SMART CONTRACTS REVIEW

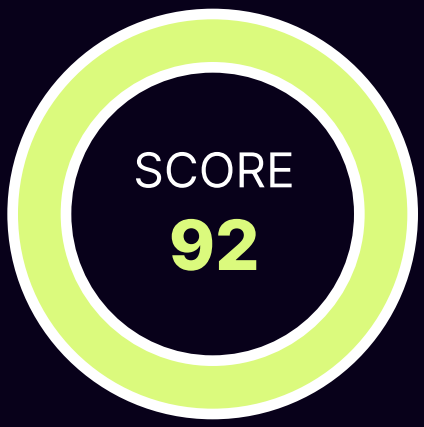
zokyo

October 21st 2024 | v. 1.0

Security Audit Score

PASS

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



ZOKYO AUDIT SCORING ISLE FINANCE

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.

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: 0 points deducted
- 2 Medium issues: 2 resolved = 0 points deducted
- 13 Low issues: 7 resolved and 6 acknowledged = - 8 points deducted
- 7 Informational issues: 3 resolved and 4 acknowledged = 0 points deducted

Thus, 100 - 8 = 92

TECHNICAL SUMMARY

This document outlines the overall security of the Isle Finance smart contract/s evaluated by the Zokyo Security team.

The scope of this audit was to analyze and document the Isle Finance smart contract/s codebase for quality, security, and correctness.

Contract Status

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

It should be noted that this audit is not an endorsement of the reliability or effectiveness of the contract/s 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 Isle Finance team put in place a bug bounty program to encourage further active analysis of the smart contract/s.

Table of Contents

AUDITING STRATEGY AND TECHNIQUES APPLIED

The source code of the smart contract was taken from the Isle Finance repository:
Repo: https://github.com/isle-labs/isle-contract
Initial commit: 1f1715851e675e381065cf24f516648abd1811ca
Final commit: d140f0a4482ab4c2a5c6037684924fee5203cd40

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

- ./ReceivableStorage.sol
- ./Pool.sol
- ./WithdrawalManager.sol
- ./libraries/types/DataTypes.sol
- ./libraries/Errors.sol
- ./libraries/PoolDeployer.sol
- ./libraries/upgradability/UUPSProxy.sol
- ./libraries/upgradability/VersionedInitializable.sol
- ./Receivable.sol
- ./PoolAddressesProvider.sol
- ./abstracts/Governable.sol
- ./PoolConfigurator.sol
- ./LoanManager.sol
- ./PoolConfiguratorStorage.sol
- ./LoanManagerStorage.sol
- ./WithdrawalManagerStorage.sol
- ./IsleGlobals.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 Isle Finance smart contract/s. 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 contract/s by industry leaders.

03 Thorough manual review of the codebase line by line.

Executive Summary

Isle Finance uses a novel approach by combining blockchain technology to achieve a decentralized protocol for an easily accessible supply chain ecosystem. Their mission is to bring the world's supply chain activity on the chain while expanding capital access and further fortifying supply chain resilience through the use of smart contracts. The protocol creates various roles such as the Governor role (The Isle Council consisting of 3-of-5 multisig wallets), the Pool Admins who manage buyer pools for the protocol, the Buyers consisting of companies that approve receivables to the corresponding Seller through the pool, the Seller who receives early payments for buyer approved receivables and the Lender which can be the everyday user. They are responsible for supplying capital to liquidity pools.

Sellers will first mint a receivable NFT through the Receivable contract. The Buyer then requests a loan from the LoanManager where the Pool Admin will review the requested loan and choose to fund the loan. The Seller can then withdraw those funds from the LoanManager and use those funds for their day-to-day operations after transferring their receivables to the LoanManager. Buyers will then be required to repay the loan via the LoanManager with interest. Should the Buyer be late on their payments, they will be penalized through an additional interest rate modifier. If they are not able to repay their loan, the Pool Admin will have the authority to place their position into a defaulted state.

