Atlendis Labs Loan Products

Security Assessment

April 18, 2023

Prepared for:
Stéphane Coquet
Atlendis Labs

Prepared by: **Nat Chin, Justin Jacob, Elvis Skozdopolj, Gustavo Grieco**

About Trail of Bits

Founded in 2012 and headquartered in New York, Trail of Bits provides technical security assessment and advisory services to some of the world's most targeted organizations. We combine high-end security research with a real-world attacker mentality to reduce risk and fortify code. With 100+ employees around the globe, we've helped secure critical software elements that support billions of end users, including Kubernetes and the Linux kernel.

We maintain an exhaustive list of publications at https://github.com/trailofbits/publications, with links to papers, presentations, public audit reports, and podcast appearances.

In recent years, Trail of Bits consultants have showcased cutting-edge research through presentations at CanSecWest, HCSS, Devcon, Empire Hacking, GrrCon, LangSec, NorthSec, the O'Reilly Security Conference, PyCon, REcon, Security BSides, and SummerCon.

We specialize in software testing and code review projects, supporting client organizations in the technology, defense, and finance industries, as well as government entities. Notable clients include HashiCorp, Google, Microsoft, Western Digital, and Zoom.

Trail of Bits also operates a center of excellence with regard to blockchain security. Notable projects include audits of Algorand, Bitcoin SV, Chainlink, Compound, Ethereum 2.0, MakerDAO, Matic, Uniswap, Web3, and Zcash.

To keep up to date with our latest news and announcements, please follow @trailofbits on Twitter and explore our public repositories at https://github.com/trailofbits. To engage us directly, visit our "Contact" page at https://www.trailofbits.com/contact, or email us at info@trailofbits.com.

Trail of Bits, Inc.
228 Park Ave S #80688
New York, NY 10003
https://www.trailofbits.com
info@trailofbits.com

Notices and Remarks

Copyright and Distribution

Test Coverage Disclaimer

All activities undertaken by Trail of Bits in association with this project were performed in accordance with a statement of work and agreed upon project plan.

Security assessment projects are time-boxed and often reliant on information that may be provided by a client, its affiliates, or its partners. As a result, the findings documented in this report should not be considered a comprehensive list of security issues, flaws, or defects in the target system or codebase.

Trail of Bits uses automated testing techniques to rapidly test the controls and security properties of software. These techniques augment our manual security review work, but each has its limitations: for example, a tool may not generate a random edge case that violates a property or may not fully complete its analysis during the allotted time. Their use is also limited by the time and resource constraints of a project.

Table of Contents

Executive Summary

Engagement Overview

Atlendis Labs engaged Trail of Bits to review the security of its Loans and Revolving Credit Lines (RCL) products from February 6 to March 11, 2023, a team of three consultants conducted a security review of the client-provided source code, with 12 person-weeks of effort. Our testing efforts focused on identifying issues that would result in attackers stealing funds, abusing the arithmetic in the products. With access to source code and documentation, we performed static and dynamic testing of the target system, using automated and manual processes.

Observations and Impact

In developing the target codebases, Atlendis Labs implemented a series of loan products: bullet loans, coupon bullet loans, installment loans, and RCLs. The arithmetic complexity of these products poses risks for many of the components. Our review suggested that the codebase would benefit from additional efforts toward unit and automated testing, data structure redesign, and specific arithmetic analysis. Many of these issues stem from the following:

- Improper precision handling in arithmetic
- Insufficient data validation in functions
- Deviations between specifications and code, in both functionality and naming

During the audit, we discovered several high-severity findings that affect many of the components, including the following:

- Attackers being able to steal funds through draining cancellation fees
- Borrowers being able to skip last payments due to arithmetic miscalculation
- Actors locking up others' funds due to incorrect data validation
- Lenders preventing each other from receiving rewards through missing data validation

Our review highlighted that the testing, specification, and arithmetic implementations of the Atlendis Labs Protocol codebase do not meet the expected code maturity requirements. The codebase appears to require further development, specifically to address data structure and arithmetic issues.

Recommendations

Based on the codebase maturity evaluation and findings identified during the security review, Trail of Bits recommends that Atlendis Labs address the following recommendations prior to deploying the Loans and RCL products:

- **Remediate the findings disclosed in this report.** The findings described in Detailed Findings pose tangible risk to the Atlendis Labs loan products. These findings should be addressed immediately as part of a direct remediation or as part of any refactor that may occur when addressing other recommendations.
- **Extend fuzzing suite to test complex system interactions.** Implementing these techniques across tests can identify additional edge cases that the system does not support correctly. These practices often help uncover obscure bugs that may not be obvious. Fuzzing campaigns should complement unit tests, and do not replace the requirement for them.
- **Expand unit tests to cover realistic scenarios that mirror expected configuration values in production.** These tests should cover all potential production configuration settings to ensure they test realistic scenarios. Integrating such unit tests can help detect bugs early during the development process.
- **Analyze all formulas in the system with respect to rounding directions and ensure that the pool is always favored over the users for all operations.** This analysis ensures that users cannot profit from the system due to unexpected behavior. Appendix C discusses our recommendations for formulas to ensure that they round in correct dimensions; we recommend that Atlendis Labs continue to develop/implement these formulas.
- **Simplify complex contracts and split logic into smaller contracts with smaller responsibilities.** Contracts should be small and concise to enhance readability and understandability. Appendix D discusses our recommendations for simplifying the system.
- **Consolidate all system documentation in a single document.** A significant portion of documentation is split across the whitepaper, flowcharts, and the repository README.
- **Consider a second security review after implementing thorough tests against the system.** The system uses complex arithmetic and rounding directions, as well as various state-changing operations whose effects take place over a long period of time. This complexity increases the attack surfaces of the system and increases the likelihood of additional issues. In particular, several areas identified in the coverage section (e.g., bullet and installment loans and RCL formulas) would benefit from further investigations. We recommend a follow-up security review.

The following tables provide the number of findings by severity and category.

EXPOSURE ANALYSIS

Severity	*Count*
High	6
Medium	10
Low	1
Informational	8

CATEGORY BREAKDOWN

Category	*Count*
Access Controls	1
Configuration	1
Data Validation	17
Denial of Service	2
Timing	3
Undefined Behavior	1

Project Summary

Contact Information

The following managers were associated with this project:

Dan Guido, Account Manager
dan@trailofbits.com

Anne Marie Barry, Project Manager
annemarie.barry@trailofbits.com

The following engineers were associated with this project:

Nat Chin, Consultant
natalie.chin@trailofbits.com

Justin Jacob, Consultant
justin.jacob@trailofbits.com

Elvis Skozdopolj, Consultant
elvis.skozdopolj@trailofbits.com

Gustavo Grieco, Consultant
gustavo.grieco@trailofbits.com

Project Timeline

The significant events and milestones of the project are listed below.

Date	Event
February 2, 2023	Pre-project kickoff call
February 13, 2023	Status update meeting #1
February 21, 2023	Status update meeting #2
March 6, 2023	Status update meeting #3
March 14, 2023	Delivery of report draft
March 14, 2023	Report readout meeting
April 18, 2023	Delivery of final report

Project Goals

The engagement was scoped to provide a security assessment of the Atlendis Labs Borrowing Protocol. Specifically, we sought to answer the following non-exhaustive list of questions:

- Is it possible for an attacker to steal funds?
- Does the arithmetic handle rounding correctly?
- Does the tick and epoch logic correctly match specifications?
- Is it possible to prevent borrows or repayments?
- Are financial calculations (e.g., interest and repayment calculations) computed correctly?

Project Targets

The engagement involved a review and testing of the following target.

Atlendis Smart Contracts V2

Repository	https://github.com/Atlendis/priv-contracts-v2
Version	`19634e2ccae22e35634f336d2acd7f48d1647ea9`
Type	Solidity
Platform	Ethereum

Project Coverage

This section provides an overview of the analysis coverage of the review, as determined by our high-level engagement goals.

We reviewed the loan products in the system through manual review, with a focus on RCLs for automated testing. Each of the loan products contained a factory, and a primary loan contract, which were reviewed using manual review. The contracts we investigated include the following:

Simple Loan Products

The following loan products (bullet loan, coupon bullet, and installment loan) inherit from a shared `LoanBase` contract.

- **Bullet loans.** These products allow borrowers to borrow funds, where the principal and interest is due at the expiration of the loan period, and provide lenders with the opportunity to gain rewards for their funds. We manually reviewed the arithmetic calculations of repayments and interest calculations, with a focus on rounding. We set up automated testing for these contracts, and implemented basic deployment checks. We recommend extending the fuzzing campaign provided during this review, with a focus on fuzzing arithmetic formulas and system invariants.

- **Coupon bullet loan.** These contracts implement a coupon bullet loan, which allows users to pay back partial interest of the loan with a lump sum payment at the expiration of the loan period. As with bullet loans, they allow lenders to gain additional interest for the deposit of the funds. We investigated any opportunities for the borrower or lender to cause undefined behavior or skip payments. We manually reviewed the logic on the borrower and lender flows with a focus on the arithmetic of the repay calculations and calculated payment amounts per interest rate. We set up end-to-end fuzzing on these contracts and implemented basic deployment checks. We recommend extending the fuzzing campaign provided during this review, with a focus on functional pre- and post-conditions.

