QuillAudits

AUDIT REPORT

September 2025

For

Qiro

Table of Content

Executive Summary

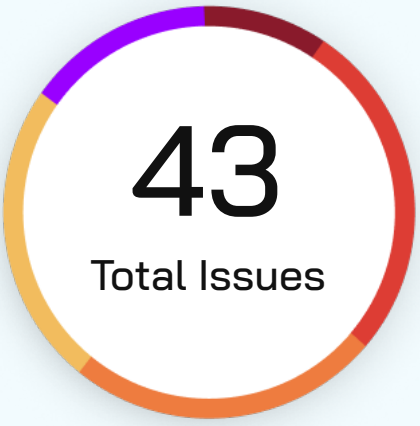
Project Name	Qiro
Protocol Type	Lending Borrowing
Project URL	https://www.qiro.fi/
Overview	Qiro is decentralized lending protocol that enables borrowing against underwritten NFT assets with flexible loan terms and tranched risk structures. The protocol supports both traditional loan pools and securitization products with customizable amortization schedules, single or dual tranche structures, and risk management features.
Audit Scope	The scope of this Audit was to analyze the Qiro Smart Contracts for quality, security, and correctness.
Source Code link	https://github.com/Qiro-Protocol/qiro-v1-foundry/tree/securitisation
Branch	securitisation
Contracts in Scope	src/borrower/QiroNFT.sol src/borrower/SecuritisationShelf.sol src/borrower/Deployer.sol src/borrower/Shelf.sol src/borrower/ShelfMath.sol src/common/qiro-math/InterestCalculator.sol src/common/qiro-math/Math.sol src/common/qiro-math/Interests.sol src/lender/Deployer.sol src/lender/distributor/Default.sol src/lender/distributor/SecuritisationDistributor.sol src/lender/tranche/operator/WhitelistEvents.sol src/lender/tranche/operator/Base.sol src/lender/tranche/operator/Whitelist.sol src/lender/tranche/operator/InvestmentOperator.sol src/lender/tranche/operator/RedemptionCalculator.sol src/lender/tranche/Reserve.sol src/lender/tranche/SeniorTranche.sol src/lender/tranche/SecuritisationReserve.sol src/lender/tranche/SecuritisationTranche.sol src/lender/tranche/TranchePool.sol src/lender/Root.sol

	src/lender/assessor/Default.sol src/lender/assessor/Base.sol src/DeployerFactory.sol src/QiroFactory.sol src/interfaces/IERC20Qiro.sol src/interfaces/IShelf.sol src/interfaces/ITranche.sol src/interfaces/IDistributor.sol src/interfaces/IAuth.sol src/interfaces/IAssessor.sol src/interfaces/ISecuritisationDefaultDistributor.sol src/interfaces/IRitualConsumer.sol src/interfaces/IQiroFactory.sol src/interfaces/IWhitelistOperator.sol src/interfaces/IDepend.sol src/interfaces/ISecuritisationTranche.sol src/common/qiro-math/Interests.sol src/common/qiro-math/InterestCalculator.sol src/common/qiro-math/Math.sol
Commit Hash	ae68bc514dd7118e42b5d48da99760482c853505
Language	Solidity
Blockchain	EVM
Method	Manual Analysis, Functional Testing, Automated Testing
Review 1	17th July 2025 - 3rd September 2025
Updated Code Received	15th September 2025 and 25th September 2025
Review 2	15th September 2025 - 26th September 2025
Fixed In	dbaf2d1b47e61a0c459121f3b241ac29aa5355c9 Branch : master-v1

Verify the Authenticity of Report on QuillAudits Leaderboard:

https://www.quillaudits.com/leaderboard

Number of Issues per Severity



Critical	4 (9.3%)
High	12 (27.9%)
Medium	10 (23.3%)
Low	10 (23.3%)
Informational	9 (20.9%)

Severity

Issues

	Critical	High	Medium	Low	Informational
Open	0	0	0	0	0
Acknowledged	0	0	0	0	0
Partially Resolved	0	0	0	0	0
Resolved	**4**	**12**	**10**	**10**	**7**

Summary of Issues

Issue No.	Issue Title	Severity	Status
1	Denial of Service in balance() Due to Strict Equality Check	Critical	Resolved
2	payOriginatorFee() Can Be Called Before Pool Initialization, Bypassing Fee Logic	Critical	Resolved
3	Unrestricted NFT Burn Function Bricks Entire Protocol	Critical	Resolved
4	Balance Manipulation DOS Attack	Critical	Resolved
5	Current Not Updated on Final Junior Supply Reaching Ceiling in InvestmentOperator:supplyJunior()	High	Resolved
6	Reentrancy Vulnerability via _safeMint in QiroNFT.sol	High	Resolved
7	Permanent Denial of Service in init_borrow() Due to Fragile Equality Check	High	Resolved
8	Missing Access Control in balance() Enables Unauthorized Servicer Fee Draining	High	Resolved
9	Unsafe Use of transferFrom Will Break on Non-Standard Tokens	High	Resolved

Issue No.	Issue Title	Severity	Status
10	Persistent token approval can cause unauthorized fund drainage	High	Resolved
11	Premature NFT Unlocking with Outstanding Interest	High	Resolved
12	isTransferAllowed Check Can Be Bypassed in transferFrom Function	High	Resolved
13	Flawed State Transition Bug Can Lock Reserve Withdrawals	High	Resolved
14	Griefing Attack on State Transition Leads to Permanent Lock of User Funds in	High	Resolved
15	repayFromReserve will double charge users	High	Resolved
16	SecuritisationShelf.payout will incorrectly revert	High	Resolved
17	Reserve Contract Created with Zero Address Operator	Medium	Resolved
18	Missing Consumer Removal Mechanism Causes Permanent System Compromise	Medium	Resolved
19	Insufficient Writeoff Time Allows Premature Loan Writeoffs	Medium	Resolved
20	Redundant High-Cost Function Calls in Redemption	Medium	Resolved

Issue No.	Issue Title	Severity	Status
21	Asymmetric redemption conditions cause unfair exits and stuck funds for senior investors	Medium	Resolved
22	DoS risk from unbounded amortization loop	Medium	Resolved
23	Incomplete Validations in Pool Creation Can Brick the System	Medium	Resolved
24	Non-standard ERC20 Tokens (e.g., USDT) Get Permanently Stuck Due to Unsafe transfer Handling in erc20Transfer	Medium	Resolved
25	Denial-of-Service via Inconsistent suppliedCurrency Tracking in TranchePool	Medium	Resolved
26	Unsafe Usage of ERC20 approve for currency	Medium	Resolved
27	Missing Input Validation in NFT Data Updates	Low	Resolved
28	Missing Validations at various functions in QiroNFT.sol	Low	Resolved
29	Missing Validations at various functions in Shelf .sol	Low	Resolved
30	Missing Validations at various functions in QiroFactory.sol	Low	Resolved

Issue No.	Issue Title	Severity	Status
31	Metadata Inconsistency Between Shelf and NFT Principal Amounts	Low	Resolved
32	Incomplete Parameter Migration in File Function	Low	Resolved
33	Incorrect Error Message for Incomplete Late Fee Payment	Low	Resolved
34	Previous Shelf Approval Not Revoked	Low	Resolved
35	Old Consumer Contract Remains Authorized	Low	Resolved
36	Unused Approval Grants Fee Collector Unnecessary Transfer Rights	Low	Resolved
37	Use of Magic Numbers and Magic Strings across Contracts	Informational	Resolved
38	Anti-Pattern - Unnecessary Self-Approval Before Transfer	Informational	Resolved
39	Missing Event Emissions in Critical State-Changing Functions	Informational	Resolved
40	Redundant require check in supplySenior	Informational	Resolved
41	Unused Modifier and Dead Code	Informational	Resolved

Issue No.	Issue Title	Severity	Status
42	Inverted / incorrect state check in withdraw (shelf branch)	Informational	Resolved
43	Redundant SafeMath Library in Solidity ⩾0.8.0	Informational	Resolved

Checked Vulnerabilities

- Access Management
- Arbitrary write to storage
- Centralization of control
- Ether theft
- Improper or missing events
- Logical issues and flaws
- Arithmetic Computations Correctness
- Race conditions/front running
- SWC Registry
- Re-entrancy
- Timestamp Dependence
- Gas Limit and Loops
- Exception Disorder
- Gasless Send
- Use of tx.origin
- Malicious libraries
- Compiler version not fixed
- Address hardcoded
- Divide before multiply
- Integer overflow/underflow
- ERC's conformance
- Dangerous strict equalities
- Tautology or contradiction
- Return values of low-level calls

- Missing Zero Address Validation
- Private modifier
- Revert/require functions
- Multiple Sends
- Using suicide
- Using delegatecall
- Upgradeable safety
- Using throw
- Using inline assembly
- Style guide violation
- Unsafe type inference
- Implicit visibility level

Techniques and Methods

Throughout the audit of smart contracts, care was taken to ensure:

- The overall quality of code
- Use of best practices
- Code documentation and comments, match logic and expected behavior
- Token distribution and calculations are as per the intended behavior mentioned in the whitepaper
- Implementation of ERC standards
- Efficient use of gas
- Code is safe from re-entrancy and other vulnerabilities

The following techniques, methods, and tools were used to review all the smart contracts:

Structural Analysis

In this step, we have analyzed the design patterns and structure of smart contracts. A thorough check was done to ensure the smart contract is structured in a way that will not result in future problems.

Static Analysis

A static Analysis of Smart Contracts was done to identify contract vulnerabilities. In this step, a series of automated tools are used to test the security of smart contracts.

Code Review / Manual Analysis

Manual Analysis or review of code was done to identify new vulnerabilities or verify the vulnerabilities found during the static analysis. Contracts were completely manually analyzed, their logic was checked and compared with the one described in the whitepaper. Besides, the results of the automated analysis were manually verified.

Gas Consumption

In this step, we have checked the behavior of smart contracts in production. Checks were done to know how much gas gets consumed and the possibilities of optimization of code to reduce gas consumption.

Tools and Platforms Used for Audit

Remix IDE, Foundry, Solhint, Mythril, Slither, Solidity Static Analysis.

Types of Severity

Every issue in this report has been assigned to a severity level. There are five levels of severity, and each of them has been explained below.

Critical: Immediate and Catastrophic Impact

Critical issues are the ones that an attacker could exploit with relative ease, potentially leading to an immediate and complete loss of user funds, a total takeover of the protocol's functionality, or other catastrophic failures. Critical vulnerabilities are non-negotiable; they absolutely must be fixed.

High (H): Significant Risk of Major Loss or Compromise

High-severity issues represent serious weaknesses that could result in significant financial losses for users, major malfunctions within the protocol, or substantial compromise of its intended operations. While exploiting these vulnerabilities might require specific conditions to be met or a moderate level of technical skill, the potential damage is considerable. These findings are critical and should be addressed and resolved thoroughly before the contract is put into the Mainnet.

