the contract. When this function is called, the owner's functions become unavailable.

In cases where the owner is always needed for the contract to function, it is recommended to use the `Ownable2Step` contract instead and override the `renounceOwnership` function to revert the transaction. This helps mitigate such undesired behavior.

Recommendation

Use the `Ownable2Step` (or `Ownable2StepUpgradeable`) contract instead of the `Ownable` contract, and override the `renounceOwnership` function to revert the transaction.

Acknowledgment 1.1

We have check before exec on these sensitive operations.

— Cian Team

[Go back to Findings Summary](#)

W6: Protocol owner can artificially mint Vault shares

Impact:	Warning	Likelihood:	N/A
Target:	*	Type:	Trust model

Description

The Vault contract has the ability to artificially mint shares to the mint authority, which is governed by the protocol owner. This functionality is used for providing liquidity for other chains. However, there is no mechanism to prevent the protocol owner from minting shares to themselves and withdrawing underlying assets.

Recommendation

Reconsider the design of providing liquidity to other chains to prevent the possibility of misuse.

Partial solution 1.1

Caps for minting unbacked shares per receiver are introduced. Since minting is performed by the operator, it is a partial remediation of the issue. However, artificial minting is still possible under certain circumstances.

Client's response

The above statement is inaccurate. There are 2 scenarios where vault shares can be minted.

In the main "yield layer vault" where users deposit funds, receipt tokens/ vault shares are generated automatically based on users' deposit. The smart contract owner has no discretion over this process. It's coded in the following

method.

— Cian Team

```
function deposit(uint256 _assets, address _receiver)
    public
    override
    nonReentrant
    whenNotPaused
    returns (uint256 shares_)
{
    if (_assets == type(uint256).max) {
        _assets = IERC20(asset()).balanceOf(msg.sender);
    }
    shares_ = super.deposit(_assets, _receiver);
}
```

CIAN's yield layer is a chain-agnostic solution that allows users to deposit funds into the same yield layer vault from all supported networks and enjoy the same yield. Out of security consideration, it was decided that the receipt token minting is only allowed on Ethereum mainnet, not on any other networks. To allow users to deposit funds on other supported networks and receive receipt tokens instantly, the approach is to pre-mint limited amount of receipt tokens on the ethereum mainnet through a "interoperability abstraction" module and bridge the pre-minted receipt tokens to other networks beforehand. It is within this module that the multi-sig owner is allowed to pre-mint receipt tokens. The method is shown below.

```
function mintUnbacked(uint256 _amount) external onlyUnbackedMinter {
    unbackedMintedAmount += _amount;
    _mint(unbackedMinter, _amount);
}
```

```
}
```

— Cian Team

[Go back to Findings Summary](#)

W7: Underflow can cause DoS in `confirmWithdrawal`

Impact:	Warning	Likelihood:	N/A
Target:	RedeemOperator.sol	Type:	Overflow/Underflow

Description

The `confirmWithdrawal` function in the `RedeemOperator` contract can cause DoS for valid users who want to withdraw their funds. When users want to withdraw their funds, they must request a withdrawal by calling `requestRedeem`, where they set how much funds they want to withdraw, and then owner must confirm it in the `confirmWithdrawal` function. The potential issue from the `confirmWithdrawal` function is in this following code snippet:

Listing 11. Excerpt from RedeemOperator

```
200 uint256 gasPerUser_ = _totalGasLimit * tx.gasprice / _users.length;
201 uint256[] memory amounts_ = new uint256[](_users.length);
202 for (uint256 i = 0; i < _users.length; ++i) {
203     thisUser_ = _users[i];
204     thisUserGet_ = _withdrawalRequest[thisUser_].mulDiv(assetPerShare_,
        PRECISION, Math.Rounding.Floor);
205     // If the user's share is not enough to cover the gas, it will fail.
206     thisUserGet_ -= gasPerUser_;
```

Firstly the `gasPerUser_` value is computed, it is the average gas cost for the user. Then there is computed the `thisUserGet_` value which stands for the amount of funds the user requested to withdraw. The value `thisUserGet_` is then reduced by the `gasPerUser_` value.

This mathematical operation can cause an underflow if the `gasPerUser_` value is higher than the `thisUserGet_` value.

Exploit scenario

Bob requests to withdraw a small amount of funds. This user can either be malicious or just unaware of the potential issue. There are multiple other users that request to withdraw bigger amounts of funds. The withdrawal process for all users is blocked because Bob's `thisUserGet_` value is smaller than the `gasPerUser_` value.