- **Installment loans.** These contracts allow borrowers to withdraw a lump sum and repay in individual installments over a period of time, until the expiry of the loan period. We reviewed the calculation of the positions' current value, accrual amounts, and due payments. We focused on the correctness of formulas in these calculations. We also set up end-to-end fuzzing on installment loans with basic checks. We recommend further investigating these formulas, as fuzzing on this section was time-limited. Additional system invariants that can be tested are noted in Appendix E: System Invariant Recommendations.

- **LoanBase.** This contract implements shared functionality for bullet loans, coupon bullets, and installment loans. We reviewed the loan's phase-transition mechanisms, as well as the allowed actions of the borrowers, lenders, and governance during each phase, to ensure users cannot force the system into an unexpected state.

Complex Loan Products

- **Revolving Credit Lines and Tick Logic.** These contracts implement the most complicated version of the credit lines, tracking amounts of borrows and lends over "ticks" and "epochs." The previous loans in a cycle are compounded to allow borrowers to use their credit from previous borrows and apply it to their new loan. We manually reviewed arithmetic calculations for user actions, looking for areas where borrowers and lenders could profit from incorrect rounding directions or precision loss. We reviewed the epoch calculations and loan cycles to see if an attacker could cause any mismatch in internal accounting variables, e.g., by de-syncing the epoch and/or the position ID such that the value of their position is incorrectly evaluated. We reviewed system operations with respect to the provided documentation to identify divergences in behavior. In addition, we investigated ways for a borrower to grief a lender and prevent them from withdrawing or earning interest. We manually reviewed the calculated price values of loans per epoch, with a focus on ways an attacker would be able to manipulate values. We also used automated testing to check expected behavior of functions and their pre- and post-conditions. Although we focused on this product for most of our review, we recommend that Atlendis Labs continue to extend the Echidna tests we have written, as more complex exploits related to subsequent credit are very likely.

Auxiliary Contracts

- **Pool Custodian (and respective adapters).** A pool can make deposits and withdrawals using funds in this contract; additionally, governance can execute a special "withdraw" feature on the contract. The Pool Custodian contract behavior is controlled by `delegatecall` operations against adapters. We manually reviewed the access control and the contract actions to ensure they are correctly limited and that known risks associated with using `delegatecall` are not present. We focused primarily on the Neutral adapter and AaveV2V3 adapter through manual review; however, as coverage was limited, we recommend using Echidna to thoroughly test the Pool Custodian and integration of adapters.

- **Non-standard repayment module.** Governance initiates this flow to trigger a non-standard or atypical repayment plan. We reviewed the proper access controls of the contract.

- **On-chain access controls.** We briefly reviewed role-based access controls related to the granting and revoking of roles.

Coverage Limitations

Because of the time-boxed nature of testing work, it is common to encounter coverage limitations. The following list outlines the coverage limitations of the engagement and indicates system elements that may warrant further review:

- **Operational risks.**
    - **Granting of roles in the system.** The Atlendis Labs system is permissioned and requires users to be given "lender" and "borrower" roles. The process of gaining these roles was not in scope for this review.
    - **Governance and control of keys.** The 3-of-5 multisig contract that controls when governance operations call the functions was not included in scope.
- **Off-chain and back-end infrastructure.**
    - **Governance-controlled state changes.** We did not review the circumstances in which Atlendis Labs would use the governance contract to execute state changes, including the initiation of a non-standard repayment schedule. We recommend that Atlendis Labs clearly identify the expected scenarios in which this would be enabled in order to ensure that users know what to expect.
    - **Deployment scripts.** While product factories were in scope for our review, we did not consider deployment scripts during our analysis of the codebase.
- **Smart contracts.** The contracts listed below were considered out of scope during this review due to time constraints.
    - **Rewards Manager.** This contract handles the rewards that can be obtained by depositing the NFT.
    - **Staking contracts.** These contracts provide fixed or continuous rewards to lenders for staking their assets. We performed a very light review of these contracts, and they were considered out of scope for the remainder of the review.
    - **Fee calculation and registration.** The fee calculation and registration across the protocol require additional investigation.

Automated Testing

Trail of Bits uses automated techniques to extensively test the security properties of software. We use both open-source static analysis and fuzzing utilities, along with tools developed in house, to perform automated testing of source code and compiled software.

Test Harness Configuration

We used the following tools in the automated testing phase of this project:

Tool	Description
Slither	A static analysis framework that can statically verify algebraic relationships between Solidity variables
Echidna	A smart contract fuzzer that can rapidly test security properties via malicious, coverage-guided test case generation

Test Results

The results of this focused testing are detailed below.

Revolving Credit Lines. The RCLs allow borrowers to take out recurring loans against their deposits. Using Echidna, we tested the following system properties.

Property	Tool	Result
If preconditions are met, the `deposit` function never reverts.	Echidna	**Passed**
If preconditions are met, the `updateRate` function never reverts.	Echidna	TOB-ATL-11, TOB-ATL-12
If preconditions are met, the `withdraw` function never reverts.	Echidna	**Passed**
If preconditions are met, the `exit` function never reverts.	Echidna	**Passed**

If preconditions are met, the `optOut` function never reverts.	Echidna	TOB-ATL-8
If preconditions are met, the `borrow` function never reverts.	Echidna	Passed
If preconditions are met, the `repay` function never reverts.	Echidna	TOB-ATL-8
If preconditions are met, the `detach` function never reverts.	Echidna	WIP

Codebase Maturity Evaluation

Trail of Bits uses a traffic-light protocol to provide each client with a clear understanding of the areas in which its codebase is mature, immature, or underdeveloped. Deficiencies identified here often stem from root causes within the software development life cycle that should be addressed through standardization measures (e.g., the use of common libraries, functions, or frameworks) or training and awareness programs.

Category	Summary	Result
Arithmetic	Imprecision and miscalculations in the codebase allow attackers to steal funds in a variety of ways, including the draining of fees, lenders affecting each others' interest accruals, and borrowers reducing lenders' accruals. The code contained many instances that used native Solidity rounding and contained arbitrary precision bound checks. The arithmetic in the system lacked corresponding tests to match the expected configuration of the codebase in production. No fuzzing was implemented to complement unit tests. We recommend that Atlendis Labs apply the analysis specified in our rounding recommendations in Appendix C and further extend the fuzzing campaign created during this review.	**Weak**
Auditing	Most state-changing operations emit events, and the Atlendis Labs team has offchain monitoring systems in place. However, there are some areas of the codebase where state-changing operations do not emit events, such as changing the orderbook phase in `RCLGovernance.sol`. In addition, we recommend documenting the monitoring systems and their use cases/limitations more thoroughly, as well as improving and documenting the incident response plans in the event of unintended behavior.	**Moderate**
Authentication / Access Controls	Different actors have been created for different roles and the access control architecture is well thought out. However, the test suite requires additional improvements with all unhappy and happy paths tested to ensure proper behavior; for example, the granting of roles, including the role manager contract, should be	**Moderate**

	tested. Moreover, the two-step process should be used for all privileged operations to ensure that re-assignment of roles is not error-prone.	
Complexity Management	The `TickLogic` and `RevolvingCreditLine` contracts are incredibly complicated due to their high branching complexity. This makes it difficult to understand the code's intended functionality, and causes obscure behavior in the codebase. We recommend following our recommendations in Appendix D, which provides ways to modularize the codebase.	**Weak**
Decentralization	Privileged operations in the products typically go through governance, which is a ⅗ multisignature wallet. However, it is unclear how voting takes place and if there are any circumstances (e.g., emergencies) where governance can be bypassed. We recommend documenting how voting is expected to take place, and under what circumstances emergency actions would be taken.	**Moderate**
Documentation	We observed substantial thorough, formal specifications and detailed flowcharts covering the broader components of the system. However, we also observed inconsistencies in the implementation; these observations are addressed more specifically in their respective categories (e.g., complexity management and authentication/access controls). We recommend addressing recommendations specified in TOB-ATL-16.	**Moderate**
Front-Running Resistance	We found an issue related to the front-running of initialization functions that would allow an attacker to take control over many of the system contracts (TOB-ATL-6). In addition, many lender actions in the RCL product can be prevented by a malicious borrower, and the way epochs and accruals are calculated currently incentivizes lenders to deposit as late as possible (TOB-ATL-23). While we did not find any errors that could lead to loss of funds, we recommend that the Atlendis team investigate potential areas of front-running and either document them thoroughly or, if possible, refactor	**Further Investigation Required**

	the codebase to mitigate them.	
Low-Level Manipulation	The adapter contracts rely on identical storage layouts between adapter and custodian storage contracts, which increase the need for careful consideration when checking storage slots. We recommend integrating slither-read-storage to ensure that these storage slots are implemented correctly. Moreover, the Pool Custodian moves funds around the contracts through a `delegatecall` operation on adapters, making these operations harder to track and mistakes easier to make.	**Moderate**
Testing and Verification	The current test suite tests only a subset of parameters, with either no fees or only some fees enabled. We recommend that Atlendis Labs extend the current unit tests to adequately cover realistic scenarios. These include situations with fees turned on, tokens with different decimals, longer loan durations, and access controls against all aspects of the codebase. These parameters and configuration should match expected deployments on live production environments. Additionally, we recommend extending the fuzzing suite to find additional unexpected behavior in the contracts. These test suites should help developers detect issues during the development process.	**Weak**

Summary of Findings

The table below summarizes the findings of the review, including type and severity details.