Medium (M): Potential for Moderate Harm Under Specific Circumstances

Medium-severity bugs are loopholes in the protocol that could lead to moderate financial losses or partial disruptions of the protocol's intended behavior. However, exploiting these vulnerabilities typically requires more specific and less common conditions to occur, and the overall impact is generally lower compared to high or critical issues. While not as immediately threatening, it's still highly recommended to address these findings to enhance the contract's robustness and prevent potential problems down the line.

Low (L): Minor Imperfections with Limited Repercussions

Low-severity issues are essentially minor imperfections in the smart contract that have a limited impact on user funds or the core functionality of the protocol. Exploiting these would usually require very specific and unlikely scenarios and would yield minimal gain for an attacker. While these findings don't pose an immediate threat, addressing them when feasible can contribute to a more polished and well-maintained codebase.

Informational (I): Opportunities for Improvement, Not Immediate Risks

Informational findings aren't security vulnerabilities in the traditional sense. Instead, they highlight areas related to the clarity and efficiency of the code, gas optimization, the quality of documentation, or adherence to best development practices. These findings don't represent any immediate risk to the security or functionality of the contract but offer valuable insights for improving its overall quality and maintainability. Addressing these is optional but often beneficial for long-term health and clarity.

Types of Issues

Open

Security vulnerabilities identified that must be resolved and are currently unresolved.

Resolved

These are the issues identified in the initial audit and have been successfully fixed.

Acknowledged

Vulnerabilities which have been acknowledged but are yet to be resolved.

Partially Resolved

Considerable efforts have been invested to reduce the risk/impact of the security issue, but are not completely resolved.

Severity Matrix

Impact

Likelihood	High	Medium	Low
High	Critical	High	Medium
Medium	High	Medium	Low
Low	Medium	Low	Low

Impact

- **High** - leads to a significant material loss of assets in the protocol or significantly harms a group of users.
- **Medium** - only a small amount of funds can be lost (such as leakage of value) or a core functionality of the protocol is affected.
- **Low** - can lead to any kind of unexpected behavior with some of the protocol's functionalities that's not so critical.

Likelihood

- High - attack path is possible with reasonable assumptions that mimic on-chain conditions, and the cost of the attack is relatively low compared to the amount of funds that can be stolen or lost.
- Medium - only a conditionally incentivized attack vector, but still relatively likely.
- Low - has too many or too unlikely assumptions or requires a significant stake by the attacker with little or no incentive.

Critical Severity Issues

Denial of Service in balance() Due to Strict Equality Check

Resolved

Path

SecuritisationDistributor.sol

Function Name

`balance()`

Description

The balance() function enforces a strict equality invariant when validating repayment amounts:

```
require(
    currencyAmount == _principalAmount + _interestAmount + _prepaymentPrincipalAmount,
    CommonErrors.CurrencyAmountMismatch()
);
```

Here, currencyAmount is obtained from the contract's token balance. However, tokens can be freely transferred to the contract by any external actor. This makes currencyAmount unreliable as an invariant check.

An attacker (or even a user) can send 1 wei of the repayment currency to the contract before balance() transaction is executed. This increases currencyAmount slightly, making the equality check fail, and causing all repayment attempts to revert.

This creates a Denial of Service (DoS) vulnerability where no one can repay, as long as the contract's balance is easily manipulated.

Impact

Complete denial of repayments due to small dust transfers.
Attack costs are negligible (sending 1 wei token)

Likelihood

High. Anyone can perform this

Recommendation

Use >= instead of ==

payOriginatorFee() Can Be Called Before Pool Initialization, Bypassing Fee Logic

Resolved

Path

SecuritisationShelf.sol

Function Name

`payOriginatorFee()`

Description

The payOriginatorFee() function allows a borrower to manually pay the originator fee. However, this function lacks a check to ensure that the borrow has been initialized via init_borrow().

As a result, a borrower can call payOriginatorFee() prematurely—before any loan principal has been set, since principalAmount remains 0 by default until initialized. This causes:

- feeToTake to be calculated as 0
- No actual transfer to happen
- originatorFeePaid to be set to true permanently

This means that when the loan is later initialized and an actual fee should be paid, the system incorrectly considers the fee already paid, thereby bypassing fee enforcement.

Impact

- Fee logic is permanently broken for the loan
- Causes protocol revenue loss
- Borrower can avoid paying any originator fee by calling payOriginatorFee() early
- Could lead to unexpected behavior or inconsistencies in downstream accounting

Likelihood

High.

Recommendation

- Add a check to prevent execution when the loan has not been initialized. Either:
- Use the existing poolInitialized modifier, or
- Add an explicit condition:
- if (principalAmount == 0) revert ShelfErrors.ZeroPrincipal();
- This mirrors the validation already used in takeOriginatorFee().

QuillAudits' Team Comment

resolved by adding poolInitialized modifier

Unrestricted NFT Burn Function Bricks Entire Protocol

Resolved

Path

src/borrower/QiroNFT.sol:178

Function Name

`burn()`

Description

The burn() function lacks proper access controls, allowing anyone to burn any NFT token regardless of ownership or approval status. This vulnerability completely breaks the protocol's core functionality by making all critical operations revert.

Impact

When an NFT is burned, the nftLocked() function in Shelf contracts reverts because of

```
function nftLocked() public view returns (bool) {
    return assetNFT.ownerOf(tokenId) == address(this);
}
```

All Core Operations Break:

1. Borrowing: borrow() reverts due to nftLocked() failure
2. Repayments: repay() reverts due to nftLocked() failure
3. Unlocking: unlock() reverts due to nftLocked() failure

Likelihood

High.

Recommendation

We recommend adding proper access controls on burn() function

QuillAudits' Team Comment

resolved by checking msg.sender is ownerOf

Balance Manipulation DOS Attack

Resolved

Path

SecuritizationShelf

Function Name

`TakeOriginatorFee()`

Description

The takeOriginatorFee() function contains a strict balance equality check that can be exploited by any external actor to permanently disable fee collection functionality:

```
function takeOriginatorFee() internal whenNotPaused(poolId) returns (uint256) {
    // … fee calculation and transfer logic …

    balance = MathQ.safeSub(balance, feeToTake);
    originatorFeePaid = true;
    originatorFeePaidAmount = feeToTake;

    // VULNERABLE LINE: Strict equality check
    require(currency.balanceOf(address(this)) == balance, "balance mismatch");

    return feeToTake;
}
```

Contract's actual token balance can always higher than expected due to the fact the anyone can freely transfer funds directly to the contract at any point.
The strict equality check currency.balanceOf(address(this)) == balance fails

The vulnerability stems from a flawed assumption that the contract's token balance should always exactly equal the internal balance accounting variable. This creates a Balance Manipulation problem:

The check relies on external state (token balance) that can be manipulated compared to an internal balance that might not be updated before the function call; Uses== instead of >=, making it fragile to any external deposits

Recommendation

Consider the best approach to addressing this from several solutions, e.g:
Remove Strict Equality

```
require(currency.balanceOf(address(this)) >= balance, "insufficient balance");
```

Or
Handle External Deposits Gracefully

```
uint256 actualBalance = currency.balanceOf(address(this));
if (actualBalance > balance) {
    // Treat excess as donation to protocol
    balance = actualBalance;
}
require(actualBalance >= balance, "insufficient balance");
```

High Severity Issues

Current Not Updated on Final Junior Supply Reaching Ceiling in InvestmentOperator:supplyJunior()

Resolved

Path

src/lender/tranche/operator/InvestmentOperator.sol

Function

`supplyJunior(uint256 currencyAmount, address user)`

Description

A state update bug exists in supplyJuniorthat prevents the protocol from immediately switching to senior tranche supply once the junior tranche ceiling is reached. After executing a supply, the function checks whether the ceiling is full:

```
if (suppliedCurrencyJunior == juniorTrancheCeiling) {
    whitelistOperator.updateCurrent(2);
}
```

However, the suppliedCurrencyJunior value is read before the new supply is executed. This means that when the junior tranche becomes exactly full after the supply, the condition will not trigger.

As a result, updateCurrent(2) is skipped, and the system temporarily remains in an inconsistent state:

- The junior tranche is fully filled.
- But current still indicates junior supply is ongoing.

Code Snippet

src/lender/tranche/operator/InvestmentOperator.sol:supplyJunior()

```
    function supplyJunior(uint256 currencyAmount, address user) external
onlyWhitelistOperator nonReentrant {
        address junior = whitelistOperator.junior();
        address senior = whitelistOperator.senior();
        address currency = whitelistOperator.currency();
        uint256 juniorTrancheCeiling =
whitelistOperator.juniorTrancheCeiling();
        uint256 suppliedCurrencyJunior = ITranche(junior).suppliedCurrency();

        require(currencyAmount > 0, ShelfErrors.ZeroSupplyAmount());
        require(suppliedCurrencyJunior < juniorTrancheCeiling,
ShelfErrors.JuniorTrancheFull());
        require(currencyAmount + suppliedCurrencyJunior <=
juniorTrancheCeiling, ShelfErrors.JuniorTrancheFull());

        // Call the internal supply function through whitelist operator
        whitelistOperator.executeSupply(currencyAmount, junior, user);

        // Check if junior tranche is now full and update current
        if (suppliedCurrencyJunior == juniorTrancheCeiling) { //@audit
suppliedCurrencyJunior did not updated for this supply
            whitelistOperator.updateCurrent(2);
        }
        ...
```

And also...

Nested nonReentrant Calls Break Core Whitelist::supply()

Path

src/lender/tranche/operator/Whitelist.sol

Function

`supply(uint256 currencyAmount)`

Description

The WhitelistOperator.supply() function is the main entrypoint for investors to deposit currency into the protocol during the capital formation period. This function internally routes deposits either to supplyJunior() or supplySenior() depending on tranche capacity.
However, all three functions (supply, supplyJunior, and supplySenior) are protected by the nonReentrant modifier from ReentrancyGuard.

This results in an unavoidable reentrancy conflict:
1. A user calls supply().
2. nonReentrant guard marks _status = ENTERED.
3. Inside supply(), the function calls supplyJunior() (or supplySenior()).
4. Because these are also marked nonReentrant, they immediately detect _status = ENTERED and revert with ReentrancyGuardReentrantCall().

As a result, all calls to supply() will always revert, making it impossible for investors to deposit into the protocol. This permanently blocks capital formation and renders the pool unusable.

Since the supply method is important in that the current value is set in this method, and considering the previous bug that I reported that current is not updated unless it is updated in the next deposit in this method, although the supplyJunior and supplySenior methods can also be called directly, the supply method has some logic that must be executed if the junior ceiling is filled.