Zokyo was tasked with conducting a security assessment of the smart contract component of Isle's tech stack. Overall the code is sound, well modularised, and separated into components which make maintenance easier with separated out contracts responsible for smaller sections of the overall ecosystem. The pool Admin actions are derived by incentive-based design that ensure all the parameters such as setting interest rate, ensuring loan repayment which reduce the centralization risk and other important configurations like pool limit and default coverage percentage are controlled by governor which mitigate the centralization risk from smart contract. A range of issues have been identified by Zokyo's security team ranging from Medium down to Informational. Various recommendations have been made alongside a description of the findings in the report to create more validation of input variables as there should be less assumption that a privileged actor is behaving honestly.

In addition to this, recommendations have been made regarding correcting the mathematical issues found in the codebase around interest rates and repayments, in addition to the various other findings. Various Low level and informational issues have also been suggested to mitigate the risks around ERC721 contracts including undiscovered reentrancy vulnerabilities.

Using the foundry framework, whilst unit tests do exist making it easier for Zokyo's security team to set up scenarios for manual testing, more positive and negative testing can always be done in order to assert that the code is doing exactly what is intended. It's generally recommended that there is at least a 90% unit test coverage of the code base. In addition to this, stress testing in the form of Fuzz testing is also suggested in order to catch edge case scenarios that may be exploitable or otherwise unexpected during day to day operations. During the security assessment, there was some time allotted in an effort to identify upgradeability issues in the IsleGlobal and Receivable contracts as well as searching for potential issues around deployment to the Plume and Base chains which are L2 blockchains closely related to Ethereum. It's recommended that the developers carefully review the audit report and implement the fixes recommended for which a fix review will take place to ensure that further vulnerabilities have not been introduced into the codebase through such fixes.

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 Isle Finance team and the Isle Finance 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	Inconsistent Loan Impairment Handling Due to Missing Reset of isImpaired Flag	Medium	Resolved
2	Lack of Validation for Loan Funding Status	Medium	Resolved
3	Rounding operations for late interest payments may result in slight overcharges	Low	Acknowledged
4	Interest rates can be arbitrary which can allow the loan requester to set them to zero bypassing interest and late interest rates if funded	Low	Acknowledged
5	The gracePeriod_ parameter in the requestLoan function of the LoanManager can be set indefinitely allowing the attacker to protect themselves from defaults if funded	Low	Acknowledged
6	Governable is an abstract upgradeable contract but doesn't contain any storage gaps which could result in collisions on upgrade	Low	Resolved
7	Users Might Miss The Withdrawal Window While The Protocol Is Paused	Low	Acknowledged
8	NFT Holder Cannot Withdraw Repaid Amount Due to Loan Seller Check	Low	Acknowledged
9	Incorrect Sharing of ERC20 Allowance State Across Functions in Pool Contract	Low	Acknowledged
10	Parallel governor value in IsleGlobal and Receivable contract may result in two separate governor addresses	Low	Resolved
11	Incorrect ReentrancyGuard Version Used in Upgradeable LoanManager Contract	Low	Resolved
12	maxWithdraw() of ERC4626 Vaults must not revert	Low	Resolved

#	Title	Risk	Status
13	Missing zero address check in Governable can lead to loss of Ownership	Low	Resolved
14	Single-step ownership transfer	Low	Resolved
15	Centralization of Critical Loan Management Functions and Parameter Configurations	Low	Resolved
16	Reentrancy Vulnerability in the createReceivable Function Allowing Multiple Asset Minting	Informational	Acknowledged
17	Incorrect Shares Attributed While Depositing	Informational	Acknowledged
18	Optimization: Redundant Checks in depositWithPermit Function	Informational	Resolved
19	The redeem() function reverts on valid values of shares	Informational	Resolved
20	Lack of reentrancy guards where ERC721 is used increases the risk profile for reentrancy issues	Informational	Acknowledged
21	Use of Floating Solidity Version (pragma solidity ^0.8.19)	Informational	Resolved
22	Imprudent Withdrawal of Funds from Impaired Loans	Informational	Acknowledged

MEDIUM-1 | RESOLVED

Inconsistent Loan Impairment Handling Due to Missing Reset of `isImpaired` Flag

Description:
The `removeLoanImpairment` function in the `LoanManager` contract fails to reset the `isImpaired` flag to `false` after removing a loan's impairment. As a result, the loan remains incorrectly marked as impaired, leading to inconsistencies in how the loan is managed by the contract. This can cause incorrect calculations of unrealized losses and improper handling of loan defaults.