ID	Title	Type	Severity
1	Borrower can drain lender assets by withdrawing the cancellationFee multiple times	Data Validation	High
2	Incorrect fee calculation on withdrawal can lead to DoS of withdrawals or loss of assets	Data Validation	High
3	Lack of zero-address checks	Data Validation	High
4	Problematic approach to data validation	Data Validation	Medium
5	Borrower can skip the last coupon payment	Data Validation	Medium
6	Initialization functions can be front-run	Timing	Informational
7	Lenders' unborrowed deposits can be locked up by a borrower	Data Validation	Medium
8	optOut can be called multiple times	Data Validation	High
9	Risks related to deflationary, inflationary, or rebasing tokens	Data Validation	Medium
10	Rounding down when computing fees benefits users	Data Validation	Low
11	Lenders can prevent each other from earning interest	Data Validation	Medium
12	Incorrect calculation in getPositionRepartition can lock a user's position	Denial of Service	Medium

13	Detached positions are incorrectly calculated	Data Validation	Medium
14	Borrower can reduce lender accruals	Data Validation	Medium
15	Borrower can start a lending cycle before deposits are made	Data Validation	Informational
16	Documentation and naming conventions can be improved	Undefined Behavior	Informational
17	Missing validation in detach	Data Validation	Informational
18	Contract architecture is overcomplicated	Configuration	Informational
19	Governance is a single point of failure	Access Controls	High
20	Pool is put in NON_STANDARD state only after executeTimelock() is called	Timing	Informational
21	Detached positions cannot be exited during subsequent loans	Denial of Service	Medium
22	Roles manager can never be updated	Data Validation	High
23	Risks with transaction reordering	Timing	Informational
24	Problematic approach to the handling precision errors	Data Validation	Informational
25	Lenders with larger deposits earn less accruals if their position is only partially borrowed	Data Validation	Medium

Detailed Findings

1. Borrower can drain lender assets by withdrawing the cancellationFee multiple times

Severity: **High**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-ATL-1
Target: `LoanBorrower.sol`	

Description

If a loan product is canceled by governance and the cancellation fee that the borrower is able to withdraw is non-zero, the borrower can drain the assets deposited by lenders by withdrawing his cancellation fee multiple times.

The protocol requires borrowers to deposit a cancellation fee equal to some percentage of the target amount of tokens the borrower would like to be able to borrow. If the loan is canceled, the borrower may be able to withdraw either the full cancellation fee or a part of the cancellation fee that was not distributed to lenders. The exact amount is defined by the `cancellationFeeEscrow` state variable, and the withdrawal can be done using the `withdrawRemainingEscrow` function.

```
77    function withdrawRemainingEscrow(address to) external onlyBorrower
onlyInPhase(LoanTypes.PoolPhase.CANCELLED) {
78        CUSTODIAN.withdraw(cancellationFeeEscrow, to);
79        emit EscrowWithdrawn(address(this), cancellationFeeEscrow);
80    }
```

Figure 1.1: The `withdrawRemainingEscrow` function in `LoanBorrower.sol`

This will transfer the assets out of the `PoolCustodian` contract, which holds the borrower's deposit as well as the lender's deposits, to the borrower-provided address.

```
152    function withdraw(uint256 amount, address to) public onlyPool {
153        if (amount > depositedBalance) revert NOT_ENOUGH_DEPOSITS();
154
155        collectRewards();
156
157        depositedBalance -= amount;
158
159        bytes memory returnData = adapterDelegateCall(adapter,
abi.encodeWithSignature('withdraw(uint256)', amount));
```

```
        if (!abi.decode(returnData, (bool))) revert DELEGATE_CALL_FAIL();
        token.safeTransfer(to, amount);

        emit Withdraw(amount, to, adapter, yieldProvider);
    }
```

Figure 1.2: The `withdraw` function in `PoolCustodian.sol`

Although the total deposited amount is reduced correctly in the `PoolCustodian` contract, the `cancellationFeeEscrow` state variable is never reduced to zero. This allows the borrower to withdraw the same amount of tokens multiple times, draining the pool of all assets deposited by the lenders.

Exploit Scenario

A bullet loan is created for Bob as the borrower with a target origination amount of 10,000 USDC. Bob deposits a 1,000 USDC cancellation fee into the pool, which begins the book-building phase of the pool.

Alice deposits 10,000 USDC into the pool at a 5% rate. After some time has passed, the governance transitions the bullet loan into the origination phase so Bob can borrow the asset.

Bob decides not to borrow from the pool and the loan is canceled by governance, which decides not to redistribute the cancellation fee to Alice.

Bob has 1,000 USDC in escrow and can withdraw it using the `withdrawRemainingEscrow` function. Since the value of the cancellationFeeEscrow state variable is never reduced, Bob can withdraw his cancellation fee nine more times, draining Alice's deposit.

Recommendations

Short term, set the `cancellationFeeEscrow` state variable to zero when the borrower withdraws their cancellation fee.

Long term, implement additional unit tests that test for edge cases. Come up with a list of system and function-level invariants and test them with Echidna.

2. Incorrect fee calculation on withdrawal can lead to DoS of withdrawals or loss of assets	
Severity: **High**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-ATL-2
Target: `LoanLender.sol`	

Description

Incorrect fee calculation in the `withdraw` function of the `LoanLender` contract can lead to inflated fees if the asset used does not have 18 decimals of precision. This will prevent any lender from withdrawing their deposit and earnings, or potentially make the lenders lose a significant part of their deposit and earnings.

The protocol allows lenders to withdraw their share of earnings once a borrower has repaid a part of their borrowed amount. These earnings are reduced by a small protocol fee in the `withdraw` function, shown in figure 2.1. The calculation uses `WAD` (shown in figure 2.2) as a scaling factor since the `FEES_CONTROLLER` uses 18 decimals of precision, while the `LoanLender` contract uses the same decimal precision as the token.

```
uint256 fees = earnings.mul(WAD).mul(FEES_CONTROLLER.getRepaymentFeesRate());
```

Figure 2.1: Fee calculation in `LoanLender.sol`

```
uint256 constant WAD = 1e18;
```

Figure 2.1: Scaling factor `WAD` in `LoanLender.sol`

However, due to an incorrect calculation, these fees will be several orders of magnitude higher (or lower) than intended. Depending on the decimal precision of the token used, this can lead to three different scenarios:

1. If the token decimals are sufficiently smaller than 18, this will stop the function from correctly executing, preventing any lenders from withdrawing their deposit and earnings.
2. If the token decimals are slightly smaller than 18, the protocol will charge a higher-than-intended fee, making the lenders lose a significant amount of their earnings.
3. If the token decimals are higher than 18, the protocol will charge a smaller-than-intended fee, which will lead to a loss for the protocol.

Exploit Scenario

A coupon bullet loan is created for Alice with a target origination amount of 1,000,000 USDC and a repayment fee of 1% (0.01e18). Bob deposits 1,000,000 USDC at a 10% rate so it is available for Alice to borrow.

Alice borrows the entire amount. Since this is a coupon loan, Alice will be repaying a share of the loan interest at a regular interval.

Assume Alice has repaid 1,000 USDC in her first coupon repayment and Bob's total earnings are 1000 USDC. Bob attempts to withdraw his earnings, but the function reverts due to an arithmetic underflow.

If we calculate the fee using the current calculation—keeping in mind that the fixed point calculation is set to the amount of decimals of USDC (i.e., six)—we get the following:

```
fee = 1000e6.mul(1e18).mul(0.01e18)
```

```
fee = 10000000000000000000000000e6 or
10,000,000,000,000,000,000,000,000 USDC
```

This fee far exceeds the deposited and earned amount, causing the function to revert due to an arithmetic underflow.

Recommendations

Short term, revise the calculation such that token amounts are properly normalized before using fixed point multiplication with the repayment fee. Consider setting the `PoolTokenFeesController` precision to be equal to the amount of token decimals.

Long term, replicate real-world system settings in unit tests with a variety of different tokens that have more (or fewer) than 18 decimals of precision. Set up fuzzing tests with Echidna for all calculations done within the system to ensure they work with different token decimals and hold proper precision.

3. Lack of zero-address checks	
Severity: **High**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-ATL-3
Target: `src/`	

Description

A number of functions in the codebase do not revert if the zero address is passed in for a parameter that should not be set to zero.

The following parameters, among others, do not have zero-address checks:

- The `token` and `rolesManager` variables in the PoolCustodian's `constructor`
- The `_rolesManager`, `_pool`, and `_rewardsOperator` in `updateRolesManager`, `initializePool`, and `updateRewardsOperator`, respectively
- The `factoryRegistry` in `LoanFactoryBase` and in the `RCLFactory`
- The `rolesManager` in `Managed.sol`'s `constructor`
- The `module` address in `RewardsManager.sol`'s `addModule` function

Exploit Scenario

Alice, the deployer of the `RCLFactory`, accidentally sets the `factoryRegistry` parameter to zero in the constructor. As a result, the `deploy` function will always revert and the contract will be unusable.