Code Snippet

Entry Function with Nested Call:

Whitelist.sol#L343

```
function supply(uint256 currencyAmount)
    external
    whenNotPaused(poolId)
    nonReentrant
    isCapitalFormationPeriod
{
    if (ITranche(junior).suppliedCurrency() + currencyAmount <=
juniorTrancheCeiling) {
        supplyJunior(currencyAmount); // ← calls another nonReentrant
    } else {
        if (current == 1) {
            current = 2;
        }
        supplySenior(currencyAmount, msg.sender); // ← calls another
nonReentrant
    }
}
```

Nested Callees (also nonReentrant)

```
function supplyJunior(uint256 currencyAmount)
    public
    whenNotPaused(poolId)
    nonReentrant // ← conflict here
    auth_junior_investor
    isJuniorNotFull(currencyAmount)
    isCapitalFormationPeriod
{
    investmentOperator.supplyJunior(currencyAmount, msg.sender);
}

function supplySenior(uint256 currencyAmount, address receiver)
    public
    whenNotPaused(poolId)
    nonReentrant // ← conflict here
    auth_senior_investor
    isCapitalFormationPeriod
{
    investmentOperator.supplySenior(currencyAmount, receiver);
}
```

Recommendation

• The condition should use the post-supply value instead of the pre-supply snapshot:

```
if (suppliedCurrencyJunior + currencyAmount == juniorTrancheCeiling) {
    whitelistOperator.updateCurrent(2);
}
```

This ensures the protocol updates current exactly when the junior tranche reaches capacity.

- Remove nonReentrant from supplyJunior and supplySenior, keeping it only on the top-level supply entrypoint.
- Alternatively, make supplyJunior and supplySenior internal/private helpers, ensuring they can't be called directly by external users, and apply reentrancy protection only once at the outermost layer.

https://github.com/Qiro-Protocol/qiro-v1-foundry/pull/62

Reentrancy Vulnerability via _safeMint in QiroNFT.sol

Resolved

Path

QiroNFT.sol

Function

`mint()`

Description

The mint() function in the Qiro NFT contract calls _safeMint(to, tokenId) before revoking the caller's minter rights (minters[msg.sender] = false). This is dangerous because _safeMint can trigger a reentrancy via ERC721Receiver's onERC721Received() callback if the recipient is a contract.

During this callback, the malicious contract could call mint() again and bypass the minters[msg.sender] = false check, since the minter role has not yet been revoked. This allows multiple unauthorized NFTs to be minted in the same transaction, potentially draining the protocol or minting unbacked positions.

Impact

- A malicious contract with minter access can reenter before their minter privilege is revoked.
- This can result in unlimited unauthorized NFT mints.

Likelihood

Medium

Recommendation

Move the minter revocation line above the _safeMint() call or best to call _safeMint() at the end of the function to ensure that the caller's privileges are removed before any external call is made

Permanent Denial of Service in init_borrow() Due to Fragile Equality Check

Resolved

Path

SecuritisationShelf.sol

Description

The SecuritisationShelf.init_borrow() function performs a strict equality check between the NFT's loan principal (assetNFT.loanPrincipal(nftTokenId)) and the pool balance (distributor.totalBalanceInTranches()):

```
require(
    assetNFT.loanPrincipal(nftTokenId) == distributor.totalBalanceInTranches(),
    "NFTPrincipalNotSameAsPoolBalance"
);
```

However, totalBalanceInTranches() computes the current currency.balanceOf of the tranche contracts:

```
function totalBalanceInTranches() public view returns (uint256) {
    return currency.balanceOf(address(senior)) + currency.balanceOf(address(junior));
}

```

This introduces a critical vulnerability: any external actor can permanently break this equality by sending a small amount of tokens to either tranche contract (senior or junior). Since ERC20.balanceOf() simply reflects the actual token balance of the contract, any unexpected transfer will break the exact match required in init_borrow() and cause it to revert.

Additionally, the function is non-reentrant and guarded by poolNotInitialized, so the failure of this check blocks the only entry point to initialize borrowing — effectively creating a permanent Denial of Service (DoS).

Impact

Permanent denial of borrowing: An attacker can frontrun or grief the system by sending a trivial amount of tokens (e.g., 1 wei) to any tranche contract, breaking the equality and causing init_borrow() to revert forever.

Likelihood

High. Anyone can perform this

Recommendation

Replace the strict equality with a less-than-or-equal (<=) comparison to allow borrowing when the pool holds at least the required principal:

```
require(
assetNFT.loanPrincipal(nftTokenId) <= distributor.totalBalanceInTranches(),
"NFTPrincipalExceedsPoolBalance"
);
```

Optionally, emit a warning if excess funds are detected, or return the surplus to depositors.

QuillAudit's Team comment

resolved, but error message can be improved. "NFTPrincipalNotSameAsPoolBalance" should be replaced with "not enough balance" etc

Missing Access Control in balance() Enables Unauthorized Servicer Fee Draining

Resolved

Path

SecuritisationDistributor.sol

Description

The balance() function in the Distributor contract is publicly callable and lacks any access control, allowing any external address to invoke it and trigger fund flows — including the transfer of arbitrary servicer fee amounts to the designated fee collector:

```
function balance(
    uint256 _principalAmount,
    uint256 _interestAmount,
    uint256 _prepaymentPrincipalAmount,
    uint256 interestShortfallAmount,
    uint256 principalShortfallAmount,
    uint256 servicerFeeAmount
) external {
    ...
    _repayTranches(
        _principalAmount,
        _interestAmount,
        _prepaymentPrincipalAmount,
        interestShortfallAmount,
        principalShortfallAmount,
        servicerFeeAmount
    );
}
```

Within _repayTranches(), the servicer fee amount is unconditionally transferred from the contract to the servicer fee collector:

```
if (servicerFeeAmount > 0) {
    // send servicer fee to reserve
    require(
        currency.transfer(shelf.servicerFeeCollector(), servicerFeeAmount),
        CommonErrors.CurrencyTransferFailed()
    );
}
```

Since the contract does not verify the caller, an attacker can call balance() with any servicerFeeAmount and force the protocol to send that amount to the fee collector address when in repayment state.

Impact

Loss of funds to the protocol

Likelihood

High

Recommendation

Restrict access to balance() e.g use auth

QuillAudit's Team comment

resolved by adding auth modfier and depend autmotically adds new shelf address to wards

Unsafe Use of transferFrom Will Break on Non-Standard Tokens

Resolved

Description

Throughout the entire codebase there are multiple unsafe use of transferFrom such as this:

```
require(currency.transferFrom(address(this), address(shelf), totalAmount), "currency-transfer-failed");
```

Even the withdrawal function is an example of bad implementation as well.

In the withdraw() function of the SecuritisationShelf contract, token transfer is performed using:

```
bool success = currency.transferFrom(address(this), usr, currencyAmount);
```

So in general, these assume that transferFrom returns a boolean, which is not true for all ERC20 tokens. For example, USDT does not return a value, and calling this with such a token would revert due to an invalid return decoding, even if the transfer succeeds.

Although the current deployment may use a standard-compliant token, future use of non-standard tokens like USDT would cause withdrawal failures.

Impact

Permanently blocks funds for users if token behavior is not compatible.

Likelihood

High. Qiro team confirmed it to be high

Recommendation

Use OpenZeppelin's SafeERC20.safeTransfer instead, which gracefully handles non-compliant tokens:

```
SafeERC20.safeTransfer(currency, usr, currencyAmount);
```

Alternatively, if transferFrom is intended, ensure compatibility with expected token behaviors

QuillAudit's Team comment

simple text search on transferFrom shows that every instance of currency.transferFrom is using safeTransferFrom. which fixes the issue

Persistent token approval can cause unauthorized fund drainage

Resolved

Path
SecuritisationShelf.sol

Function
`takeOriginatorFee()`

Description
The takeOriginatorFee() function grants token approval to the fee collector for transferring fees but does not revoke this approval afterward. This leaves a persistent token approval in place that can be abused.

Impact
Persistent token approval can cause unauthorized fund drainage if the fee collector is compromised or malicious.

Likelihood
High

Recommendation
Revoke the token approval immediately after the transfer is completed. A safer alternative is to avoid persistent allowances altogether by using safeTransfer() to directly transfer the required amount.

QuillAudit's Team comment
resolved, direct transfers are used

Premature NFT Unlocking with Outstanding Interest

Resolved

Path

src/borrower/Shelf.sol

Function

`writeOff(uint256 fromPrincipal)`

Description

The writeOff() function prematurely unlocks and returns the NFT collateral to the borrower when only the principal amount is written off, even if outstanding interest remains unpaid. The function checks getOutstandingPrincipal() == 0 to determine loan closure, but ignores outstanding interest obligations.

```
function writeOff(uint256 fromPrincipal) public poolInitialized auth_admin
whenNotPaused nonReentrant {
     ...
     require(fromPrincipal==getOutstandingPrincipal(),
ShelfErrors.WriteoffMismatchOutstandingPrincipal());//@audit no partial
writeoff


       ...


        totalWriteOffAmount += fromPrincipal;


        // close loan if debt is zero after writeoff
        if (getOutstandingPrincipal() == 0) {
            // Mark loan as ended
            loanEnded = true;


            // Unlock and transfer NFT back to borrower
            _unlockNFT(); //@audit


                }
    }
```

Impact

1. Outstanding interest becomes unrecoverable once NFT is returned
2. Loan marked as “ended” while debt still exists

Likelihood

Medium

Recommendation

Replace the principal-only check with total debt before _unlockNFT()

Intended for the design

isTransferAllowed Check Can Be Bypassed in transferFrom Function

Resolved

Path

src/common/token/ERC20.sol:60

Function

`transferFrom()`

Description

The QiroERC20 contract implements a transfer restriction mechanism using the isTransferAllowed boolean flag. While the transfer() function properly checks this flag before allowing transfers, the transferFrom() function bypasses this restriction entirely. In src/common/token/ERC20.sol:

```
   function transferFrom(address from, address to, uint256 amount) public
override auth returns (bool) {
       return super.transferFrom(from, to, amount);     }
```

The transferFrom() function only requires the auth modifier (which checks if the caller is authorized via the wards mapping) but completely ignores the isTransferAllowed flag that is checked in the transfer() function:

```
require(isTransferAllowed, "QERC20/transfers-not-allowed");
```

Impact

When isTransferAllowed is set to false (the default state), authorized contracts can still transfer tokens using transferFrom(), completely circumventing the intended transfer restriction.

As a side note, The auth modifier on transferFrom() completely breaks standard ERC20 composability. For example following standard ERC20 usage patterns can be broken:

- DEX Trading: Most DEXs (Uniswap, SushiSwap, Curve, etc.) rely on transferFrom() to move tokens during trades.
- Lending Protocols: Protocols like Aave, Compound rely on transferFrom() for deposits.