Recommendation

Due to the requirement of subtracting gas, it is hard to avoid this issue completely. Minimal withdrawals are an unpleasant constraint for users. It is still possible that some gas will be at the expense of the protocol. This can be done by setting the `_totalGasLimit` parameter to a lower or zero value. Therefore, it is recommended to perform transaction simulations and set the `_totalGasLimit` parameter to a reasonable value according to the status of withdrawals.

Acknowledgment 1.1

We have simulation on withdraw handling routines that check actions by simulation.

— Cian Team

[Go back to Findings Summary](#)

W8: Users are not able to request more than one withdrawal

Impact:	Warning	Likelihood:	N/A
Target:	RedeemOperator.sol	Type:	Trust model

Description

The `registerWithdrawal` function in the `RedeemOperator` contract checks if a user has already requested a withdrawal. If the user has already requested a withdrawal, the function reverts. This mechanism prevents users from requesting more than one withdrawal.

This becomes an issue when users want to request withdrawal of additional funds, as they are forced to wait until their first withdrawal is processed.

Recommendation

Add a possibility for users to request more than one withdrawal.

Acknowledgment 1.1

This is by design.

— Cian Team

[Go back to Findings Summary](#)

W9: Potential issues with retrieving borrow and supply caps

Impact:	Warning	Likelihood:	N/A
Target:	AaveV3FlashLeverageHelper.sol	Type:	Logic error

Description

The contract retrieves borrow and supply caps from Aave. These caps are masked with 0x7FFFF, while the full range is 36 bits (0xFFFFFFFF). This masking can cause issues when the caps have higher values. Additionally, the contract can return 0 for unlimited caps, and this case should be handled.

Listing 12. Excerpt from AaveV3FlashLeverageHelper

```
67 function getSupplyCap() internal view returns (uint256) {
68     uint256 totalSupplied_ = IERC20(A_EZETH_AAVEV3).totalSupply();
69     uint256 configMap_ =
        POOL_AAVEV3.getReserveData(EZETH).configuration.data;
70     // Cut out bit 116-151 to get supply cap
71     return ((configMap_ >> 116) & 0x7FFFF) * 1e18 - totalSupplied_;
72 }
73
74 function getBorrowCap() internal view returns (uint256) {
75     uint256 totalBorrowed_ = IERC20(D_WSTETH_AAVEV3).totalSupply();
76     uint256 configMap_ =
        POOL_AAVEV3.getReserveData(WSTETH).configuration.data;
77     // Cut out bit 80-151 to get borrow cap
78     return ((configMap_ >> 80) & 0x7FFFF) * 1e18 - totalBorrowed_;
79 }
```

Recommendation

Add an appropriate mask and implement proper handling for zero values.

Partial solution 1.1

The mask is adjusted; however, if the cap is zero, the subsequent logic does not handle it as an unlimited cap.

[Go back to Findings Summary](#)

I1: The function can be declared as a view function

Impact:	Info	Likelihood:	N/A
Target:	VaultYieldBasic.sol, IStrategy.sol	Type:	Code quality

Description

The `underlyingTvl` function can be declared as a view function since it does not modify the state.

Listing 13. Excerpt from VaultYieldBasic

```
346 function underlyingTvl() public virtual returns (uint256) {
```

To implement this change, the following dependent functions must also be declared as view: - the `totalStrategiesAssets` function called from `underlyingTvl`; - the `getNetAssets` function in the `IStrategy` interface, which is currently declared as state-changing despite being view in strategy implementations.

Listing 14. Excerpt from IStrategy

```
5 function getNetAssets() external returns (uint256);
```

Recommendation

Change the functions mutability to `view`.

Acknowledgment 1.1

We need to prepare for future strategies that may require running hook functions within `getNetAssets()`, for example, we

may run claim reward logic before net value calculation.

— Cian Team

[Go back to Findings Summary](#)

I2: Missing documentation

Impact:	Info	Likelihood:	N/A
Target:	*	Type:	Code quality

Description

The codebase does not have any up-to-date documentation. While the codebase mostly contains NatSpec comments, it would be beneficial to have in-repository documentation with described architecture, design decisions, specifications and other relevant information, such as user scenarios and flows.

Recommendation

Add documentation and NatSpec code comments for the cross-chain contracts in the codebase (PoolArb, LzBridgeArb, ...).