The governance in control of the `RewardsManager` contract accidentally adds an invalid module via the `addModule` function. Since all user actions loop through all of the added modules, all user-staked positions become permanently locked, and users are prevented from unstaking or claiming rewards.

Recommendations

Short term, add zero-address checks for the parameters listed above and for all other parameters for which the zero address is not an acceptable value. Add a `supportsInterface` check for any modules added to the `RewardsManager` contract to ensure an invalid module cannot be added.

Long term, review input validation across components. Avoid relying solely on the validation performed by front-end code, scripts, or other contracts, as a bug in any of those components could prevent them from performing that validation.

4. Problematic approach to data validation

Severity: **Medium**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-ATL-4
Target: `src/*`	

Description

The Atlendis Labs infrastructure relies on validation functions that exist outside of contract constructors. This requires an explicit call to a validate helper function, which can result in unexpected behavior if a function forgets to call it prior to construction.

For example, the `LoanState` constructor saves a `totalPaymentsCount` variable, meant to represent the total number of payments for the loan, denominated by the `LOAN_DURATION` and `PAYMENT_PERIOD`. This means that this calculation can potentially round down to zero, but the constructor lacks checks to ensure it is non-zero.

```
/**
 * @dev Constructor
 * @param rolesManager Address of the roles manager contract
 * @param custodian Address of the custodian contract
 * @param feesController Address of the fees controller contract
 * @param ratesAmountsConfig Other Configurations
 * @param durationsConfig Other Configurations
 */
constructor(
    address rolesManager,
    IPoolCustodian custodian,
    IFeesController feesController,
    bytes memory ratesAmountsConfig,
    bytes memory durationsConfig
) Managed(rolesManager) {
    [...]

    totalPaymentsCount = PAYMENT_PERIOD > 0 ? LOAN_DURATION / PAYMENT_PERIOD : 0;

    CREATION_TIMESTAMP = block.timestamp;
    CUSTODIAN = custodian;
    FEES_CONTROLLER = feesController;
}
```

Figure 4.1: Constructor function in `LoanState.sol`

Instead, the checks occur in the `LoanFactoryBase` contract, when `deployLoan` is called. This flow *assumes* that the helper function will be used.

```
function deployLoan(
    StandardRolesManager rolesManager,
    PoolCustodian custodian,
    PoolTokenFeesController feesController
) private returns (address) {
    if (ratesAmountsConfigTmp.length != 256) revert INVALID_PRODUCT_PARAMS();
    if (durationsConfigTmp.length != 160) revert INVALID_PRODUCT_PARAMS();

    [...]

    uint256 ONE = 10**custodian.getAssetDecimals();

    if (minOrigination > targetOrigination) revert INVALID_ORIGINATION_PARAMETERS();
    if (minRate >= maxRate) revert INVALID_RATE_BOUNDARIES();
    if (rateSpacing == 0) revert INVALID_ZERO_RATE_SPACING();
    if ((maxRate - minRate) % rateSpacing != 0) revert INVALID_RATE_PARAMETERS();
    if (cancellationFeePc >= ONE) revert INVALID_PERCENTAGE_VALUE();

    if (paymentPeriod > 0 && loanDuration % paymentPeriod != 0) revert
PAYMENT_PERIOD();
    if (paymentPeriod > 0 && repaymentPeriod > paymentPeriod) revert
REPAYMENT_PERIOD_TOO_HIGH();

    return deployPool(address(rolesManager), custodian, feesController);
}
```

Figure 4.2: `deployLoan` function in `LoanFactoryBase`

This pattern is duplicated across the codebase in the form of helper functions, which are used in other contracts. As a result, the code relies on assumptions of prior validation elsewhere in the system. This practice is error-prone, as the validation checks are not guaranteed to run.

Exploit Scenario

Alice, an Atlendis protocol developer, changes the logic to deploy a loan. Because the constructor of the `LoanState` contract has no data validation checks, she makes an erroneous assumption about what is validated in another part of the system.

Recommendations

Short term, move the validation logic to the constructor of the `LoanState` contract instead of using the factory deployment.

Long term, always keep the validation logic as close as possible to locations in the code where variables are being used and declared. This ensures all the inputs are properly validated.

5. Borrower can skip the last coupon payment

Severity: **Medium**	Difficulty: **Medium**
Type: Data Validation	Finding ID: TOB-ATL-5
Target: `LoanLogic.sol`	

Description

If a borrower calls `repay` at the exact moment of the start of the repayment period, the borrower can skip paying the last coupon amount and keep the last payment instead.

The protocol allows the creation of a coupon bullet loan, toward which the borrower repays a part of the loan interest in regular intervals before finally repaying the principal amount. To account for late repayment, the protocol defines a repayment period duration over which the borrower can repay the loan before it reaches maturity. This is shown in figure 5.1.

```
   if (block.timestamp < borrowTimestamp + LOAN_DURATION -
REPAYMENT_PERIOD_DURATION)
       revert LoanErrors.LOAN_REPAY_TOO_EARLY();
```

Figure 5.1: `_repay` validation in `LoanBorrower.sol`

Upon calling the `repay` function to repay the entire loan, the function will calculate and attempt to repay any outstanding coupon payments before paying the leftover principal amount, as shown in figure 5.2.

The full repayment will first calculate how many coupon payments the borrower has left and pay them.

```
function repay() external onlyBorrower {
   uint256 paymentsToMake = _getNumberOfPaymentsExpired() - paymentsDoneCount;
   _makePaymentsDue(paymentsToMake);
   _repay(CouponBullet.calculateRepayAmount);
}
```

Figure 5.2: `repay` function in `CouponBullet.sol`

However, the contracts are missing a check for the case where the repayment timestamp is equal to the start of the repayment period. In this case, the calculation will return one less coupon payment but still allow for principal repayment.

```
    uint256 loanMaturity = borrowTimestamp + loanDuration;
    if (referenceTimestamp > loanMaturity - repaymentPeriodDuration) return loanDuration / paymentPeriod;
    uint256 maturityCappedTimestamp = referenceTimestamp > loanMaturity ? loanMaturity : referenceTimestamp;
    return (maturityCappedTimestamp - borrowTimestamp) / paymentPeriod;
```

Figure 5.3: Expired coupons calculation in `LoanLogic.sol`

This will allow the borrower to repay the principal, which will transition the loan to the `REPAID` state, allowing the borrower to keep the last coupon payment for themselves.

Exploit Scenario

Alice has a coupon bullet loan with a target origination amount of 1,000,000 USDC that is borrowed at a 10% interest rate and lasts for 10 days. She needs to pay 10,000 USDC in coupon payments, and then pay the last 1,000,000 USDC as principal.

Alice makes the first eight coupon payments and sets up a script to repay the principal at the exact timestamp when the repayment period starts.

Her script runs successfully, skipping the last coupon payment and repaying the principal amount. Alice keeps the last 1,000 USDC payment for herself.

Recommendations

Short term, update the check in `LoanLogic` to account for the case where the repayment timestamp is equal to the start of the repayment period:

```
referenceTimestamp >= loanMaturity - repaymentPeriodDuration
```

Long term, explicitly specify preconditions and postconditions for all functions to more easily identify what is being checked and what needs to be checked in a function. Set up fuzzing tests with Echidna to identify unexpected behavior.

6. Initialization functions can be front-run

Severity: **Informational**	Difficulty: **High**
Type: Timing	Finding ID: TOB-ATL-6
Target: `src/FactoryRegistry.sol`	

Description

The `FactoryRegistry` contract is designed to register product factories and deploy products. As an access control mechanism, the `initialize` function transfers ownership to the `governance` parameter, allowing governance to execute owner-privileged actions.

However, the initialization function can be front-run by an attacker. An attacker can monitor the public mempool for the transaction that will call the `initialize` function and send their own transaction with a higher gas cost, calling the `initialize` function with their controlled address.

```
/**
 * @dev Implementation of the initialize method according to Initializable contract
specs
 */
function initialize(address governance) public initializer {
    __Ownable_init();
    transferOwnership(governance);
    version++;
}
```

Figure 6.1: The `initialize` function in `FactoryRegistry.sol`

Exploit Scenario

Alice, the deployer of the `FactoryRegistry` contract, calls `initialize`, passing in the address of the governance contract. Eve notices her call in the mempool and front-runs her, calling `initialize` with her own address. As a result, the legitimate governance contract cannot register a new factory or deploy any products. Alice will need to redeploy.

Recommendations

Short term, ensure that the deployment scripts have robust protections against front-running attacks.

Long term, create an architecture diagram that includes the system's functions and their arguments to identify any functions that can be front-run. Additionally, document all cases in which front-running may be possible, along with the effects of front-running on the codebase.