Scenario:
1. A loan is impaired by calling the `impairLoan` function, which correctly sets the `isImpaired` flag to `true`.
2. Later, the `removeLoanImpairment` function is called to remove the impairment. However, the function does not reset the `isImpaired` flag to `false`.
3. Despite the loan no longer being impaired, subsequent operations (such as `triggerDefault`) continue to treat the loan as impaired, leading to incorrect behavior and calculations.
4. For example, the unrealized losses calculated in `PoolConfigurator::triggerDefault` may be overstated because the loan is still incorrectly considered impaired.

Recommendation:
Update the `removeLoanImpairment` function to include a line that resets the `isImpaired` flag to false after successfully removing the impairment. This ensures that the loan's state is correctly managed and that subsequent operations handle the loan as expected

Lack of Validation for Loan Funding Status

Description:
The `withdrawFunds` function does not validate whether the loan has been fully funded before allowing the seller to withdraw funds. This could lead to a situation where a seller attempts to withdraw funds before the loan is adequately capitalized.

Scenario:
- A loan is created, but for some reason, the funding process is delayed or incomplete.
- The seller, without realizing the loan isn't fully funded, calls the `withdrawFunds` function.
- Since there's no validation check, the function might proceed with transferring funds, leading to an unexpected contract state or even financial discrepancies.

Impact: This can lead to an inconsistent state where the loan is not properly funded but funds are withdrawn. This could also potentially open up avenues for abuse, where a seller could try to withdraw funds from a partially funded loan.

Recommendation:
Add a validation check in the `withdrawFunds` function to ensure that the loan's funding status is confirmed before allowing any withdrawal. This can be achieved by checking if the loan's `startDate` is set (indicating the loan is funded).

Client comment: This is confirmed to be an issue: if the Pool Admin has not invoked `LoanManager::fundLoan` and the seller has already triggered `LoanManager::withdrawFunds`, the receivable will be transferred from the seller to the LoanManager contract. However, no asset tokens will be transferred since the loan's `drawableFunds` is not a non-zero amount. This prevents the user from withdrawing the funds again, making the receivable and the corresponding loan unusable.

Rounding operations for late interest payments may result in slight overcharges

Location: LoanManager.sol#_getLateInterest,_getInterest

Description:
Users of the protocol are able to utilize the LoanManager contract in order to take out loans which are required to be repaid with interest. Should a user be late on repayments, they will incur a late modifier on their interest rate. This is denoted by the periodicInterestRate through the following formula:

*fullDaysLate=((currentTime - dueDate + (86400 - 1)) / 86400 * 86400;*
*periodicInterestRate = (interest * (1e18 / 1e6) * fullDaysLate) / uint256(86400 * 365);*

Let `interest` be `0.12*1e6`, `currentTime` be `block.timestamp` and `dueDate` be `block.timestamp - 1 days` meaning repayments are late by one day. If this is inserted into the formula, a one day late periodic interest rate of `328767123287671` is paid. With this in mind, if the equation is reevaluated to modify the `dueDate` to `block.timestamp - (2 days + 5 hours)` meaning the user is late on repayments by 2 days and 5 hours. This would return a result of `986301369863013`. If `986301369863013` is divided by `328767123287671` it can be seen that the periodic interest of the 1 day late value perfectly fits into the overdue value by 2 days and 5 hours exactly 3 times which means if a user is late on their repayments by just one second, they will be required to pay the full days interest (in this case, the user will have to pay a 3 day late interest rate when in reality they are late by 2 days and 5 hours). The result is that users could be overpaying on their late interest rates resulting in an unnecessary loss of funds.

Recommendation:
It's recommended that the developers refactor the late interest repayment units to reflect an interest rate per second as opposed to interest rate per day. This is because Solidity will truncate values when dividing in mathematical operations and as a result, rounding errors may cause users to pay more than what is necessary to return their position to a healthy state.