Fix 1.1

Several NatSpec comments and basic documentation explaining the cross-chain contract flows were added to the codebase.

[Go back to Findings Summary](#)

I3: Typos and incorrect NatSpec comments

Impact:	Info	Likelihood:	N/A
Target:	*	Type:	Code quality

Description

The codebase contains some typos and incorrect NatSpec comments. The following code excerpts highlight some of them that were encountered during the review.

Missing comment for `_positionLimit` parameter:

Listing 15. Excerpt from StrategyFactory

```
85 /**
86  * @dev Allows the owner to create a new strategy.
87  * @param _impl The implementation address of the strategy.
88  * @param _initBytes The initialization parameters for the strategy.
89  */
90 function createStrategy(address _impl, bytes calldata _initBytes, uint256
   _positionLimit) external onlyOwner {
```

Missing comment for the `_offset` parameter and incorrect comment for the `_newPositionLimit` parameter:

Listing 16. Excerpt from StrategyFactory

```
111 /**
112  * @dev Update the temporary address of shares when users redeem.
113  * @param _newPositionLimit The new redeem operator address.
114  */
115 function updateStrategyLimit(uint256 _offset, uint256 _newPositionLimit)
   external onlyOwner {
```

Missing comment for the `_asset` parameter:

Listing 17. Excerpt from RedeemOperator

```
55 /**
56  * @dev Initializes the contract with the vault, operator, fee receiver, and
    gas parameters.
57  * @param _vault Address of the vault contract.
58  * @param _operator Address of the operator.
59  * @param _feeReceiver Address to receive fees.
60  */
61 constructor(address _admin, address _vault, address _asset, address
    _operator, address _feeReceiver)
```

Comment starts with "Then" instead of "The":

Listing 18. Excerpt from VaultYieldBasic

```
76 // Then allowed contract to mint unbacked shares
77 address public unbackedMinter;
```

Incorrect comment for the `_token` parameter ("deposit" → "withdrawal"):

Listing 19. Excerpt from VaultYieldBasic

```
466 /**
467  * @dev Redemption operation executed by the redeemOperator. Currently, only
    STETH and EETH redemptions are supported.
468  * @param _token The address of the token to deposit.
469  * @param _shares The amount of share tokens to be redeemed.
470  * @param _cutPercentage The percentage of the rebalancing loss incurred.
471  * @param _receiver The address of the receiver of the assets.
472  * @param _owner The owner address of the shares.
473  * @return assetsAfterFee_ The amount of assets obtained.
474  */
475 function optionalRedeem(address _token, uint256 _shares, uint256
    _cutPercentage, address _receiver, address _owner)
```

Incorrect comment for bits range (should be "80-115"):

Listing 20. Excerpt from AaveV3FlashLeverageHelper

```
77 // Cut out bit 80-151 to get borrow cap
```

Recommendation

Fix all typos and update NatSpec comments in the codebase.

Fix 1.1

Several typos were fixed.

[Go back to Findings Summary](#)

I4: Missing underscore in internal function's name

Impact:	Info	Likelihood:	N/A
Target:	*	Type:	Code quality

Description

The following internal functions should start with an underscore to distinguish them from external or public functions, ensuring better readability.

- Vault.sol: `optionalDepositDeal`
- BridgeHelper.sol: `getStorage`
- LayerZeroBridgeHelper.sol: `getHelperStorage`
- LzBridgeArb.sol: `getArbBridgeStorage`
- LzBridgeOp.sol: `getOpBridgeStorage`
- UnbackedMintAuthority.sol: `getStorage`
- Pool.sol: `getStorage`
- PoolArb.sol: `getArbPoolStorage`
- PoolOp.sol: `getOpPoolStorage`
- Timelock.sol: `getBlockTimestamp`

Recommendation

Add an underscore prefix to all internal function names to follow naming conventions.

[Go back to Findings Summary](#)

15: Modifier consistency on access controls

Impact:	Info	Likelihood:	N/A
Target:	VaultYieldBasic.sol	Type:	Code quality

Description

The codebase uses modifiers for access control; however, in some places there are inconsistencies, and access controls are checked in different ways. These inconsistencies can potentially lead to bugs in future development. Therefore, it is better to choose one way of checking access controls and use it consistently.