7. Lenders’ unborrowed deposits can be locked up by a borrower

Severity: **Medium**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-ATL-7
Target: `TickLogic.sol`	

Description

Due to an incorrect check in the `TickLogic` library contract, any lender whose position is borrowed for less than the minimum deposit amount will have their entire deposit locked up. The lenders’ deposits will be unlocked if the loan is repaid or their position becomes borrowed for an amount greater than the minimum deposit amount.

The protocol allows lenders to freely withdraw the unborrowed part of their deposit during an active loan by using the `detach` function. Before a withdrawal is made, the function checks that the position value at the end of the loan, subtracted by the unborrowed amount to be withdrawn, is greater than or equal to the minimum deposit amount:

```
if (endOfLoanPositionValue - unborrowedAmount < minDepositAmount)
    revert RevolvingCreditLineErrors.RCL_REMAINING_AMOUNT_TOO_LOW();
```

Figure 7.1: Withdrawal amount validation in `TickLogic.sol`

However, this check will prevent the withdrawal of unborrowed assets for any lenders whose position is borrowed for less than the minimum deposit amount. This allows the borrower to either intentionally or accidentally lock up the lenders’ funds.

Exploit Scenario

Assume a Revolving Credit Line is created for Alice with a minimum deposit amount of 1 ether and a loan duration of one year.

1. Ten lenders deposit 10 ether each into the pool at a 5% rate before the first borrow is made.
2. Alice borrows 9 ether. Since all of the lenders have deposited at the same rate and the same epoch, their positions are equally borrowed by 0.9 ether.

Although each lender has 9.1 ether in unborrowed assets, they are prevented from detaching their unborrowed assets due to their position being borrowed for less than the minimum deposit amount. This allows Alice to force the lenders into keeping their funds in the pool.

Recommendations

Short term, revise the check to allow lenders to withdraw their unborrowed amount as long as the sum of the leftover amount and the borrowed amount left in the contract is larger than or equal to the minimum deposit amount.

Long term, use extensive smart contract fuzzing to test that users are never blocked from performing expected operations.

8. optOut can be called multiple times

Severity: **High**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-ATL-8
Target: `TickLogic.sol`	

Description

Lenders are allowed to trigger an opt out operation multiple times over the same loan, with unclear implications.

When a lender wishes not to be a part of the next loan cycle, they can call the `optOut` function to signal their intent. This function will validate that the position has been borrowed and that the loan has not passed maturity.

```
function validateOptOut(
    DataTypes.Position storage position,
    DataTypes.Tick storage tick,
    DataTypes.Loan storage referenceLoan,
    DataTypes.Loan storage currentLoan
) external view {
    (, uint256 borrowedAmount) = getPositionRepartition(tick, position,
referenceLoan, currentLoan);

    if (borrowedAmount == 0) revert
RevolvingCreditLineErrors.RCL_POSITION_NOT_BORROWED();
    if (block.timestamp > currentLoan.maturity) revert
RevolvingCreditLineErrors.RCL_MATURITY_PASSED();
}
```

Figure 8.1: The `validateOptOut` function in `TickLogic.sol`

To compute the `borrowedAmount`, the `getPositionRepartition` function is called. However, this function does not validate that the lender has not yet opted out of their position if the loan has not yet been repaid:

```
function getPositionRepartition(
    DataTypes.Tick storage tick,
    DataTypes.Position storage position,
    DataTypes.Loan storage referenceLoan,
    DataTypes.Loan storage currentLoan
) public view returns (uint256, uint256) {
    DataTypes.Epoch storage epoch = tick.epochs[position.epochId];
    uint256 unborrowedAmount;
    uint256 borrowedAmount;
    if (position.optOutLoanId > 0) {
        bool optOutLoanRepaid = (position.optOutLoanId < currentLoan.id) ||
currentLoan.maturity == 0;
        if (optOutLoanRepaid) {
            unborrowedAmount = getAdjustedAmount(tick, position, referenceLoan).mul(
                tick.endOfLoanYieldFactors[position.optOutLoanId]
            );
            return (unborrowedAmount, 0);
        }
[...]
unborrowedAmount = (epoch.deposited -
epoch.borrowed).mul(position.baseDeposit).div(epoch.deposited);
borrowedAmount = epoch.borrowed.mul(position.baseDeposit).div(epoch.deposited);
return (unborrowedAmount, borrowedAmount);
```

Figure 8.2: The `getPositionRepartition` function in `TickLogic.sol`

This allows a malicious lender to arbitrarily inflate the `optedOut` variable, which could break internal accounting.

Exploit Scenario

Eve, a malicious lender, calls the `optOut` function multiple times. This inflates the `adjustedOptOut` amount variable of her position drastically. Bob, a borrower, is unable to repay his loan due to an underflow when calculating the `tickAdjusted` variable in the `prepareTickForNextLoan` function.

Recommendations

Short term, ensure that users cannot call the `optOut` function again if they have already opted out of the next loan cycle.

Long term, improve unit test coverage and implement smart contract fuzzing to uncover potential edge cases and ensure intended behavior throughout the system.

9. Risks related to deflationary, inflationary, or rebasing tokens

Severity: **Medium**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-ATL-9
Target: `TickLogic.sol`	

Description

The loan product contract and the pool custodian contract do not check that the expected amount has been sent on transfer. This can lead to incorrect accounting in both contracts if they do not receive enough assets because tokens are inflationary, deflationary, or rebase.

Each lending product uses a pool custodian contract that holds custody of all ERC20 assets and through which the product contract deposits and withdraws the assets:

```
function _deposit(uint256 amount, address from) private {
    collectRewards();

    depositedBalance += amount;

    token.safeTransferFrom(from, address(this), amount);
    bytes memory returnData = adapterDelegateCall(adapter,
abi.encodeWithSignature('deposit(uint256)', amount));
    if (!abi.decode(returnData, (bool))) revert DELEGATE_CALL_FAIL();

    emit Deposit(amount, from, adapter, yieldProvider);
}
```

Figure 9.1: `deposit` function in the `PoolCustodian` contract

`safeTransferFrom` does not check that the expected value was transferred. If the tokens take a fee, the amount transferred can be less than expected. Similarly, the product contracts assume that the received amount is the same as the inputted amount. As a result, the custodian contract may hold fewer tokens than expected, and the accounting of the product may be incorrect.

Exploit Scenario

USDT enables a 0.5% fee on transfers. Every deposit to a product leads to the creation of positions with more tokens than received. As a result, the positions become insolvent.

Recommendations
Short term, check the custodian balance during deposits and withdraws, and adjust the expected position amounts. Be aware that some commonly used assets, such as USDT, can enable fees in the future. Additionally, prevent the usage of rebasing tokens.

Long term, review the token integration checklist in Appendix I, and ensure that the system is robust against edge cases to the ERC20 specification.

References

- Incident with non-standard ERC20 deflationary tokens

10. Rounding down when computing fees benefits users

Severity: **Low**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-ATL-10
Target: `LoanBorrower`, `RCLBorrowers`	

Description

Fees are computed using an incorrect rounding direction that reduces the amount collected by the protocol. As a result, fees may be less than intended when being transferred to the protocol.

After the `borrow` function has finished its primary execution, it calculates the total dust value in the contracts, then attributes allocation to pools.

```
uint256 fees = FEES_CONTROLLER.registerBorrowingFees(borrowedAmount);
FundsTransfer.chargedWithdraw({
    token: TOKEN,
    custodian: CUSTODIAN,
    recipient: to,
    amount: borrowedAmount - fees,
    fees: fees
});
```

Figure 10.1: External call for fee calculation in RCL`Borrowers.sol`

To compute the fees, the `registerBorrowingFees` function is called.

```
function registerBorrowingFees(uint256 amount) external onlyPool returns (uint256
fees) {
    fees = amount.mul(BORROWING_FEES_RATE);
    dueFees += fees;
    totalFees += fees;

    emit BorrowingFeesRegistered(TOKEN, fees);
}
```

Figure 10.2: Borrowing fees calculation function in `PoolTokenFeesController.sol`

However, this function does not seem to be doing any division; in fact, it uses the `mul` function from `FixedPointMathLib`, which is implemented using `mulDivDown` with the denominator as 10 ** `token.decimals()`. The division will not be exact most of the time, reducing the computed value.

Exploit Scenario

Alice deploys a pool, which requires a certain fee. Over time, a large number of users interact with the pool, but the rounding down of the fee computation produces less fees than expected.

Recommendations

Short term, ensure that the fees computation rounds in favor of the protocol.

Long term, improve unit test coverage and implement smart contract fuzzing to uncover potential edge cases and ensure intended behavior throughout the system.

11. Lenders can prevent each other from earning interest

Severity: **Medium**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-ATL-11
Target: `RCLLenders.sol`	

Description

A lender can prevent another lender's position from earning interest by opting out of their position and then updating their position's rate after the loan is repaid.

A lender can prevent their position from being borrowed in any future lending cycles by opting out of their position. An opted-out position will not be borrowed and can be withdrawn only once the loan is repaid; however, the protocol also allows the lender to update the rate of their position even though the position can never be borrowed.