Client comment: We acknowledged optimizing the interest calculation per second would improve the accuracy and fairness of the protocol, however, using the current rounding does not pose a significant risk to users.

LOW-2 | ACKNOWLEDGED

Interest rates can be arbitrary which can allow the loan requester to set them to zero bypassing interest and late interest rates if funded

Location: LoanManager.sol#requestLoan

Description:
The `LoanManager` allows a buyer to request a loan from the pool using the `requestLoan` function. The buyer can pass various parameters to the function such as the receivable asset, the receivables token id, grace period, principal requested and the interest rates as an array. The `interestRate` (as defined by the 0th index of the `rates_` array) and the `lateInterestPremiumRate` (as defined by the 1st index of the `rates_` array) can effectively be arbitrary. This can allow the buyer to specify these as zero in order to bypass interest rates.

Recommendation:
It's recommended that the contract admin sets these interest rates using setters within the contract. The interest rates array should be validated against these state variables.

Client comment: Pool Admin use a multi-signature wallet, such as Safe, instead of a single wallet. This will enhance security and ensure that loan approvals are more robust, reducing the risk of arbitrary or bad loans being approved by a single party.

The gracePeriod_ parameter in the requestLoan function of the LoanManager can be set indefinitely allowing the attacker to protect themselves from defaults if funded

Location: LoanManager.sol#requestLoan

Description:
The `requestLoan` function of the `LoanManager` contract allows a buyer when requesting a loan to specify a grace period for the loan. This is seen as a "warning" period for which the borrower will have to return their loan to a healthy state or else their position will be put into a defaulted state by the pool configurator admin. There exists a condition in the `triggerDefault` function of the loan manager which will check if the loan is past the supplied grace period.

```
/// @inheritdoc ILoanManager
    function triggerDefault(uint16 loanId_)
        external
        override
        whenNotPaused
        onlyPoolConfigurator
        returns (uint256 remainingLosses_, uint256 protocolFees_)
    {
        // check if current time is past the due date plus grace period
        if (block.timestamp <= _loans[loanId_].dueDate + _loans[loanId_].gracePeriod) {
            revert Errors.LoanManager_NotPastDueDatePlusGracePeriod({ loanId_: loanId_ });
        }
```

Because this value can be set to an indefinite value (ie. a date that is excessively far into the future), this can prevent the admin from triggering a default on the loan if repayments aren't made.

Recommendation:
Consider allowing the pool configurator admin to set a maximum threshold after a loan due date for which grace periods can be set to prevent overly extensive grace periods. It's recommended that there are additional validations made on the grace period when requesting a loan.

Client comment: This scenario is similar to the one mentioned in this issue's comments. While the buyer can set arbitrary parameters in the LoanManager::requestLoan function, the request may still be rejected by the Pool Admin.

LOW-4 | RESOLVED

Governable is an abstract upgradeable contract but doesn't contain any storage gaps which could result in collisions on upgrade

Location: Governable.sol

Description:
The Governable.sol contract is a part of the Receivable and IsleGlobals contracts. For Upgradeable contracts, there must be a storage gap in order to freely add new state variables in future versions without compromising the integrity of the storage compatibility. The absence of storage gaps can result in the overwriting of variables in the child contract if new variables are added to the Goverenable contract.

Recommendation:
It's recommended that storage gaps are added to the Governable contract in order to avoid risk of storage collisions.

Users Might Miss The Withdrawal Window While The Protocol Is Paused

User withdrawals only happen in respective windows , which is current window +2 , but the user might miss the withdrawal window while the protocol was paused . The pausing mechanism does not extend the window timeframe therefore if a user had his withdrawal window between x and y and the protocol was paused within this time frame , he would miss his window and now he has to remove some of his tokens when the protocol resumes in order to push his window to a new current window where he can withdraw.

Also, a loan might get subject to a default while the protocol was paused . In the paused state user won't be able to repay the loan and as soon as the system gets unpaused the loan is subject to a default and gets defaulted.

Recommendation:
Account for the paused timeframe within the withdrawal window or the users should be acknowledged of such behavior in advance.