Listing 21. Excerpt from VaultYieldBasic

```
308 function transferToStrategy(address _token, uint256 _amount, uint256
    _strategyIndex) external {
309     address caller_ = msg.sender;
310     if (_strategyIndex == 0) {
311         if (caller_ != owner() && caller_ != vaultParams.rebalancer) revert
            Errors.InvalidOperator();
312     } else {
313         if (caller_ != owner()) revert Errors.InvalidOperator();
314     }
```

Listing 22. Excerpt from VaultYieldBasic

```
475 function optionalRedeem(address _token, uint256 _shares, uint256
    _cutPercentage, address _receiver, address _owner)
476     public
477     override
478     nonReentrant
479     whenNotPaused
480     returns (uint256 assetsAfterFee_)
481 {
482     if (!tokens.contains(_token)) revert Errors.InvalidAsset();
483     if (msg.sender != vaultParams.redeemOperator) revert
        Errors.UnsupportedOperation();
```

Listing 23. Excerpt from VaultYieldBasic

```
584 function collectManagementFee() external {  
585     if (msg.sender != vaultParams.feeReceiver) revert  
    Errors.InvalidFeeReceiver();
```

Listing 24. Excerpt from VaultYieldBasic

```
600 function collectRevenue() external {  
601     if (msg.sender != vaultParams.feeReceiver) revert  
    Errors.InvalidFeeReceiver();
```

Listing 25. Excerpt from VaultYieldBasic

```
607 function pause() external {  
608     if (msg.sender != owner() && msg.sender != vaultParams.rebalancer)  
    revert Errors.UnSupportedOperation();
```

Recommendation

Unify the modifier usage for access control across the codebase.

[Go back to Findings Summary](#)

I6: Unused variable

Impact:	Info	Likelihood:	N/A
Target:	AaveV3FlashLeverageHelper.sol	Type:	Code quality

Description

The contract contains an unused variable `leverageableAmount_` in the code.

Listing 26. Excerpt from AaveV3FlashLeverageHelper

```
103 uint256 leverageableAmount_ = getLeverageableAmount(balance_); // Hold 5%  
    for success rate  
104 leverageableAmount_ = leverageableAmount_ * 95 / 100;  
105 if (leverageableAmount_ > supplyCap_ - balance_) {  
106     leverageableAmount_ = supplyCap_ - balance_;  
107 }
```

Recommendation

Utilize or remove the unused variable.

Fix 1.1

The unused variable is removed.

[Go back to Findings Summary](#)

I7: Unused using-for directives

Impact:	Info	Likelihood:	N/A
Target:	LayerZeroBridgeHelper.sol, StrategyFactory	Type:	Code quality

Description

The contracts contain unused using-for directives. The following code excerpts list all of them.

Listing 27. Excerpt from LayerZeroBridgeHelper

```
9 using SafeERC20 for IERC20;
```

Listing 28. Excerpt from StrategyFactory

```
21 using SafeERC20 for IERC20;
```

Recommendation

Remove the unused using-for directives.

Fix 1.1

The unused using-for directives are removed.

[Go back to Findings Summary](#)

I8: Unused imports

Impact:	Info	Likelihood:	N/A
Target:	StrategyFactory.sol	Type:	Code quality

Description

The contract contains unused imports. The following code excerpt lists all of them.

Listing 29. Excerpt from StrategyFactory

```
8 import "@openzeppelin/contracts-upgradeable/utils/PausableUpgradeable.sol";
9 import "@openzeppelin/contracts-upgradeable/utils/ReentrancyGuardUpgradeable.sol";
10 import "../interfaces/IRedeemOperator.sol";
```

Recommendation

Remove the unused imports.

Fix 1.1

The unused imports are removed.

[Go back to Findings Summary](#)

I9: Unused events

Impact:	Info	Likelihood:	N/A
Target:	StrategyFactory.sol	Type:	Code quality

Description

The contract contains unused events. The following code excerpt enumerates all of them.

Listing 30. Excerpt from StrategyFactory

```
36 event UpdateOperator(address oldOperator, address newOperator);
```

Recommendation

Remove the unused events or utilize them.

Fix 1.1

The unused events are removed.

[Go back to Findings Summary](#)

I10: Unchecked return value for OFT receipt

Impact:	Info	Likelihood:	N/A
Target:	LzSend.sol	Type:	Logging

Description

The `send` function in the OFT contract returns a receipt containing cross-chain transfer data. This receipt is not captured or logged, which could be valuable for logging purposes and user experience improvements.

Listing 31. Excerpt from LzSend

```
27 IOFT(_oftAdapter).send{value: msg.value}{  
28     params_,  
29     fee_,  
30     msg.sender  
31 };
```

Recommendation

Capture the receipt data from the external call and emit it in an event.