Likelihood

Medium

Recommendation

We recommend implementing the below check in transferFrom

```
require(isTransferAllowed, "QERC20/transfers-not-allowed");
```

Flawed State Transition Bug Can Lock Reserve Withdrawals

Resolved

Path

src/lender/tranche/operator/Whitelist.sol

Function

`updateState()`

Description

The pool state machine does not allow transition from RepaymentStarted to Redeem.

```
        if (shelf.loanEnded()) {
            if (state == State.Active || state == State.Pending) { //@audit the
state is not changed to redeem if repayment state was there
                state = State.Redeem;
            }
            if (state == State.Redeem && IERC20(currency).balanceOf(senior) +
IERC20(currency).balanceOf(junior) == 0) {
                state = State.Ended;
            }
        }
```

Once repayments begin, the state becomes RepaymentStarted. When the loan ends, this condition excludes RepaymentStarted, meaning the system never reaches Redeem. Since Ended can only be reached from Redeem, the pool lifecycle becomes permanently stuck.

While tranche investors can still redeem in RepaymentStarted (as the redemption flow only checks that state is not Active or Pending), reserve withdrawals remain impossible. Both Reserve.sol and SecuritisationReserve.sol require the pool to be in Ended (or Revoked) before allowing withdrawals.

```
function withdraw(uint256 amount) external nonZeroAmount(amount) {
        ...
        } else {
            uint256 state = operator.getState();
            require(state == uint256(State.Revoked) || state ==
uint256(State.Ended), OnlyEndedAndRevokedStateAllowed());
            // Distributor can withdraw
            require(balances[msg.sender] >= amount, InsufficientBalance());
            balances[msg.sender] -= amount;
            currency.safeTransfer(msg.sender, amount);
            // Emit the withdraw event
            emit Withdraw(msg.sender, amount);
        }
    }
```

Because the pool never reaches Ended, reserve funds are permanently locked, leaving only the SuperAdmin able to access them.

Impact

Reserve funds permanently locked for distributors and investors

Likelihood

High

Recommendation

Update the loan ending condition to also handle RepaymentStarted:

Griefing Attack on State Transition Leads to Permanent Lock of User Funds in

Resolved

Path

src/lender/tranche/reserve/Reserve.sol

Function

`withdraw(uint256 amount)`

Description

A denial-of-service vulnerability exists that allows a malicious actor to permanently freeze funds deposited by users into a pool's Reserve contract. The attack leverages a flawed state transition check in the WhitelistOperator contract, which can be easily griefed.

The protocol's pool state machine is designed to transition from Redeem to Ended only when all funds have been withdrawn from the tranches. The WhitelistOperator.updateState() function enforces this by checking if the currency balance of both the senior and junior tranche contracts is exactly zero.

A malicious actor can deliberately break this condition by sending a "dust" amount (e.g., 1 wei) of the pool's currency directly to either the SeniorTranche or JuniorTranche contract address.

This action prevents the combined balance from ever reaching zero, causing the pool to be permanently stuck in the Redeem state.

The Reserve.sol contract's withdraw function allows regular users (i.e., not the Super Admin or the Shelf contract) to retrieve their supplied funds only when the pool is in the Revoked or Ended state. Because the griefing attack prevents the pool from ever reaching the Ended state, any user who supplied funds to the reserve will be permanently unable to withdraw them, leading to a direct loss of funds.

Code Snippet

1. The Root Cause (The Griefable Check in Whitelist.sol):

src/lender/tranche/operator/Whitelist.sol

```
function updateState() public whenNotPaused(poolId) {
        uint256 totalRequiredCapital = juniorTrancheCeiling + seniorTrancheCeiling;
        uint256 totalInvested = totalDepositCurrencyJunior + totalDepositCurrencySenior;
        uint256 shelfLoanStartTimestamp = shelf.LOAN_START_TIMESTAMP();

        // Calculate the threshold requirement (e.g., 70% of totalRequiredCapital)

        if (state == State.CapitalFormation) {
            // If capital formation period is over
            if (block.timestamp > capitalFormationEnd) {
                uint256 thresholdAmount = (totalRequiredCapital * threshold) / THRESHOLD_MAX;
                if (totalInvested >= thresholdAmount) {
                    // If the threshold is met, pool is considered Active
                    state = State.Pending;
                } else {
                    // If threshold is not met, revoke the pool
                    state = State.Revoked;
                }
            }
            if (
                shelfLoanStartTimestamp == 0 // borrower not init
                    && block.timestamp < capitalFormationEnd // under capital formation
                    && totalDepositCurrencyJunior + totalDepositCurrencySenior
                        >= seniorTrancheCeiling + juniorTrancheCeiling
            ) {
                state = State.Pending;
            }
        }

        if (shelfLoanStartTimestamp != 0) {
            // borrow init done
            if (
                state == State.Pending || state == State.CapitalFormation // pending
            ) {
                state = State.Active;
            }
            if (
                state == State.Active // active
                    && shelf.totalRepayedAmount() != 0 // active
            ) {
                state = State.RepaymentStarted;
            }
        }

        if (shelf.loanEnded()) {
            if (state == State.RepaymentStarted) {
                state = State.Redeem;
            }
            if (state == State.Redeem && IERC20(currency).balanceOf(senior) + IERC20(currency).balanceOf(junior) == 0) {
                state = State.Ended;
            }
        }

        WhitelistEvents.emitUpdatedPoolState(poolId, state);
    }
```

2. The Vulnerable Function (The Strict Check in Reserve.sol):

src/lender/tranche/reserve/Reserve.sol

```
function withdraw(uint256 amount) external nonZeroAmount(amount) whenNotPaused(poolId) {
        require(currency.balanceOf(address(this)) >= amount, AmountMoreThanBalance());
        if (msg.sender == shelf) {
            uint256 state = IWhitelistOperator(IShelf(shelf).lender()).getState();
            // Shelf can withdraw anytime during non-active states
            require(
                state == uint256(State.Active) || state == uint256(State.RepaymentStarted), OnlyNonActiveStateAllowed()
            );
            currency.safeTransfer(shelf, amount);
            // Emit the withdraw event
            emit Withdraw(shelf, amount);
        } else {
            uint256 state = IWhitelistOperator(IShelf(shelf).lender()).getState();
            require(state == uint256(State.Revoked) || state == uint256(State.Ended), OnlyEndedAndRevokedStateAllowed());
            // Distributor can withdraw
            require(balances[msg.sender] >= amount, InsufficientBalance());
            balances[msg.sender] -= amount;
            currency.safeTransfer(msg.sender, amount);
            // Emit the withdraw event
            emit Withdraw(msg.sender, amount);
        }
    }
```

Impact

High (Leads to a permanent, irrecoverable loss of user funds).

Likelihood

High (The attack is simple and cheap to execute for a malicious actor).

Recommendation

The most robust solution is to fix the root cause in the WhitelistOperator.updateState() function. The state transition logic should not depend on the raw token balance of the tranche contracts, as this is an unreliable and gameable metric.Or a simpler, but less ideal, fix would be to allow user withdrawals from the reserve during the Redeem state. This would involve modifying therequire statement in Reserve.sol:

```
// In src/lender/tranche/reserve/Reserve.sol
require(
    state == uint256(State.Revoked) ||
    state == uint256(State.Ended) ||
    state == uint256(State.Redeem), // <-- Add this condition
    OnlyEndedAndRevokedStateAllowed()
);
```

repayFromReserve will double charge users

Resolved

Path

src/borrower/Shelf

Function

`Shelf.repayFromReserve`

Description

Shelf.repayFromReserve will repay a portion of a borrower's debt through the reserve. This is done by calling reserve.withdraw. The issue is that after withdrawing the assets from the reserve repay will be called, which will also transfer the withdrawn assets from the borrower. Essentially, the same assets will be withdrawn from both the reserve and the borrower, leading to a loss for the borrower.

```
1  // Reserve transfers funds to Shelf
2        reserve.withdraw(amountToRepay); // Assuming Reserve has this function
3
4        repay(amountToRepay);
5    }
```

Impact

The impact is High as borrowers will be double-charged for repaying their debts, rendering the function unusuable.

Likelihood

The likelihood will be Frequent as the issue will always occur when the function is called.

Recommendation

Do not use repay when withdrawing from the reserve. Simply perform the necessary state changes, without charging the borrower.

SecuritisationShelf.payout will incorrectly revert

Resolved

Path

src/borrower/SecuritisationShelf

Function

`SecuritisationShelf.payout`

Description

SecuritisationShelf.payout users specify the principal and interest they want to repay and also the arrearsPrincipal and arrearsInterest which are amounts to be deposited into the reserve. These amounts are deducted from principal and interest and not actually used to repay the loan. The issue is that in the beginning of payout it is validated that interest and principal do not surpass the current outstanding interest and principal:

```
1  if (principal + prePaymentPrincipal > getOutstandingPrincipal()) {
2              revert ShelfErrors.ExceedsTotalPrincipal();
3          }
4          if (interest > getOutstandingInterest()) {
5              revert ShelfErrors.ExceedsTotalInterest();
6          }
```

This is problematic as principal and interest include the arrears principal and interest, which are not used to decrease the outstanding principal and interest. Therefore, as principal and interest are higher than the actual amounts used to repay the loan the validation may fail incorrectly. For example, the current outstanding principal is 1000. A user wants to repay 900 principal and deposit 200 into the reserve, therefore, they set principal to 1100. Nowthis will cause the validation to revert, even though the actual repayed principal is less than the outstanding principal.

```
1  if (principal + prePaymentPrincipal > getOutstandingPrincipal()) {
2              revert ShelfErrors.ExceedsTotalPrincipal();
3          }
4          if (interest > getOutstandingInterest()) {
5              revert ShelfErrors.ExceedsTotalInterest();
6          }
```

When validating whether the interest and principal are higher than the reserves decrease the values by the arrearsPrincipal and arrearsInterest.

Impact

The impact will be High as repayments will fail to occur.

Likelihood

The likelihood will be Rare, as the issue only occur in certain scenarios described below.

Medium Severity Issues

Reserve Contract Created with Zero Address Operator

Resolved

Path

src/borrower/Shelf.sol

Function Name

`_initializeContract`

Description

During shelf deployment, the Reserve contract is created with address(0) as the lender/operator, permanently breaking lender rights on the reserve. When the lender is later set on the shelf via the depend() function in Root.sol, this change does not propagate to the already-created Reserve contract, leaving the Reserve's operator set to the zero address.

Likelihood

Medium

Recommendation

We recommend initializing the reserve contract with proper operator address

Missing Consumer Removal Mechanism Causes Permanent System Compromise

Resolved

Path

src/borrower/QiroNFT.sol

Description

The QiroNFT contract lacks any mechanism to remove malicious or compromised consumer contracts from the isConsumerContract mapping. Once a consumer is added via addConsumerContract() or depend(), it becomes permanently authorized to call updateDataForTokenID() and update critical NFT risk data.