This occurs because the `updateRate` function lacks a check that the lender has opted out of their position, as shown in figures 11.1 and 11.2.

```
function updateRate(uint256 positionId, uint256 newRate)
   external
   onlyLender
   onlyInPhase(DataTypes.OrderBookPhase.OPEN)
{
   [...]

   TickLogic.validateWithdraw({
       position: position,
       tick: tick,
       referenceLoan: referenceLoan,
       currentLoan: currentLoan,
       amountToWithdraw: type(uint256).max,
       minDepositAmount: minDepositAmount
   });
   [...]
```

Figure 11.1: updateRate in RCLLenders.sol

```
[...]
if (position.creationTimestamp == block.timestamp) revert
RevolvingCreditLineErrors.RCL_TIMELOCK();

(, uint256 borrowedAmount) = getPositionRepartition(tick, position,
referenceLoan, currentLoan);
```

```
if (borrowedAmount > 0) revert
RevolvingCreditLineErrors.RCL_POSITION_BORROWED();
uint256 positionCurrentValue = getPositionCurrentValue(tick, position,
referenceLoan, currentLoan);
if (amountToWithdraw != type(uint256).max && amountToWithdraw >
positionCurrentValue)
    revert RevolvingCreditLineErrors.RCL_AMOUNT_TOO_HIGH();
if (amountToWithdraw != type(uint256).max && positionCurrentValue -
amountToWithdraw < minDepositAmount)
    revert RevolvingCreditLineErrors.RCL_REMAINING_AMOUNT_TOO_LOW();
if (amountToWithdraw < minDepositAmount) revert
RevolvingCreditLineErrors.RCL_AMOUNT_TOO_LOW();
```

Figure 11.2: `validateWithdraw` in `TickLogic.sol`

Updating the rate of an opted-out position will internally withdraw the amount held in the position using the current rate and then deposit it using a new rate.

```
updatedAmount = withdraw(tick, position, type(uint256).max, referenceLoan,
currentLoan);
deposit(newTick, currentLoan, updatedAmount);
```

Figure 11.3: `updateRate` function in `TickLogic.sol`

After depositing using the new rate, the system will consider the amount held in the opted-out position as borrowable; however, the `getPositionCurrentValue` function will still return the entire amount as unborrowed. This allows the lender to withdraw their position.

```
uint256 positionCurrentValue = getPositionCurrentValue(tick, position,
referenceLoan, currentLoan);

if (expectedWithdrawnAmount == type(uint256).max) {
    withdrawnAmount = positionCurrentValue;
} else {
    withdrawnAmount = expectedWithdrawnAmount;
}

// if the position was optedOut to exit, its value has already been removed
from the tick data
if (position.optOutLoanId > 0) {
    return positionCurrentValue;
}
[...]
```

Figure 11.4: `withdraw` function in `TickLogic.sol`

This means that if a lender's position is borrowed, they can withdraw another lender's assets. The other lender will still be able to withdraw their assets if the loan is repaid or if there are more unborrowed assets in the pool, but they will not earn interest.

Exploit Scenario

A Revolving Credit Loan is created for Alice with a maximum borrow amount of 50 ether with a loan duration of 10 days.

1. Bob deposits 50 ether into the pool, which Alice borrows.
2. Bob opts out of his position, causing the system to consider his position unborrowable for future loans.
3. Charlie sees that there are no other lenders in the pool with borrowable assets and deposits 50 ether into the pool.
4. Alice repays her loan.
5. Bob updates the rate of his position to be lower than Charlie's.
6. Alice borrows another 50 ether, starting the next loan cycle.
7. Since Bob's position had a lower rate, his position is borrowed first. Bob then withdraws the 50 ether because the system incorrectly considers his position as not borrowed.
8. There is no ether left in the pool, so Charlie is not able to withdraw his assets, and his position will not earn interest for the duration of the loan.

Recommendations

Short term, update the validation in `updateRate` to prevent lenders who have opted out of their positions to update the rates of the positions.

Long term, improve unit test coverage and come up with a list of system properties that can be tested with smart contract fuzzing. This issue could be discovered by implementing a property test that checks that an opted-out position can never be borrowed from again.

12. Incorrect calculation in getPositionRepartition can lock a user's position

Severity: **Medium**	Difficulty: **Low**
Type: Denial of Service	Finding ID: TOB-ATL-12
Target: `TickLogic.sol`	

Description

A lender can lock their position if they deposit using (or update their position rate to equal) a rate that was borrowed in the previous lending cycle, but ends up not being borrowed during the next cycle.

The protocol uses interest rates and epochs to determine the order in which assets will be borrowed from during a lending cycle. The `deposit` function in `RCLLenders` is designed to account for this by updating different state variables depending on the rate and the epoch in which the assets were deposited.

```
positions[positionId] = DataTypes.Position({
   baseDeposit: amount,
   rate: rate,
   epochId: ticks[rate].currentEpochId,
   creationTimestamp: block.timestamp,
   optOutLoanId: 0,
   withdrawLoanId: 0,
   withdrawn: DataTypes.WithdrawalAmounts({borrowed: 0, expectedAccruals: 0})
});
```

Figure 12.1: Deposit function in `RCLLenders.sol`

This function internally uses the `deposit` function from `TickLogic` to perform the accounting:

```
function deposit(
   DataTypes.Tick storage tick,
   DataTypes.Loan storage currentLoan,
   uint256 amount
) public {
   if ((currentLoan.maturity > 0) && (tick.latestLoanId == currentLoan.id)) {
       tick.newEpochsAmounts.toBeAdjusted += amount;
       tick.newEpochsAmounts.available += amount;
       tick.epochs[tick.currentEpochId].deposited += amount;
   } else {
       tick.baseEpochsAmounts.available += amount;
       tick.baseEpochsAmounts.adjustedDeposits +=
amount.div(tick.yieldFactor);
        }
    }
```

Figure 12.2: Deposit function in `TickLogic.sol`

However, if the rate was borrowed during the last lending cycle but does not end up being borrowed during the next lending cycle, the protocol will incorrectly consider the position as part of the new epoch based on the `position.epochId`.

Since the protocol will include the value in the base epoch, the `epoch.deposited` value will not be updated. This will cause the `getPositionRepartition` function to incorrectly calculate the position's unborrowed and borrowed amount, causing a division-by-zero error because `epoch.deposited` is equal to 0:

```
function getPositionRepartition(
DataTypes.Tick storage tick,
DataTypes.Position storage position,
DataTypes.Loan storage referenceLoan,
DataTypes.Loan storage currentLoan
) internal view returns (uint256, uint256) {
DataTypes.Epoch storage epoch = tick.epochs[position.epochId];
[...]
unborrowedAmount = (epoch.deposited -
epoch.borrowed).mul(position.baseDeposit).div(epoch.deposited);
borrowedAmount =
epoch.borrowed.mul(position.baseDeposit).div(epoch.deposited);
return (unborrowedAmount, borrowedAmount);
}
```

Figure 12.3: The `getPositionRepartition` function in `TickLogic.sol`

This same issue can occur if a user updates the rate of their position to a previously borrowed rate that ends up not being borrowed in the next lending cycle. This is due to the `updateRate` function updating the `position.epochId`.

```
function updateRate(
    DataTypes.Position storage position,
    DataTypes.Tick storage tick,
    DataTypes.Tick storage newTick,
    DataTypes.Loan storage referenceLoan,
    DataTypes.Loan storage currentLoan,
    uint256 newRate
) public returns (uint256 updatedAmount) {
    updatedAmount = withdraw(tick, position, type(uint256).max, referenceLoan,
currentLoan);
    deposit(newTick, currentLoan, updatedAmount);

    position.baseDeposit = updatedAmount;
    position.rate = newRate;
```

```
        position.epochId = newTick.currentEpochId;
    }
```

Figure 12.4: The `updateRate` function in `TickLogic.sol`

In both cases, the user will be prevented from withdrawing their position, essentially locking up their assets until the borrower borrows from their position or repays the current loan.

Exploit Scenario

A Revolving Credit Line is created for Alice with a maximum borrowable amount of 10 ether.

Bob deposits 5 ether into the pool at a 10% rate, and Charlie deposits 10 ether into the pool at a 15% rate. Alice borrows 10 ether and, after some time has passed, she repays the loan.

```
    vm.startPrank(lenderOne);
    uint256 positionBob = revolvingCreditLineInstanceA.deposit(0.1 ether, 5 ether,
lenderOne);
    uint256 positionCharlie = revolvingCreditLineInstanceA.deposit(0.15 ether, 10
ether, lenderOne);
    vm.stopPrank();

    vm.startPrank(borrower);
    revolvingCreditLineInstanceA.borrow(borrower, 10 ether);
    skip(10 hours);
    revolvingCreditLineInstanceA.repay();
    vm.stopPrank();
```

Figure 12.3: Bob and Charlie deposit, and Alice borrows and repays

Bob updates the rate of his position to 5% but later changes his mind and updates it back to 10%. Charlie sees this and updates his rate to 9% so his position ends up being fully borrowed.

```
    vm.startPrank(lenderOne);
    revolvingCreditLineInstanceA.updateRate(positionBob, 0.05 ether);
    revolvingCreditLineInstanceA.updateRate(positionBob, 0.1 ether);

    revolvingCreditLineInstanceA.updateRate(positionCharlie, 0.09 ether);
    vm.stopPrank();
```

Figure 12.4: Bob and Charlie update the rate of their positions

Alice borrows 10 ether for the next loan cycle. Bob's position is left unborrowed and he attempts to withdraw; however, this reverts with a division-by-zero error.

```
    vm.prank(borrower);
    revolvingCreditLineInstanceA.borrow(borrower, 10 ether);

    vm.expectRevert();
    vm.prank(lenderOne);
    revolvingCreditLineInstanceA.withdraw(positionBob, type(uint256).max);
```

Figure 12.5: Alice borrows and Bob attempts to withdraw his unborrowed position

Since Bob's position was accounted for in the base epoch, but the `position.epochId` was updated as if it was in the new epoch due to `updateRate`, the system accounting breaks for his position. Bob is prevented from performing any action on his position until his position is borrowed or the loan is repaid.