Fix 1.1

The receipt data is captured and emitted in an event.

[Go back to Findings Summary](#)

Report Revision 1.1

Revision Team

Revision team is the same as in [Report Revision 1.0](#).

System Overview

The main logic of the system remains unchanged.

Trust Model

There are introduced changes that help improve the trust model. Minting caps for receivers of unbacked tokens are introduced (see [W6](#) issue), and pool exchange price is now more limited (see [W4](#) issue).

Fuzzing

The fuzz tests are updated to cover the new changes, and no new issues were found.

Appendix A: How to cite

Please cite this document as:

[Ackee Blockchain Security](#), Cian: Yield Layer, 14.1.2025.

Appendix B: Wake Findings

This section lists the outputs from the [Wake](#) framework used for testing and static analysis during the audit.

B.1. Fuzzing

The following table lists all implemented execution flows in the [Wake](#) fuzzing framework.

ID	Flow	Added
F1	Confirm withdrawal of assets from the RedeemOperator	1.0
F2	Request withdrawal of assets	1.0
F3	Burn unbacked shares	1.0
F4	Mint with asset from the UnbackedMintAuthority	1.0
F5	Collect management fee	1.0
F6	Collect revenue	1.0
F7	Update exchange price	1.0
F8	Mint to the Vault	1.0
F9	Deposit to the Vault	1.0
F10	Optional deposit to the Vault	1.0
F11	Transfer funds to Strategy from the Vault	1.0
F12	Mint funds to Strategy (mocking the created yield in the Strategy mock)	1.0
F13	Transfer funds from Strategy to the Vault	1.0
F14	Create a new Strategy	1.0
F15	Remove existing Strategy	1.0
F16	Update the Strategy limit	1.0

ID	Flow	Added
F17	Send OFT from L2 to mainnet with LayerZero	1.0
F18	Receive asset on the Optimism bridge contract	1.0
F19	Receive asset on the Arbitrum bridge contract	1.0
F20	Transfer funds from bridge to the mint authority	1.0
F21	Add liquidity from mainnet to the Optimism pool	1.0
F22	Add liquidity from mainnet to the Arbitrum pool	1.0
F23	Collect funds from the Optimism pool	1.0
F24	Collect funds from the Arbitrum pool	1.0
F25	Deposit to pool (Arbitrum, Optimism)	1.0
F26	Update pool exchange price (Arbitrum, Optimism)	1.0
F27	Pause pool (Arbitrum, Optimism)	1.0
F28	Unpause pool (Arbitrum, Optimism)	1.0
F29	Set pool deposit fee rate (Arbitrum, Optimism)	1.0
F30	Set pool allowed deposit token (Arbitrum, Optimism)	1.0
F31	Enable pool direct withdrawal (Arbitrum, Optimism)	1.0
F32	Disable pool direct withdrawal (Arbitrum, Optimism)	1.0
F33	Add token to the Vault	1.0
F34	Remove token from the Vault	1.0
F35	Call stakeTo on the Vault	1.0
F36	Pause the Vault	1.0
F37	Unpause the Vault	1.0
F38	Update roles on the Vault	1.0
F39	Set the unbacked mint authority	1.0
F40	Set the Vault exit fee rate	1.0

ID	Flow	Added
F41	Set the Vault revenue fee rate	1.0
F42	Set the Vault max price update period	1.0
F43	Set the Vault management fee claim period	1.0
F44	Set the Vault management fee rate	1.0
F45	Set the Vault market capacity	1.0

Table 4. Wake fuzzing flows

The following table lists the invariants checked after each flow.

ID	Invariant	Added	Status
IV1	Transactions do not revert except where explicitly expected	1.0	Success
IV2	Underlying TVL matches the returned value by <code>totalAssets</code> with a defined tolerance	1.0	Success
IV3	The <code>totalSupply</code> of the Vault matches the expected value	1.0	Success
IV4	The <code>unbackedMintedAmount</code> of the Vault matches the expected value	1.0	Success
IV5	The total supply of OFTs matches the expected value	1.0	Success
IV6	Pending withdrawal amounts and addresses matches the expected value	1.0	Success

Table 5. Wake fuzzing invariants



Thank You

Ackee Blockchain a.s.

Rohanske nabrezi 717/4
186 00 Prague
Czech Republic

hello@ackee.xyz