Impact

One-time compromise = Permanent compromise: Legitimate consumer gets compromised, no way to revoke access

Likelihood

Low

Recommendation

We recommend adding Consumer Removal Function

Insufficient Writeoff Time Allows Premature Loan Writeoffs

Resolved

Path

src/borrower/Shelf.sol:86

Function Name

`State variable: writeOffTime`

Description

The writeOffTime is set to only 1 hour, which is extremely insufficient for proper due diligence and loan validation. The TODO comment indicates this should be 90 days, highlighting a critical gap between current implementation and intended security measures.

Impact

Malicious borrowers may exploit the short writeOffTime by delaying repayment for just 1 hour to trigger a writeoff, which subsequently unlocks the NFT collateral. While this scenario assumes off-chain services automatically execute writeoffs, their behavior is out of scope for this evaluation. If the off-chain service does indeed process writeoffs automatically, the severity would be critical. In this assessment, we assign a Medium severity due to uncertainty regarding the off-chain service.

Likelihood

High

Recommendation

increase writeOffTime to 90 days as indicated in the TODO comment

Redundant High-Cost Function Calls in Redemption

Resolved

Path

src/lender/assessor/Base.sol

Function Name

`_calcJuniorDebtWithoutWriteoff()`

Description

The _calcJuniorDebtWithoutWriteoff() function calls pool.totalInterestForLoanTerm() on every redemption request, which is a high gas-consuming function. Since loan terms are calculated at deployment time and remain constant throughout the loan lifecycle, this value can be cached to avoid repeated expensive calculations.

Impact

Every redemption request incurs unnecessary gas overhead

Likelihood

High

Recommendation

Cache the totalInterestForLoanTerm value to avoid repeated calculations

QuillAudit's Team comment

resolved by storing and reading the laonTermInterest once after init_borrow

Asymmetric redemption conditions cause unfair exits and stuck funds for senior investors

Resolved

Path

src/lender/tranche/operator/InvestmentOperator.sol

Function Name

`redeemJunior()`

Description

A vulnerability exists in the redemption mechanism where junior tranche holders can redeem their tokens after capital formation ends, while senior tranche holders remain locked. This occurs if an admin increases the junior tranche ceiling during the Pending state, after capital formation but before loan initialization.

Because the junior redemption path uses a ceiling-based condition:

```
function redeemJunior(uint256 tokenAmount, address user) external
onlyWhitelistOperator nonReentrant {
   …
 if (juniorTrancheCurrencyBalance < juniorTrancheCeiling || state ==
State.Revoked || state == State.Redeem
                || state == State.RepaymentStarted || state == State.Ended
        ) {
  }
```

while the senior redemption path relies on a state-based condition:

```
function redeemSenior(uint256 tokenAmount, address user) external
onlyWhitelistOperator nonReentrant {
        …


        if (state != State.Active && state != State.Pending) {
     …
          }
```

the mismatch creates an unfair economic advantage for juniors. If the admin raises the junior ceiling using:

```
function setTrancheCeilings(uint256 amount, string memory tranche) public
whenNotPaused auth_admin {
        if (keccak256(abi.encodePacked(tranche)) == JUNIOR) {
            juniorTrancheCeiling = amount; //@audit if the tranche ceiling is
set after the capital formation period
        } else if (keccak256(abi.encodePacked(tranche)) == SENIOR) {
            seniorTrancheCeiling = amount;
        } else {
            revert("unknown tranche");
        }
    }
```

then juniors may exit, but seniors remain locked. Once juniors redeem, the pool balance no longer matches the loan principal, causing loan initialization to fail due to strict validation (due to not enough pool balance):

```
function init_borrow(address _borrower, uint256 nftTokenId, bool
takeFeeFromPrincipalOrNot)
        external
        whenNotPaused
        poolNotInitialized
        auth_admin
        nonReentrant
    {
      …
        require(
            assetNFT.loanPrincipal(nftTokenId) ==
distributor.totalBalanceInTranches(),
            "NFTPrincipalNotSameAsPoolBalance"
        ); //@audit strict equality check can brick the contract;
```

Impact

This leads to unfair exits for juniors, stuck funds for seniors, and a deadlock in loan origination.

Likelihood

Low

Recommendation

Admins should not be able to change the ceilings after capital formation is completed

QuillAudit's Team comment

but why senior and junior redemption conditions are not equal. they are equivalent somehow but code readability is messed up

DoS risk from unbounded amortization loop

Resolved

Path

src/borrower/ShelfMath.sol

Function Name

`calculateTotalInterestForAmortisation`

Description

The calculateTotalInterestForAmortisation function loops over the full repayment schedule without an upper bound.

```
 function calculateTotalInterestForAmortisation(
       uint256 _principalAmount,
       uint256 _periodCount,
       uint256 _periodLength,
       uint256 _aprInBasisPoints,
       uint256 _pStartFrom,
       uint256 _pRepayFrequency
   ) public pure returns (uint256) {
       uint256 accumulatedInterest = 0;
       for (uint256 I = 0; I < _periodCount; I++) { //@audit unbounded loop
           (, uint256 periodInterest) = calculatePeriodPayment(
               I + 1, _principalAmount, _aprInBasisPoints, _periodLength,
_periodCount, _pStartFrom, _pRepayFrequency
           );
           accumulatedInterest += periodInterest;
       }
       return accumulatedInterest;
   }
```

This function is executed both at loan initialization and later during redemption (redeemSenior() and related flows). The risk emerges because a loan may be created successfully when its gas usage is just under the block limit, but redemption re-runs the same loop and may exceed the available gas.

Scenario timeline

1.Loan creation
   a)A loan with very fine repayment intervals and a high _periodCount is initialized.
   b)The transaction consumes close to the maximum block gas but still succeeds.
2. Redemption
   a)Later, a user calls redeemSenior().
   b)The call propagates into calcLoanTermInterest() –> calculateTotalInterestForAmortisation().

The same loop now requires more gas than the block allows (either due to crafted parameters or a reduction in block gas limits of the network).

Impact

Redemption always reverts with out-of-gas.

Likelihood

LOW, due to nondeterministic available gas limits, and network changing block gas limits

Recommendation

Cap the number of repayment periods

Incomplete Validations in Pool Creation Can Brick the System

Resolved

Path

src/QiroFactory.sol

Function Name

`deployPool()`

Description

The deployPool function currently performs only partial parameter checks when creating a pool. Specifically, it only ensures that the senior interest rate does not exceed the shelf rate, and in case of a single tranche, that both rates match:

```
function deployPool(
        Pool memory _pool,
       ...
    ) internal {
        // Ensure seniorRate does not exceed interestRatePerSecond //@audit if
shelf interst rate is more than lender expectdd interest rate, where does the
excess money go?  to underwriter? I think it goes to junior tranche
        require(params.lenderParams[0] <= params.shelfParams[0], "senior
interest rate exceeds financing fee");
        if (params.lenderParams[1] == 0) {
            require(
                params.lenderParams[0] == params.shelfParams[0],
                "for single tranche, both financing fee and senior rate should
be same"
            );
        }


        // Validate that performanceFee and originatorFee are within acceptable
limits
        // Ensure that both fees are between 0.01% (1 basis point) and 100%
(10000 basis points) //@audit validatio should be sume of both fees can not
exceed to 10000
        require(params.fee[0] >= 0 && params.fee[0] <= 10000, "performanceFee
fee out of bounds");
        require(params.fee[1] >= 0 && params.fee[1] <= 10000, "originator fee
out of bounds");


        // both ceiling can't be 0, and senior can never be zero
        require(params.lenderParams[2] > 100, "senior ceiling can't be less than
100");
```

However, two critical validations are missing:

1. Interest rate alignment: The sum of senior and junior rates is not validated against the overall shelf interest rate, allowing inconsistencies in interest accrual.
2. Ceilings vs principal amount: There is no check ensuring that the combined tranche ceilings are at least as large as the principal amount of the loan. This can create a pool where the intended loan cannot be funded.

Impact

1. Deadlocked pools: A pool may enter the Pending state via the ceilings, but fail to initialize the loan (due to ceilings are not enough to cover the funds), leaving capital permanently locked.
2. System bricking: Once deployed with invalid parameters, the pool cannot be recovered without governance or manual intervention, effectively bricking that instance.

Likelihood

LOW, due to admin controlled params

Recommendation

Expand the validation logic in deployPool to enforce:

1. The sum of tranche interest rates must equal the shelf interest rate.
   The combined tranche ceilings must be at least equal to the loan's principal amount.
2. The threshold percentage, when applied to total ceilings, must still meet or exceed the principal amount.

Non-standard ERC20 Tokens (e.g., USDT) Get Permanently Stuck Due to Unsafe transfer Handling in erc20Transfer

Resolved

Path

Tranche.sol

Function Name

`erc20Transfer`

Description

The erc20Transfer function uses IERC20Qiro(erc20).transfer(...) with a boolean return check. However, not all ERC20 implementations return a boolean (e.g., USDT on Ethereum). If such tokens are accidentally sent to the Tranche contract, recovery attempts via this function will always revert, permanently locking funds.

```
1  function erc20Transfer(address erc20, address usr, uint256 amount)
2      public
3      nonZeroAddress(erc20)
4      nonZeroAddress(usr)
5      nonZeroAmount(amount)
6      auth
7      whenNotPaused(poolId)
8  {
9      // Restrict erc20Transfer() to exclude main protocol tokens
10     require(erc20 != address(token) && erc20 != address(currency), ForbiddenTokenTransfer());
11
12 @>    require(IERC20Qiro(erc20).transfer(usr, amount), Erc20TransferFailed());
13
14     emit Erc20Transfer(erc20, usr, amount);
15 }
```

Impact

Non-standard ERC20 tokens (e.g., USDT, BNB, some legacy tokens) cannot be recovered from the contract, leading to permanent loss of mistakenly sent tokens.

Recommendation

UseOpenZeppelin's SafeERC20.safeTransfer instead of directly calling transfer. SafeERC20 handles both standard and non-standard ERC20 implementations safely.

Denial-of-Service via Inconsistent suppliedCurrency Tracking in TranchePool

Resolved

Path

src/lender/tranche/TranchePool.sol

Function Name

```
supply(address usr, uint256 currencyAmount, uint256 tokenAmount)
redeem(address usr, uint256 currencyAmount, uint256 tokenAmount)
```

Description

A critical Denial-of-Service (DoS) vulnerability exists due to inconsistent accounting of suppliedCurrency in TranchePool.

The function supply() correctly increments suppliedCurrency when funds are deposited. However, in the corresponding redeem() function, suppliedCurrency is never decremented when users withdraw their funds.