Recommendations

Short term, consider updating the `getPositionRepartition` function to correctly account for positions that are deposited into, or whose rate was updated to, a previously borrowed rate. This could be done by adding a case where `position.epochId == tick.currentEpochId`. Ensure that the code is properly tested after this change.

Long term, consider simplifying the way epochs are accounted for in the system. Improve unit test coverage and create a list of system properties that can be tested with smart contract fuzzing.

13. Detached positions are incorrectly calculated

Severity: **Medium**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-ATL-13
Target: `TickLogic.sol`	

Description

Lenders who detach their position earn less interest in subsequent loan cycles due to an incorrect calculation in `getAdjustedAmount`.

The protocol allows users whose position has not been fully borrowed to `detach` their position and withdraw their leftover unborrowed assets. To avoid storing the state of each individual position, the protocol derives the position's borrowed and unborrowed amounts. This is done via the `adjustedAmount` and the total available and borrowed amount of that interest rate, as shown below:

```
    if (position.epochId <= tick.loanStartEpochId) {
        uint256 adjustedAmount = getAdjustedAmount(tick, position, referenceLoan);
        unborrowedAmount =
tick.baseEpochsAmounts.available.mul(adjustedAmount).div(
            tick.baseEpochsAmounts.adjustedDeposits
        );
        borrowedAmount = tick.baseEpochsAmounts.borrowed.mul(adjustedAmount).div(
            tick.baseEpochsAmounts.adjustedDeposits
        );
        return (unborrowedAmount, borrowedAmount);
    }
```

Figure 13.1: The `getPositionRepartition` function in `TickLogic.sol`

The `getAdjustedAmount` function has a special case for calculating detached positions using the position information and the `endOfLoanYieldFactor` of the loan in which the position was detached:

```
    if (position.withdrawLoanId > 0) {
        uint256 lateRepayFees = accrualsFor(
            position.withdrawn.borrowed,
            referenceLoan.lateRepayTimeDelta,
            referenceLoan.lateRepayFeeRate
        );
        uint256 protocolFees = (position.withdrawn.expectedAccruals +
lateRepayFees).mul(
            referenceLoan.repaymentFeesRate
        );
        return
            adjustedAmount = (position.withdrawn.borrowed +
                position.withdrawn.expectedAccruals +
                lateRepayFees -
           protocolFees).div(tick.endOfLoanYieldFactors[position.withdrawLoanId]);
    }
```

Figure 13.2: The `getAdjustedAmount` function in `TickLogic.sol`

However, this calculation returns an incorrect value, causing the protocol to consider the position's borrowed amount to be less than it actually is. This will result in the lender earning less interest than they should.

Exploit Scenario

A Revolving Credit Line is created for Alice with a maximum borrowable amount of 10 ether and a duration of 10 weeks.

1. Bob deposits 15 ether at a 10% rate.
2. Alice borrows 10 ether from the pool.
3. Bob detaches his position, withdrawing the unborrowed 5 ether to his account.
4. Alice repays the loan on time and takes another loan of 10 ether.
5. Although Alice took a 10 ether loan, the protocol considers Bob's position to be borrowed for 9.9998 ether.
6. Bob earns less accruals than he should for the duration of the loan and all subsequent loans.

Recommendations

Short term, investigate the ways in which detached position values are calculated and fees are subtracted in the system to ensure position values are derived correctly.

Long term, consider redesigning how fees are charged throughout the system so that derived values are not incorrectly impacted. Improve unit test coverage and come up with a list of system properties that can be tested with smart contract fuzzing. For example, this issue could have been discovered by implementing a property test that checks that the total borrowed amount should be equal to the sum of the borrowed amounts of all positions.

14. Borrower can reduce lender accruals

Severity: **Medium**	Difficulty: **Medium**
Type: Data Validation	Finding ID: TOB-ATL-14
Target: `TickLogic.sol`	

Description

A borrower can reduce the accrued interest of lenders by borrowing a smaller amount of assets multiple times.

The protocol allows borrowers to borrow any amount of assets up to the maximum borrowable amount defined for that product instance. Once the assets have been borrowed, the protocol uses the tick yield factor for each interest rate to calculate the accrued interest of lender positions. Whenever a borrow action occurs using a certain interest rate, the protocol calculates the new yield factor increase and adds it to the previous yield factor of that interest rate:

```
tick.yieldFactor += accrualsFor(
   amountBorrowed + accrualsAllocated,
   currentLoan.maturity - block.timestamp,
   rate
).div(tick.baseEpochsAmounts.adjustedDeposits);
```

Figure 14.1: `yieldFactor` increase in the `borrowFromBase` function in `TickLogic.sol`

However, since the `yieldFactor` has the same precision as the token and the function uses fixed-point division that rounds down, the yield factor increase will truncate and return a lower-than-expected value for lower-precision tokens (e.g., USDC). This precision loss is compounded during multiple borrow actions and leads to loss of accruals for lenders.

Exploit Scenario

A Revolving Credit Loan is created for Alice with a maximum borrowable amount of 10 million USDC and a duration of 10 weeks. Bob deposits 10 million USDC at a 10% rate.

If Alice were to borrow the entire amount at once and repay on time, at the end of the loan she would need to repay ~10,191,780 USDC and Bob would earn 10,191,780 USDC.

If Alice were to borrow 100,000 USDC one hundred times and repay on time, at the end of the loan she would have to repay ~10,191,780 USDC, but Bob would have earned only 10,191,000 USDC, or 780 fewer USDC than if the entire amount was borrowed at once.

`

Assuming a gas cost of 100 gwei and a block gas limit of 30 million gas, Alice can perform ~500 smaller borrow actions within the block gas limit, which would cost her approximately 3 MATIC or around 3.4 USD. If Alice filled two blocks with ~1,000 borrows total, it would reduce Bob's accruals by 1,780 USDC but would cost Alice only ~6.5 MATIC or ~7.3 USDC.

Recommendations

Short term, consider increasing the precision of the `yieldFactor` and evaluate which rounding direction should be used to minimize the impact of the precision loss.

Long term, analyze the system arithmetic for precision issues and decide on the rounding direction of each operation so that precision loss is minimized. Improve unit test coverage and thoroughly test the system using assets with different decimal precision. Use smart contract fuzzing to test system invariants. This issue could have been discovered by implementing a property test; with a total borrowed amount X and a duration Y, the total accruals should be equal to Z, no matter if the amount was borrowed at once or via multiple smaller amounts.

15. Borrower can start a lending cycle before deposits are made

Severity: **Informational**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-ATL-15
Target: `RCLBorrower.sol`	

Description

A borrower can initiate a loan before any deposits are made by borrowing 10 or fewer tokens.

The `borrow` function of the `RCLBorrower` implements a check for precision issues in an effort to avoid disabling the function due to compounding rounding errors.

```
if (remainingAmount > 10) revert RevolvingCreditLineErrors.RCL_NO_LIQUIDITY();
uint256 borrowedAmount = amount - remainingAmount;
```

Figure 15.1: `borrow` function in `RCLBorrower.sol`

However, this also allows the borrower to initiate a loan without any deposits being made. This is because the `remainingAmount` equals the amount passed into the function, which allows the `borrow` function to succeed even though the `borrowedAmount` is zero.

Recommendations

Short term, implement a check that ensures the `borrowAmount` is always greater than zero.

Long term, investigate and minimize any rounding issues present in the protocol. Explicitly specify preconditions and postconditions of all functions to more easily identify what is being checked and what needs to be checked in a function. Set up fuzzing tests with Echidna to identify unexpected behavior.

16. Documentation and naming conventions can be improved

Severity: **Informational**	Difficulty: **Low**
Type: Undefined Behavior	Finding ID: TOB-ATL-16
Target: `src/*`	

Description

Although the documentation is extensive, there are numerous areas where the documentation is either lacking or differs from the specifications in the codebase.