Client comment: If for some reason the protocol must be temporarily paused, for example, if the pool admin is undergoing legal procedures to recover defaulted funds, we will inform users in advance that the protocol will be temporarily paused.

NFT Holder Cannot Withdraw Repaid Amount Due to Loan Seller Check

Location LoanManager.sol, Receivable.sol

The Receivable NFT, which represents a transferable debt claim, restricts the ability to withdraw the repayment of the debt to the original lender (loan_.seller) while allowing the transfer of the NFT. In the `LoanManager.sol` contract, the `withdrawFunds()` function checks if the caller is the original lender, preventing new NFT holders from accessing repaid funds. This disallows the transfer of the debt along with its claim, leading to potential disputes and misrepresentations in transactions involving the NFT.

Recommendation:
Since that the debt is meant to be claimed by the original lender, it would be consistent to have non-transferrable Receivable NFT. This change will ensure that the current owner of the Receivable NFT can rightfully claim the repayment of the debt.

Client comment: The amount that can be withdrawn is not the repaid amount, it is the early payment amount from Isle's pool. Right now we fix this issue through a removed seller address check in the `withdrawalFunds()`, we let the holder of NFT have the right to withdraw.

Incorrect Sharing of ERC20 Allowance State Across Functions in Pool Contract

In the Pool contract, the ERC20 allowance state of the shares is shared across multiple functions (`requestRedeem`, `redeem`, and `removeShares`), which can lead to potential misuse by a spender in a manner not intended by the owner. Specifically, if a user approves a certain amount of shares to one spender for `requestRedeem` and later to another spender for redeem, the second spender could mistakenly or maliciously call `requestRedeem` or `removeShares`, exploiting the granted allowance.

This scenario demonstrates that sharing approval states across different functionalities introduces complex behavior not originally intended in the ERC20 standard, which is designed primarily for token transfers between wallets. Such complexities necessitate a more robust logic to ensure security and proper handling of ERC20 allowances.

Recommendation:
To mitigate this issue, it is recommended to introduce a new wrapped token in the Pool contract to represent the shares under the custody of the `WithdrawalManager`. This wrapped token should have its own allowance and transfer mechanisms, separate from the primary ERC20 token. By doing so, the risk of misuse shall be reduced and align the implementation more closely with the original ERC20 standard's intended use case.

Client comment: Since the ways of spending the allowance are clearly defined within the contract, there's no need to implement a 1:1 allowance control logic like approve() and transferFrom(). For example, if a user approves an allowance to a spender using an external ERC20, we implement transferFrom() within our requestRedeem, redeem, and removeShares functions. This means the user still can't control what specific actions the spender might take (if the spender is an EOA). However, if the spender is a verified contract (open source), the user can review the function logic to verify the specific actions being executed.
The reason ERC20 requires a 1:1 approve() and spending logic is to maintain compatibility and flexibility. This setup allows other contracts to directly integrate with transferFrom() to use the allowance, ensuring seamless interaction with various external contracts. Within the ERC20 standard itself, the logic is straightforward: one approve() function is directly tied to the specific spending function transferFrom(), which keeps things clean and simple.
However, outside of ERC20, the actual action performed still depends on the function being executed in the contract address (CA) spender or, if it's an EOA, on the spender's behavior.

Parallel governor value in IsleGlobal and Receivable contract may result in two separate governor addresses

Location: IsleGlobal.sol,Receivable.sol

Description:
The governor address for the Isle protocol has the authority to make changes and modify settings which are critical to the protocol's day to day operations. This address is stored in both the Receivable and IsleGlobal contract however, since the address is set twice (once for each contract), this may cause inconsistent addresses resulting in inconsistent data if one were to be changed but not the other due to human error or some other reason.

Recommendation:
It's recommended that the Governable inherited contract is removed from the Receivable contract and the governor address is obtained from IsleGlobal.

LOW-9 | RESOLVED

Incorrect ReentrancyGuard Version Used in Upgradeable LoanManager Contract

The `LoanManager` contract is intended to be upgradeable; however, it imports the non-upgradeable version of `ReentrancyGuard` from OpenZeppelin. This can lead to issues, as the non-upgradeable version does not have the necessary initializers and storage gap reserved for upgradeable contracts, which are essential for proxy-based upgradeability mechanisms.