This mismatch creates a permanent inconsistency between:

- the actual currency balance in the tranche contracts, and
- the logical ceiling check enforced in InvestmentOperator.

Because supplyJunior() relies on suppliedCurrency (not actual balance) to enforce the juniorTrancheCeiling, while supplySenior() checks the actual ERC20 balance of the junior tranche, the system becomes deadlocked after a normal cycle of deposit + redemption.

Exploitation Scenario

Consider the following parameters:

Junior tranche ceiling: 1000
Step 1: Deposit near the ceiling
A junior investor deposits 999.

suppliedCurrency = 999
junior.balance = 999
Step 2: Redeem funds
The investor redeems 999.

junior.balance = 0

suppliedCurrency = 999 (❌ not decremented)
Step 3: Further deposits

Any deposit >1 reverts because suppliedCurrency + amount > 1000.
Only exactly 1 can be deposited successfully.
Step 4: Senior deposits
When the junior ceiling is logically “full,” further deposits should route to the senior tranche.

However, supplySenior() requires:

```
require(IERC20(currency).balanceOf(address(junior)) >= juniorTrancheCeiling && currencyAmount >= 1000, ...);
```

Since the actual junior balance is far below the ceiling, this condition always fails.

Result: No senior deposits are possible either.

✅ Outcome: Both junior and senior tranches are permanently locked. The protocol is effectively halted (DoS).

Proof of Concept (PoC)

The following Foundry test demonstrates the issue:
Add this script to the file: test\lender\Operator.t.sol

`import {ITranche} from "src/interfaces/ITranche.sol";`

```
function test_JuniorCeiling_SupplyAfterRedeem_DoS_POC() public {
    // whitelist junior investor
    vm.startPrank(deployment.admin);
    mainOperator.relyInvestor(juniorInvestor, "junior");
    factory.addMember(juniorInvestor);
    mainOperator.relyInvestor(seniorInvestor, "senior");
    factory.addMember(seniorInvestor);
    vm.stopPrank();

    ITranche juniorTr = ITranche(deployment.juniorTranche);
    ITranche seniorTr = ITranche(deployment.seniorTranche);

    // Step 1: initial supply 100
    uint256 firstSupply = 100e6;
    vm.startPrank(juniorInvestor);
    currency.approve(deployment.juniorTranche, firstSupply);
    mainOperator.supply(firstSupply);
    vm.stopPrank();

    console.log("--------     initial supply 100       --------");
    console.log("Step 1: JUNIOR_CEILING:", JUNIOR_CEILING);
    console.log("Step 1: suppliedCurrency:", juniorTr.suppliedCurrency());
    console.log("Step 1: junior balance:", currency.balanceOf(deployment.juniorTranche));

    // Step 2: supply 899 more => suppliedCurrency becomes 999
    uint256 secondSupply = 899e6;
    vm.startPrank(juniorInvestor);
    currency.approve(deployment.juniorTranche, secondSupply);
    mainOperator.supply(secondSupply);
    vm.stopPrank();

    console.log("--------     supply 899 more       --------");
    console.log("Step 2: suppliedCurrency:", juniorTr.suppliedCurrency());
    console.log("Step 2: junior balance:", currency.balanceOf(deployment.juniorTranche));

    // Step 3: redeem 899 -> balance returns to 100, suppliedCurrency stays 999
    vm.startPrank(juniorInvestor);
    IERC20(mainOperator.juniorToken()).approve(deployment.juniorTranche, secondSupply);
    mainOperator.redeemJunior(secondSupply);
    vm.stopPrank();

    console.log("--------     redeem 899       --------");
    console.log("Step 3: suppliedCurrency:", juniorTr.suppliedCurrency());
    console.log("Step 3: junior balance:", currency.balanceOf(deployment.juniorTranche));

    // Step 4: attempt to supply 2 -> expect revert
    uint256 failSupply = 2e6;
    vm.startPrank(juniorInvestor);
    currency.approve(deployment.juniorTranche, failSupply);
    vm.expectRevert();
    mainOperator.supply(failSupply);
    vm.stopPrank();

    console.log("--------     attempt to supply 2 -> expect revert      --------");
    console.log("Step 4: suppliedCurrency:", juniorTr.suppliedCurrency());
    console.log("Step 4: junior balance:", currency.balanceOf(deployment.juniorTranche));
```

Run Test with : forge test -vv —mt “test_JuniorCeiling_SupplyAfterRedeem_DoS_POC”
Console logs confirm the state at each step, demonstrating the DoS condition:

```
[PASS] test_JuniorCeiling_SupplyAfterRedeem_DoS_POC() (gas: 1016207)
Logs:
  --------     initial supply 100       --------
  Step 1: JUNIOR_CEILING: 1000000000
  Step 1: suppliedCurrency: 100000000
  Step 1: junior balance: 100000000
  --------     supply 899 more       --------
  Step 2: suppliedCurrency: 999000000
  Step 2: junior balance: 999000000
  --------     redeem 899       --------
  Step 3: suppliedCurrency: 999000000
  Step 3: junior balance: 100000000
  --------     attempt to supply 2 -> expect revert      --------
  Step 4: suppliedCurrency: 999000000
  Step 4: junior balance: 100000000
  --------     attempt larger supply also reverts      --------
  Step 5: suppliedCurrency: 999000000
  Step 5: junior balance: 100000000
  --------     only supply 1 (should succeed)   ceiling reached    --------
  Step 6: suppliedCurrency: 1000000000
  Step 6: junior balance: 101000000
  --------     after ceiling reached, any further supply goes to senior  reverts called by junior   --------
  Step 7: suppliedCurrency (junior): 1000000000
  Step 7: junior balance: 101000000
  Step 7: senior balance: 0
  --------  DoS   after ceiling reached, any further supply goes to senior called by senior and reverted  --------
  Step 7: suppliedCurrency (junior): 1000000000
  Step 7: suppliedCurrency (senior): 0
  Step 7: junior balance: 101000000
  Step 7: senior balance: 0

Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 13.47ms (3.43ms CPU time)

Ran 1 test suite in 23.19ms (13.47ms CPU time): 1 tests passed, 0 failed, 0 skipped (1 total tests)
```

Impact

1. Permanent DoS: After one large redemption, no further deposits can be made into either tranche.
2. Protocol Halt: All new capital formation is blocked.
3. User Losses: Investors are unable to continue normal participation, and senior tranche investors are blocked completely.

Recommendation

Update suppliedCurrency to reflect the actual token balance of the contract instead of incrementing manually. For example, in TranchePool.sol:supply() replace:

```
suppliedCurrency = suppliedCurrency + actualAmount;
```

with:

```
suppliedCurrency = currency.balanceOf(address(this)) + actualAmount;
```

In TranchePool.sol:redeem(), ensure that suppliedCurrency is updated when the state is CapitalFormation. For example:

```
if (state == State.CapitalFormation) {
    suppliedCurrency = currency.balanceOf(address(this)) - actualAmount;
}
```

Unsafe Usage of ERC20 approve for currency

Resolved

Path & Function

all locations where currency.approve is used should be listed here — can be extracted via grep -R "currency.approve(" src/

Description

The contract directly uses currency.approve(spender, amount) without checking the return value. This is unsafe because:

- Silent failures: Some tokens (e.g., DAI-like variants) return false instead of reverting. Ignoring the return value may lead to allowance not being applied while execution continues.
- Non-standard behavior: Tokens such as USDT revert if the allowance is updated without first setting it to 0. Direct usage of approve breaks compatibility.
- Denial of Service: If approval fails silently or reverts, any logic depending on token transfers will be blocked.

This unsafe usage affects all functions in which approve is called without validation.

Code Snippet (Vulnerable)

```
currency.approve(spender, amount);
```

Impact

Medium — Approval may silently fail or revert, causing denial of service in core flows (e.g., transfers, repayment, borrowing).

Likelihood

Medium — Certain for non-standard tokens like USDT

Recommendation

Replace all direct calls to approve with OpenZeppelin's SafeERC20.forceApprove, which handles non-standard ERC20 implementations safely.

```
1  import "@openzeppelin/contracts/token/ERC20/utils/SafeERC20.sol";
2  using SafeERC20 for IERC20Qiro;
3
4  currency.forceApprove(spender, amount);
```

This guarantees that if the token does not allow direct allowance updates, the approval will first reset to 0 and then set the intended amount, preventing silent failures and ensuring compatibility across standard and non-standard tokens.

Low Severity Issues

Missing Input Validation in NFT Data Updates

Resolved

Path

src/borrower/QiroNFT.sol

Function Name

`_initializeContract`

Description

The updateDataForTokenID() function lacks proper input validation, allowing updates to non-existent tokens and accepting invalid data values. This can lead to data inconsistencies and confusion in the system.

Key Issues:

- No validation that tokenId exists (can update data for tokenId = 0 or non-existent tokens)
- No validation of inferenceResults array length or data ranges
- No validation of data field values (prob_of_default, risk_score, etc.)

Impact

1. Phantom data: Risk data can be created for non-existent tokens
2. Data overwrites: Existing risk assessments can be silently replaced
3. Invalid values: Risk scores and probabilities can be set to impossible values
4. Array bounds: No protection against array index out of bounds
5. Pool Creation: Pool can be created for non-existent tokens due to only isUnderwritten check.

Likelihood

Medium

Recommendation

Validate all the inputs

Qiro Team's comment

this data comes from our underwriting infra and the values and their ranges might vary on each nft underwriting and the ranges therefore are not defined.

Missing Validations at various functions in QiroNFT.sol

Resolved

Path

src/borrower/QiroNFT.sol

Function Name

`Multiple Functions in contract`

Description

The QiroNFTcontract has multiple functions that lack proper input validation, creating potential issues with data integrity and system behavior. These functions accept inputs without validating their correctness or existence.

Functions with Missing Validation:

- updateConsumerNFTFeed() - No access control
- setMinter() - No duplicate minter check
- mint() - Missing validation on portfolio data ranges
- depend() - No zero address or contract validation
- burn() - No access control, anyone can burn NFTs (reported as separate issue due to high severity)
- setArweave() - No tokenId existence validation
- addConsumerContract() - No zero address or duplicate checks
- resetUnderwritten() - Redundant no-op logic
- updateDataForTokenID() - No tokenId or data validation
- file() - No tokenId existence or value validation

Likelihood

Medium

Recommendation

Add comprehensive input validation to all functions

Qiro Team's comment

mint portfolio data comes from off chain infra and limits of those input data comes from external parties and are therefore not defined, therefore we cannot set validations. the ranges might keep changing

Missing Validations at various functions in Shelf .sol

Resolved

Path

src/shelf.sol

Description

The Shelf contract has multiple functions that lack proper input validation, creating potential issues with data integrity and system behavior. These functions accept inputs without validating their correctness or existence.

Functions with Missing Validation:

• Constructor - Missing validation on upper caps of annualInterestRateInBps and zero address check on borrower
• depend() function - Missing validation on contract addresses
• _initializeContract() - Missing validation on pStartFrom parameter

Likelihood

Medium

Recommendation

Add comprehensive input validation to all functions

Missing Validations at various functions in QiroFactory.sol

Resolved

Path

src/borrower/QiroFactory.sol

Description

The Shelf contract has multiple functions that lack proper input validation, creating potential issues with data integrity and system behavior. These functions accept inputs without validating their correctness or existence

Functions with Missing Validation:

• setContracts() - No zero address checks for any of the 7 contract addresses
• file() - No zero address validation for _value parameter
• setCreatePoolAccess() - No zero address check for user parameter
• updateWhitelistManager() - No zero address check for _whitelistManager
• addMember() - No zero address check for address_ parameter
• removeMember() - No zero address check for address_ parameter
• addMemberBulk() - No zero address checks in the loop
• removeMemberBulk() - No zero address checks in the loop
• changePoolAdmin() - No zero address check for newAdmin
• relyContractFromRoot() - No zero address checks for contract_ or usr
• denyContractFromRoot() - No zero address checks for contract_ or usr
• relyContract() - No zero address checks for contract_ or usr
• denyContract() - No zero address checks for contract_ or usr
• Total: 13 functions with missing validation issues.
• Note: The constructor also has missing input validation for all 4 parameters (_currency, _qiroFeeCollector, _qiroConsumer, _qiroAssetNFT)

Likelihood

Medium

Recommendation

Add comprehensive input validation to all functions

Metadata Inconsistency Between Shelf and NFT Principal Amounts

Resolved

Path

src/borrower/QiroNFT.sol

Function Name

`file()`

Description

The file() function in QiroNFTallows updating the loanPrincipal amount for an NFT,but this creates an inconsistency with the totalPrincipalAmount stored in the NFT'smetadata. The metadata remains unchanged while the loan principal is updated, leading to data inconsistency.

```
function file(bytes32 what, uint256 tokenId, uint256 _value) external auth {

        if (what == "principal") {
            loanPrincipal[tokenId] = _value; //@audit this can cause
`Metadata`'s `totalPrincipalAmount` go out of sync
            emit File(what, tokenId, _value);
        } else {
            revert("unknown parameter");
        }
    }
```

1. loanPrincipal[tokenId] gets updated to new value
2. tokenIdToData[tokenId].totalPrincipalAmount remains unchanged
3. Two different principal amounts exist for the same NFT

Impact

1. Conflicting information: NFT shows different principal amounts in different places
2.Metadata mismatch: NFT metadata doesn't reflect actual loan principal

Likelihood

Medium

Recommendation

Consider making principal amounts immutable after loan initiation to prevent such inconsistencies.

Incomplete Parameter Migration in File Function

Resolved

Path

src/borrower/Shelf.sol

Function Name

`file()`

Description

The file() function allows updating the annualInterestRateInBps parameter but fails to recalculate dependent loan parameters. When the interest rate changes, the total loan amount to repay should also be updated to reflect the new rate, but this recalculation is missing.

Likelihood

High

Recommendation

Add automatic recalculation when interest rate changes

Incorrect Error Message for Incomplete Late Fee Payment

Resolved

Description

The _processComponentWisePayment() function uses the wrong error message when late fee payment is incomplete. The function throws IncompleteLateFeePaymentForBulletLoan() error for all loan types, but this error should only be used for bullet loans, not normal amortized loans.

```
            for (uint256 p = processRepaymentFromPeriod; p <= timeBasedPeriod;
p++) {
                if (p <= periodCount) {
                    uint256 periodLateFee = _calculatePeriodLateFee(p,
outstandingPrincipal);

                    if (isBulletRepay && periodLateFee > payment) {
                        revert
ShelfErrors.IncompleteLateFeePaymentForBulletLoan();
                    }
                    if (periodLateFee > 0 && payment > 0) {
                       ...


                        // ensure full payment of late fees for each period
                        if (paidLateFee < periodLateFee) {
                            revert
ShelfErrors.IncompleteLateFeePaymentForBulletLoan(); //@audit, wrong error.
Instead it should be late payment form normal loans.
                        }
                    }
                }
            }
```

Impact

Incorrect troubleshooting: Users may try to fix bullet loan issues when they have amortized loans

Likelihood

High

Recommendation

Create separate error messages for different loan types

Previous Shelf Approval Not Revoked

Resolved

Path

SecuritisationDistributor.sol

Description

In the depend() function of the SecuritisationDistributor contract, when updating the shelf address, the contract calls currency.approve(newShelf, type(uint256).max) to grant unlimited token allowance to the new shelf.

However, it does not revoke the allowance previously granted to the old shelf address. As a result, the old shelf contract retains approval rights, which can lead to unintended token transfers if the old contract becomes compromised or misused.

Impact

- The previously approved shelf retains unlimited allowance.
- Breaks the principle of least privilege.
- Introduces potential risk in the future.

Likelihood

Low

Recommendation

Before setting approval for the new shelf, revoke the allowance granted to the old one:
currency.approve(address(shelf), 0);
This ensures only the currently active shelf has token transfer rights.

Old Consumer Contract Remains Authorized

Resolved

Path

QiroNFT.sol

Description

The depend(address addr) function updates the qiroConsumer reference and sets the new address as authorized in the isConsumerContract mapping. However, it does not revoke authorization from the previous consumer contract, leaving it permanently marked as authorized.

Even though access is controlled via auth, this could unintentionally lead to multiple consumer contracts retaining access indefinitely.

Impact

Legacy consumer contracts remain authoriz ed in isConsumerContract, potentially leading to multiple access point.
Reduces clarity and control over which contracts are currently valid consumers.

Likelihood

Low

Recommendation

Update the function to revoke authoriz ation from the previous consumer contract:

```
function depend(address addr) public auth {
  address oldConsumer = address(qiroConsumer);
  isConsumerContract[oldConsumer] = false;
  qiroConsumer = IRitualConsumer(addr);
  isConsumerContract[addr] = true;
  emit ConsumerContractUpdated(oldConsumer, addr);
}
```

Unused Approval Grants Fee Collector Unnecessary Transfer Rights

Resolved

Path

SecuritisationShelf.sol

Function Name

`takeOriginatorFee()`

Description

In the takeOriginatorFee() function of the SecuritisationShelf contract, the contract calls currency.approve(qiroFeeCollector, feeToTake) before performing an internal transferFrom to send the fee to the fee collector. However, this approve is not required, since the transfer is executed by the contract itself — not the fee collector.

This results in the fee collector being granted an unnecessary allowance of feeToTake tokens, which remains valid even after originatorFeePaid is set to true. If the feeCollector address is compromised or becomes malicious, it could call transferFrom to extract the fee again, leading to unintended fund outflows.

lthough the qiroFeeCollector is expected to be a trusted address, this grants persistent and redundant transfer rights, violating the principle of least privilege.

Context: SecuritisationShelf.sol::takeOriginatorFee()

currency.approve(qiroFeeCollector, feeToTake); // unnecessary

This line is not used anywhere afterward, and the transferFrom is performed directly by the contract.

Impact

Redundant token approval to the qiroFeeCollector.
Potential double withdrawal of fee amount via transferFrom by qiroFeeCollector.
Future integrations or address changes to qiroFeeCollector could pose risks if not tightly controlled.

Likelihood

Low

Recommendation

Remove the approve() call:
// currency.approve(qiroFeeCollector, feeToTake); // Remove this

Since the transfer is executed by the contract itself, this approval is unnecessary and potentially misleading.

Alternatively, if the approval is intended for compatibility with future upgrades, it should be revoked after the transfer to limit risk:

currency.approve(qiroFeeCollector, 0);

Informational Issues

Use of Magic Numbers and Magic Strings across Contracts

Resolved

Path

src/borrower/Shelf.sol:1065
src/borrower/SecuritisationShelf.sol:310
src/common/qiro-math/Interests.sol:127
src/borrower/Shelf.sol:875
src/borrower/Shelf.sol:256
src/QiroFactory.sol:236-237

Description

- Minimum principal amount
  Location: Shelf.sol:1065, SecuritisationShelf.sol:310
  The minimum principal amount is hardcoded as 100,000. This value is directly enforced in the principalAmount validation and reduces flexibility if protocol parameters change.
- Days in year assumption
  Location: Interests.sol:127
  The year length is hardcoded as 365 days in interest calculations, while common, it should be expressed via a constant for readability and maintainability.
- Fee rate division factor
  Location: Shelf.sol:875
  The denominator 10,000 is hardcoded when calculating fees, representing basis points. While common, it should be expressed via a constant for readability and maintainability.
- Magic strings for fee categories
  Location: Shelf.sol:256
  Fee category names “performanceFees” and “originatorFees” are hardcoded as string literals. This introduces fragility since strings are prone to typos and cannot be compiler-checked.
- Fee validation bounds
  Location: QiroFactory.sol:236–237
  Individual fees are validated against a hardcoded maximum of 10,000 basis points. However, combined validation is missing, allowing the sum of fees to exceed the intended cap.

Likelihood

Low

Recommendation

Replace hardcoded values with named constants or configuration parameters.

Anti-Pattern - Unnecessary Self-Approval Before Transfer

Resolved

Description

Several functions across the codebase unnecessarily call approve(address(this), ...) before executing transferFrom(address(this), ...). Since the contract already owns the tokens, this pattern is redundant. Direct transfer() calls should be used instead. In cases where no transfer follows, the approval is entirely unnecessary.

Instances found

- Shelf.sol:432–434: withdraw() uses approve() + transferFrom() instead of transfer().
- Shelf.sol:1084: borrow_init() approves type(uint256).max without a subsequent transfer.
- SecuritisationShelf.sol:680–682: withdraw() unnecessarily uses approve() + transferFrom().
- SecuritisationShelf.sol:329: borrow_init() sets type(uint256).max approval without transfer usage.
- Default.sol:84–85: withdraw() redundantly approves before transferFrom().
- SecuritisationDistributor.sol:126–127: withdraw() redundantly approves before transferFrom().
- SecuritisationTranche.sol:157; self approval in constructor

Impact

Gas inefficiency: Redundant approvals increase transaction costs without benefit.