Recommendation:
To resolve this issue, replace the inheritance of the non-upgradeable `ReentrancyGuard` with the upgradeable version from OpenZeppelin, i.e., `ReentrancyGuardUpgradeable`. Make sure to call the `__ReentrancyGuard_init` function as part of the initialization process in the initializer function of `LoanManager`. Additionally, ensure that the contract abides by the storage gap requirements to maintain proper alignment and compatibility between contract upgrades.

Comment: The client is considering updating the library accordingly.

LOW-10 | RESOLVED

maxWithdraw() of ERC4626 Vaults must not revert

According to EIP-4626, **maxWithdraw()** must not revert. But the function maxWithdraw() reverts in the when called in the Pool contract which uses the ERC-4626 standard. According to the EIP, if withdrawals are entirely disabled (even temporarily) it MUST return `0`.

Recommendation:
It is advised to NOT revert the maxwithdraw() function but to instead return `0` in order to comply with the original EIP-4626 standard.

Missing zero address check in Governable can lead to loss of Ownership

In Governable.sol's transferGovernor() function, there is missing zero address check. This can lead to accidentally losing ownership if a zero address is passed as a parameter to this function. This will lead to the admin losing access to the onlyGovernor() functions forever and make it unusable.

In addition to this, the Receivable.sol's initialize() function is missing zero address check for initialGovernor_ parameter, which can also make onlyGovernor functions uncallable and make transferGovernor unusable too.

Recommendation:

It is advised to do the following changes:

1. Introduce a zero address check in the Governable.sol's transferGovernor() function for the newGovernor parameter
2. Introduce a zero address check for Receivable.sol's initialize() function for the initialGovernor_ parameter

Single-step ownership transfer

Location: Governable.sol

The `transferGovernor` function in the Governable.sol contract performs the governance transfer in a single step. This approach poses a risk of accidental ownership transfers, which can lead to potential loss of control over the contract. Implementing a two-step governance transfer process, where the new owner must confirm the acceptance of governance, would mitigate this risk and ensure a more secure ownership transfer mechanism.

Recommendation:
Modify the `transferGovernor` function to implement a two-step governance transfer process. In the first step, set the new owner as a pending governor. In the second step, require the pending governor to accept the ownership transfer explicitly. This will ensure that the governance transfer is intentional and verified by the new owner, reducing the risk of unintended ownership changes.

Comment: The client confirmed to implementation a mechanism to prevent unexpected ownership transfers.

Centralization of Critical Loan Management Functions and Parameter Configurations

Loan management functions, such as approving loans, marking loans as impaired, reversing impairment decisions, and declaring borrower defaults, are centralized and controlled by an admin. Parameter settings like `adminFee`, `maxCoverLiquidation`, `minCover`, `poolLimit`, `windowDuration`, and `cycleDuration` are also managed by the admin. Although a multisig wallet is intended for these functions to reduce individual risks, significant centralization remains inherent in the protocol.

Recommendation:
To mitigate the risks associated with centralization, consider the following recommendations:

1. **Multisig Wallet Adoption:** Implement the intended multisig wallet to add multiple layers of approval for critical functions. Ensure that the multisig contract is robust and secure.
2. **Decentralized Governance:** Gradually migrate to a decentralized governance model where the community can partake in significant decisions. Utilize governance tokens and voting mechanisms to distribute the decision-making process.
3. **Time-lock Mechanism:** Introduce a time-lock mechanism for critical administrative functions, allowing the community to review and react to changes before they are finalized.

Client comment: These three variable's setter function hava onlyGovernor modifier (maxCoverLiquidation, minCover and poolLimit), so pool admin have no right to control them, and the governor is a muti-sig

Reentrancy Vulnerability in the createReceivable Function Allowing Multiple Asset Minting

Location: Receivable.sol

The createReceivable function in the `Receivable` contract is vulnerable to reentrancy attacks. This vulnerability arises when the `_safeMint` function is called, leading to the potential for reentrancy exploits where the minting event can execute multiple times, causing inconsistency between the actual state and emitted events.

Recommendation:
To mitigate the reentrancy vulnerability, it is recommended to use a reentrancy guard to prevent the `_safeMint` function from being invoked multiple times within the same transaction.

Client comment: This is a valid operation; however, the seller can mint multiple receivables through Receivable::createReceivable by default. Although multiple receivables are created, they must still be approved by the Pool Admin before proceeding with the LoanManager::requestLoan operation. We are uncertain about the potential impact of minting a large number of receivables at once. While re-entrancy can occur during minting, the event emission remains consistent and in the correct order.

Suggested action: Go back through the code and absolutely assert and triple check that free minting receivables doesn't have any impact to the protocol

Incorrect Shares Attributed While Depositing

When depositing the Pool contract follows the ERC4626 standard as follows →

```
function deposit(uint256 assets, address receiver) public override returns (uint256
shares) {
        // Checks: receiver is not the zero address.
        if (receiver == address(0)) revert Errors.Pool_RecipientZeroAddress();

        // Checks: deposit amount is less than or equal to the max deposit.
        if (assets > maxDeposit(receiver)) revert
Errors.Pool_DepositGreaterThanMax(assets, maxDeposit(receiver));

        shares = previewDeposit(assets);
        _deposit(_msgSender(), receiver, assets, shares);

        return shares;
```

The shares minted are calculated via the previewDeposit function(which uses _convertToShares()) in the ERC4626 library , but in the isle ecosystem this would be incorrect and more than required shares would be minted to the user . This is because this calculation does not include the unrealized losses in the system . To correctly account for the unrealized losses , instead of using previewDeposit , use _convertToExitShares() .

The same is applicable for the minting function where _convertToExitShares should be used instead.

Recommendation:
Use the recommended functions instead.

Client comment: We use unrealizedLoss to create a less favorable exchange rate, hoping to discourage users from withdrawing in the event of a loan impairment. This approach serves as a soft deterrent rather than a direct prevention of withdrawals. The unrealizedLoss itself only affects withdrawals and does not impact deposits.

Optimization: Redundant Checks in `depositWithPermit` Function

Location: Pool.sol

The current implementation of the `depositWithPermit` function uses the `deposit` function, which performs redundant checks that `depositWithPermit` has already handled.

```
66        shares_ = deposit(assets_, receiver_);
```

This inefficiency can be mitigated by using the `previewDeposit` function to calculate the share amount and then directly calling `_deposit` for the actual deposit.

Recommendation:
Replace the line `shares_ = deposit(assets_, receiver_);` in the `depositWithPermit` function with:

```
shares = previewDeposit(assets);

_deposit(_msgSender(), receiver, assets, shares);
```

This change eliminates redundant checks and improves the efficiency of the `depositWithPermit` function.

Comment: Client confirms to follow the recommendation to fix it.

The redeem() function reverts on valid values of shares

In **Pool.sol**, the **redeem()** function does not allow any value of shares as a parameter to be passed other than a value equal to **lockedShares**. This is because, **redeem()** function calls **processRedeem() from PoolConfigurator**, which itself calls **processExit() from WithdrawalManager**.

Line: 200 in Pool.sol

```
(redeemableShares_, assets_) = IPoolConfigurator(configurator).processRedeem(shares_, owner_, _msgSender());
```

Line: 228 in PoolConfigurator.sol

```
(redeemableShares_, resultingAssets_) = _withdrawalManager().processExit(shares_, owner_);
```

Now in this processExit() function, there is an if statement on line:252 which uses strict equality (or inequality here) which results in a revert if requestedShares_ is not equal to the lockedShares.

Line: 252 in WithdrawalManager.sol

```
 if (requestedShares_ != lockedShares_) {
            revert Errors.WithdrawalManager_InvalidShares(owner_, requestedShares_, lockedShares_);
        }
```

But this strict inequality results in the pool.redeem(uint256 shares) to fail for any value of shares other than lockedShares_.

Ideally this is not expected as the function should work for any valid value of shares which can be set to any value as a parameter shares_ of the redeem() function.