Likelihood

High

Missing Event Emissions in Critical State-Changing Functions

Resolved

Path

This issue reduces transparency and weakens accountability within the protocol. Several critical functions update ceilings, thresholds, loan parameters, permissions, or investor access without emitting events. Withoutthese logs, off-chain monitoring tools, auditors, and users cannot reliably detect or reconstruct important state transitions

Instances found

- Ceiling changes (high impact)
- Whitelist.sol:182–190; setTrancheCeilings() updates tranche limits without emitting events.
- Threshold updates
- Whitelist.sol:403–405; setThreshold() modifies risk thresholds with no visibility.
- Supply permission changes
  Whitelist.sol:335–337; setSupplyAllowed() alters supply controls silently.
- State updates (public function)
  Whitelist.sol:407; updateState() allows state transitions, but intermediate updates lack events.
- Loan parameter updates
  Shelf.sol:266–278; file() modifies loan parameters such as interest rate, repayment structure, and period count without emitting events.
- Write-off operations
  Shelf.sol:299; writeOff() performs write-offs without logging the amounts.
- Transfer permission changes
  ERC20.sol:42–44; setTransferAllowed() silently disables/enables transfers.
- Investor whitelist changes
  Whitelist.sol:212–218; denyInvestor() alters investor access without emitting events.

Impact

Monitoring systems and external integrations cannot track protocol state accurately.

Likelihood

High

Recommendation

Emit missing events for important updates

QuillAudit's Team comment

few functions are removed like setTrancheCeilings is removed

Redundant require check in supplySenior

Resolved

Path

src/lender/tranche/operator/InvestmentOperator.sol

Function Name

`supplySenior()`

Description

The supplySenior function contains two consecutive require statements that validate the seniorSuppliedCurrency against the seniorTrancheCeiling.

```
function supplySenior(uint256 currencyAmount, address receiver) external
onlyWhitelistOperator nonReentrant {
    ...
require(seniorSuppliedCurrency<=seniorTrancheCeiling,ShelfErrors.SeniorTrancheF
ull());
require(currencyAmount+seniorSuppliedCurrency<=seniorTrancheCeiling,ShelfErrors
.SeniorTrancheFull());
    ...
    }
```

The first check is redundant, because if seniorSuppliedCurrency already exceeds the ceiling, the second condition will also fail since currencyAmount > 0. Thus, the second check alone is sufficient to enforce the intended invariant.

Likelihood

High

Recommendation

Remove the redundant first check and rely only on the second require statement:

Unused Modifier and Dead Code

Resolved

Path

src/lender/tranche/operator/Whitelist.sol

Function Name

`modifier isSupplyAllowed()`

Description

The isSupplyAllowed modifier and related supplyAllowed variable are completely unused in the codebase. All supply functions rely on the isCapitalFormationPeriod modifier for access control, making this code dead and creating unnecessary complexity.

Impact

Dead code increase the byte size during deployments

Likelihood

High

Recommendation

Remove the dead code

Inverted / incorrect state check in withdraw (shelf branch) Resolved

Path
src/ lender/ tranche/ Reserve.sol

Function Name
`withdraw`

Description
The withdraw function's branch for msg.sender == shelf contains a logic inversion. Right now, it allows withdrawals when the pool state is Active or RepaymentStarted — which are active states — while the comment and docs require that the shelf may withdraw only in non-active states. The check is reversed, so the shelf can withdraw at the wrong times.

```
function withdraw(uint256 amount) external nonZeroAmount(amount) whenNotPaused(poolId) {
        require(currency.balanceOf(address(this)) >= amount, AmountMoreThanBalance());
        if (msg.sender == shelf) {
            uint256 state = IWhitelistOperator(IShelf(shelf).lender()).getState();
            // Shelf can withdraw anytime during non-active states
            require(
                state == uint256(State.Active) || state == uint256(State.RepaymentStarted), OnlyNonActiveStateAllowed()   //@audit
            );
            currency.safeTransfer(shelf, amount);
            // Emit the withdraw event
            emit Withdraw(shelf, amount);
        } else {
            uint256 state = IWhitelistOperator(IShelf(shelf).lender()).getState();
            require(state == uint256(State.Revoked) || state == uint256(State.Ended), OnlyEndedAndRevokedStateAllowed());
            // Distributor can withdraw
            require(balances[msg.sender] >= amount, InsufficientBalance());
            balances[msg.sender] -= amount;
            currency.safeTransfer(msg.sender, amount);
            // Emit the withdraw event
            emit Withdraw(msg.sender, amount);
        }
    }
```

Attack Path
The contract grants the shelf the ability to transfer reserve funds out of the contract.

Because the check is inverted, the shelf can withdraw during active operation of the pool (or repayment), when funds should generally be locked for borrowers/lenders or used for other lifecycle operations.

This breaks protocol invariants and can allow premature or unauthorized draining of the reserve.

Recommendation
Option A — allow only specific non-active states
Option B — explicitly disallow active states

Qiro's Team comment
The comment and error message is incorrect and should be fixed, logic is correct, shelf should be able to withdraw during loan active and repayment started states to withdraw and repay loan incase there is a late payment, used in repayFromReserve function.

Redundant SafeMath Library in Solidity ≥0.8.0

Resolved

Path

src/common/qiro-math/Interests.sol

Description

The Interests.sol contract includes a custom SafeMath library with functions for safe addition, subtraction, multiplication, and division. However, the contract uses Solidity version >=0.8.19. Starting from Solidity 0.8.0, overflow and underflow checks are performed natively by the compiler. Therefore, this custom SafeMath library is redundant. Its presence increases contract deployment size and slightly raises gas costs for arithmetic operations compared to using native operators.

```
library SafeMath {
    function mul(uint256 a, uint256 b) internal pure returns (uint256) { ... }
    function div(uint256 a, uint256 b) internal pure returns (uint256) { ... }
    function sub(uint256 a, uint256 b) internal pure returns (uint256) { ... }
    function add(uint256 a, uint256 b) internal pure returns (uint256) { ... }
}

library Interest {
    using SafeMath for uint256;
    ...
}
```

Recommendation

- Remove the custom SafeMath library.
- Remove using SafeMath for uint256; directive.
- Replace all calls like a.mul(b) with the native a * b, a.add(b) with a + b, etc.

This change will reduce contract size, lower gas costs, and rely on Solidity's built-in arithmetic safety checks.

Functional Tests

Some of the tests performed are mentioned below:

- **Only minters can mint**
- **Only authorized can update shelf, tranches, or currency references**
- **Pool can only be initialized once with valid borrower, NFT ownership, and approval**
- **Fee can only be paid once, either upfront or from principal, cannot bypass payment**
- **Repayment cannot be blocked by small token transfers**
- **Currency approvals work correctly with new Shelf contract**

Automated Tests

No major issues were found. Some false positive errors were reported by the tools. All the other issues have been categorized above according to their level of severity.

Threat Model

Threat Modeling is a systematic process used to identify, analyze, and evaluate potential security risks within a protocol, helping to understand how threats could exploit vulnerabilities and to design effective mitigation strategies. The below table represents the threat model and not actual issues, all identified issues have been detailed in the sections above.

Contract	Function	Threats
QiroNFT	mint() / burn()	• Treasury hijacking (malicious mint to drain resources) • Griefing burns by compromised admin
SecuritisationShelf	init_borrow()	• Under-collateralized loan creation • Incorrect principal assignment affects all loan logic
	takeOriginatorFee()	• Unlimited approval to feeCollector (backdoor risk) • Potential permanent fee bypass
	payOriginatorFee()	Can be called before borrow initialized, permanently skipping fee
Tranches	deposit() / withdraw()	• Rounding errors allow dust balance lockups • Possible reentrancy on non-standard ERC20
BaseAssessor	depend()	• State update accuracy
Securitisation Distributor	balance()	• strict == check on repayment allows attacker to dust-transfer tokens, blocking all repayments

Risks and Limitations

- Limited Scope
  This audit focused on the Qiro codebase as provided at the time of review. While key contracts such as SecuritisationShelf, borrowers logic, QiroNFT, and the tranche logic were examined, it is possible that vulnerabilities exist in non-reviewed files, inherited libraries, or deployment scripts.
- Third-Party Dependencies
  Compatibility issues or unsafe assumptions about token standards can introduce vulnerabilities. Additionally, any security flaws in external libraries or dependencies (e.g., SafeMath-like utilities, ERC20 implementations) may propagate into this system.
- Integration Risks
  The protocol relies on external actors and systems, such as fee collectors, borrower wallets, and tranche depositors. Compromise or misbehavior of these entities may lead to unintended outcomes. This audit did not cover price oracle integrations, bridges, or other off-chain infrastructure that may be used alongside this system.
- Operational and Deployment Risks
  Deployment parameters such as borrower addresses, tranche ratios, fee amounts, and admin assignments must be configured correctly. Misconfiguration may lead to fund lockup, unfair distribution, or permanent denial-of-service. Operational security of multisig wallets, admin keys, and governance processes was not reviewed in this engagement.
- Trust Assumptions
  Certain contracts, such as the fee collector and admin roles, are implicitly trusted. If these parties behave maliciously, they can redirect funds, bypass protections, or disable critical functionality. The protocol assumes these roles are controlled securely.
- Network and Ecosystem Risks
  At the time of this audit, the broader ecosystem in which Qiro operates (chain stability, ERC20 compliance, validator behavior) has not been audited. Failures or exploits in the underlying blockchain or dependent infrastructure may impact the protocol regardless of smart contract correctness.

Closing Summary

In this report, we have considered the security of Qiro Contracts. We performed our audit according to the procedure described above.

During Audit,we found a lot of issues of Critical, High, Medium, Low and Informational Severity, Qiro team resolved all the issues mentioned.

Disclaimer

At QuillAudits, we have spent years helping projects strengthen their smart contract security. However, security is not a one-time event—threats evolve, and so do attack vectors. Our audit provides a security assessment based on the best industry practices at the time of review, identifying known vulnerabilities in the received smart contract source code.

This report does not serve as a security guarantee, investment advice, or an endorsement of any platform. It reflects our findings based on the provided code at the time of analysis and may no longer be relevant after any modifications. The presence of an audit does not imply that the contract is free of vulnerabilities or fully secure.

While we have conducted a thorough review, security is an ongoing process. We strongly recommend multiple independent audits, continuous monitoring, and a public bug bounty program to enhance resilience against emerging threats.

Stay proactive. Stay secure.

About QuillAudits

QuillAudits is a leading name in Web3 security, offering top-notch solutions to safeguard projects across DeFi, GameFi, NFT gaming, and all blockchain layers. With seven years of expertise, we've secured over 1400 projects globally, averting over $3 billion in losses. Our specialists rigorously audit smart contracts and ensure DApp safety on major platforms like Ethereum, BSC, Arbitrum, Algorand, Tron, Polygon, Polkadot, Fantom, NEAR, Solana, and others, guaranteeing your project's security with cutting-edge practices.

7+ Years of Expertise	**1M+** Lines of Code Audited
50+ Chains Supported	**1400+** Projects Secured

Follow Our Journey

AUDIT REPORT

September 2025

For

Qiro

QuillAudits

Canada, India, Singapore, UAE, UK

www.quillaudits.com audits@quillaudits.com