Recommendation:

It is advised to change the if statement to > instead of != on line: 252 of WithdrawalManager.sol's processExit() function. Also review the business logic and operational logic for the said changes.

INFORMATIONAL-5 | ACKNOWLEDGED

Lack of reentrancy guards where ERC721 is used increases the risk profile for reentrancy issues

Location:Receivable.sol#createReceivable,LoanManager.sol#withdrawFunds

Recommendation:
There are various locations in the codebase where an ERC721 token is being transferred or minted; however, there are no reentrancy guards used. It's recommended that for functions that are interacting with external contracts that support a callback functionality (for instance ERC721.safeMint and ERC721.safeTransferFrom calls upon an onERC721Received callback during a safe transfer) uses a reentrancy guard where recursive programming is not desired to mitigate the risk of undiscovered reentrancy vulnerabilities.

INFORMATIONAL-6 | RESOLVED

Use of Floating Solidity Version (pragma solidity ^0.8.19)

The current smart contracts use a floating pragma for the Solidity version (`pragma solidity ^0.8.19`). While this approach allows for compatibility with future minor releases, it introduces potential risks. Future updates might include changes and newly discovered bugs or vulnerabilities that could affect the contract's functionality and security. Locking the Solidity version would ensure the contract behaves consistently as tested and accounts for all known issues at that version.

Recommendation:
It is recommended to specify a fixed Solidity version for your contracts, such as `pragma solidity 0.8.19;`, instead of using `pragma solidity ^0.8.19;`. This will ensure that the contract operates as expected without unintended changes due to future compiler updates.

Comment: Client confirms to follow the recommendation to fix it in order to prevent any existing compiler-related issues.

Imprudent Withdrawal of Funds from Impaired Loans

Description: In the current implementation of the LoanManager::`withdrawFunds` function, there is no explicit check to prevent withdrawals from impaired loans. As a result, even if a loan has been impaired, which indicates a high-risk or non-performing status, the seller can still withdraw the drawable funds associated with that loan. This loophole undermines the purpose of impairing loans and can lead to significant financial losses.

Scenario:

1. A loan is created and funded, with the seller having drawable funds.
2. Due to some unforeseen circumstances, the loan is impaired, indicating that the loan is at high risk or is no longer performing as expected.
3. Despite the impairment, the seller calls the `withdrawFunds` function, withdrawing the funds associated with the impaired loan.
4. The platform or the pool suffers financial losses because the impaired loan was allowed to proceed with fund withdrawal, contrary to standard risk management practices.

Impact: Allowing withdrawals from impaired loans can lead to significant financial losses, undermining the integrity of the lending platform. The impairment status should act as a safeguard to prevent any further financial transactions on high-risk loans until the issues are resolved.

Recommendation:

Add Impairment Check:

- Modify the `withdrawFunds` function to include a check that prevents withdrawals if the loan is impaired

Client Comment :
In our protocol design, roles are separated to ensure that the seller can always withdraw funds, while any impairment primarily affects the lender. The issue description states that `LoanManager::withdrawFunds` prevents potential financial losses, but this is not entirely accurate. The Pool Admin can call `LoanManager::fundLoan`, transferring funds from the pool contract to the LoanManager contract. As a result, restricting `LoanManager::withdrawFunds` does not mitigate the impairment scenario.

	./ReceivableStorage.sol **./Pool.sol** **./WithdrawalManager.sol** **./libraries/types/DataTypes.sol** **./libraries/Errors.sol** **./libraries/PoolDeployer.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

	./libraries/upgradability/UUPSProxy.sol **./libraries/upgradability/VersionedInitializable.sol** **./Receivable.sol** **./PoolAddressesProvider.sol** **./abstracts/Governable.sol** **./PoolConfigurator.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

	./LoanManager.sol ./PoolConfiguratorStorage.sol ./LoanManagerStorage.sol ./WithdrawalManagerStorage.sol ./IsleGlobals.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 Isle Finance 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 Isle Finance team put in place a bug bounty program to encourage further analysis of the smart contract by third parties.

zokyo