Mismatch between documentation and code

For example, regarding the `optOut` functionality, the documentation states that if a position is fully matched, a lender can `optOut`. However, it is ambiguous if this means a lender can `optOut` regardless of whether the position is fully matched or partially matched:

```
- Opt out: if their position is matched and the lender wants to signal their
intention to not be part of the next loan cycle, they can opt the position out. In
that case, the position will stop being available for borrowing at the end of the
current loan cycle. At that point, the user can withdraw its funds at any time.
```

Figure 16.1: Documentation regarding `optOut`

In the `validateOptOut` function implementation, the function reverts if the position has not been matched at all:

```
 function validateOptOut(
    DataTypes.Position storage position,
    DataTypes.Tick storage tick,
    DataTypes.Loan storage referenceLoan,
    DataTypes.Loan storage currentLoan
) external view {
    (, uint256 borrowedAmount) = getPositionRepartition(tick, position,
referenceLoan, currentLoan);

    if (borrowedAmount == 0) revert
RevolvingCreditLineErrors.RCL_POSITION_NOT_BORROWED();
    if (block.timestamp > currentLoan.maturity) revert
RevolvingCreditLineErrors.RCL_MATURITY_PASSED();
}
```

Figure 16.2: The `validateOptOut` function in `TickLogic.sol#L1131-L1141`

`

Obscure naming conventions in codebase
Some naming conventions used throughout the codebase are misleading and create confusion. For example, the `didPartiallyWithdraw` variable is ambiguous and implies that the current process is a partial withdrawal, when in reality the variable indicates whether a partial withdrawal has already taken place:

```
if (
    tick.borrowedAmount > 0 &&
    tick.borrowedAmount < tick.depositedAmount &&
    (poolPhase == LoanTypes.PoolPhase.ISSUED || poolPhase ==
LoanTypes.PoolPhase.NON_STANDARD)
) {
    uint256 unborrowedPart = depositedAmount.mul(tick.depositedAmount -
tick.borrowedAmount).div(
        tick.depositedAmount
    );
    unborrowedAmount = didPartiallyWithdraw ? 0 : unborrowedPart;
    borrowedAmount = depositedAmount - unborrowedPart;
    partialWithdraw = true;
    return (unborrowedAmount, borrowedAmount, partialWithdraw);
}
```

Figure 16.3: The code block containing the `didPartiallyWithdraw` variable in `LoanLogic.sol#L223-L235`

Incorrect variable names
The `withdrawLoanId` is meant to be used only in the `detach` function; however, its name suggests that it should also be used to track withdrawn loans:

```
function detach(
    DataTypes.Tick storage tick,
    DataTypes.Position storage position,
    DataTypes.Loan storage referenceLoan,
    DataTypes.Loan storage currentLoan,
    uint256 minDepositAmount
) external returns (uint256 unborrowedAmount) {
[...]
position.withdrawn = DataTypes.WithdrawalAmounts({borrowed: borrowedAmount,
expectedAccruals: accruals});
    position.withdrawLoanId = currentLoan.id;
}
```

Figure 16.4: The `withdrawLoanId` variable in the `detach` function in `TickLogic.sol#L533-L581`

The `deposit` function of the `LoanLender` contract uses the `CREATION_TIMESTAMP` to validate that the `BOOK_BUILDING` phase is still ongoing, as shown in figure 16.5. However, the `BOOK_BUILDING` phase does not actually start until the borrower executes the `enableBookBuildingPhase` function. This means that

`BOOK_BUILDING_PERIOD_DURATION` defines the maximum duration of the period instead of the actual duration, as its name suggests.

```
if (
    poolPhase != LoanTypes.PoolPhase.BOOK_BUILDING ||
    block.timestamp > CREATION_TIMESTAMP + BOOK_BUILDING_PERIOD_DURATION
) revert LoanErrors.LOAN_ALLOWED_ONLY_BOOK_BUILDING_PHASE();
```

Figure 16.5: The `deposit` function in `LoanLender.sol`

```
function enableBookBuildingPhase() external onlyBorrower
onlyInPhase(LoanTypes.PoolPhase.INACTIVE) {
    poolPhase = LoanTypes.PoolPhase.BOOK_BUILDING;
```

Figure 16.6: The `enableBookBuildingPhase` function in `LoanBorrower.sol`

Duplicate variable names

The duplicate use of variable names, even when referring to different values, creates confusion and makes reviewing the codebase difficult. The constant `ONE` is reused throughout the codebase and meant to indicate the decimals used for fixed-point calculations; however, it is assigned different values, such as `10**token.decimals()` or `10**18`. This makes it difficult to track the correct fixed point denominator.

Recommendations

Short term, update the documentation to correctly match the specification, and rename variables to provide clarity and remove ambiguity.

Long term, follow the following known system risks and limitations to keep documentation as close to the code as possible. Ensure that this behavior can be derived explicitly from naming conventions:

- Ensure implementation remains consistent with the specification and code comments
- Differentiate documentation between end-users and developers
- Cover all expected and unexpected behavior
- Follow similar naming conventions between code and specifications

17. Missing validation in detach

Severity: **Informational**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-ATL-17
Target: `RCLLenders.sol`	

Description

The `detach` function lacks validation that the position being detached has been partially borrowed.

The `detach` function is intended to allow lenders to withdraw their unborrowed assets from a partially borrowed position. The function fetches the unborrowed and borrowed amounts of the position and performs validation on them:

```
542    (unborrowedAmount, borrowedAmount) = getPositionRepartition(tick, position,
referenceLoan, currentLoan);
543    if (unborrowedAmount == 0) revert
RevolvingCreditLineErrors.RCL_POSITION_FULLY_BORROWED();
544
545    uint256 endOfLoanPositionValue = getPositionEndOfLoanValue(tick, position,
referenceLoan, currentLoan);
546
547    if (endOfLoanPositionValue - unborrowedAmount < minDepositAmount)
548       revert RevolvingCreditLineErrors.RCL_REMAINING_AMOUNT_TOO_LOW();
```

Figure 17.1: The `detach` function in `TickLogic.sol`

The execution will continue until it reaches the highlighted line in figure 17.2, at which point it will revert due to a division-by-zero error, since the `endOfLoanValue` is equal to zero:

```
162    uint256 endOfLoanValue = epoch.deposited + accruals - protocolFees;
163
164    equivalentYieldFactor =
tick.endOfLoanYieldFactors[epoch.loanId].mul(epoch.deposited).div(endOfLoanValue);
```

Figure 17.2: `getPositionEndOfLoanValue` function in `TickLogic.sol`

Although the function validates that a position is borrowed, it may not properly validate all cases. If a lender attempts to `detach` a completely unborrowed position, the function will revert before reaching the validation on line 547 of figure 17.1.

Recommendations

Short term, add validation to the `detach` function that reverts with a descriptive error if the position being detached has not been borrowed.

Long term, explicitly specify preconditions and postconditions of all functions to more easily identify what is being checked and what needs to be checked in a function. Set up fuzzing tests with Echidna to identify unexpected behavior. This issue could have been discovered by implementing a unit test that checks that detaching an unborrowed position reverts with the proper error message.

18. Contract architecture is overcomplicated

Severity: **Informational**	Difficulty: **Low**
Type: Configuration	Finding ID: TOB-ATL-18
Target: `throughout the codebase`	

Description

The codebase relies on a significant amount of cross-contract validation, obscure storage solutions, and a series of confusing branches, which obscures the understandability of the codebase.

For example, the `getPositionCurrentValue` function in the `TickLogic` contract determines the value, depending on the state of the position. As a result, the number of branches becomes difficult to follow and reason about. This pattern is repeated across the codebase, which is especially problematic with conversions to different precision types.

```
if OPTED_IN and REPAID
   RETURN getAdjustedAmount(tick.position, referenceLoan) • end of loan yield factor

if WITHDRAWS:
  if before maturity:
      timeUntilMaturity = maturity - timestamp
      factor = - accrualsFor(withdraw borrow, timeUntilMaturity, position rate)
   otherwise (a user is paying late):
      lateRepay = timestamp - current maturity
      factor = accrualsFor(withdraw borrow, lateRepayTimeDelta, lateRepayFeeRate)
   RETURN withdraw borrow + withdraw expected accrual + factor

if new epoch has started && (loan has borrowed in BASE_EPOCH OR borrowed in new
epoch)
   amount = epoch.borrowed
   timeDelta = time to maturity (if before maturity) OR time since maturity (if
late)
   rate = position rate (if before maturity) OR lateRepayFee (if late)
   differenceInAccrual = accruals ( amount, timeDelta, rate)
   if before maturity (i.e: paying early):
      factor = accruals - differenceInAccrual
   else (late payment):
      factor = -(accruals + lastRepayFees)
   RETURN base deposit + (factor * baseDeposit) / epoch.deposited

reference timestamp = non standard repayment time (if exists) OR current timestamp
timediff = reference timestamp - current loan maturity (if late) OR current loan
maturity - reference timestamp
```

`

```
rate = lateRepayFee (if late) or position.rate
adjustment factor = calculateYieldFactorIncrease(tick, timediff, rate)
if before maturity (i.e: paying early):
   new yield factor += adjustment factor
else (late payment):
   new yield factor -= adjustment factor
RETURN getAdjustedAmount(tick, position, referenceLoan) * new yield factor
```

Figure 18.1: Pseudocode of the `getPositionCurrentValue` function to track different branches

Recommendations

Short term, wherever possible, simplify functions to be as small and self-contained as possible and clearly document all expectations in the codebase.

Long term, thoroughly document the expected flow of each contract, the contracts' expected interactions with each other, and function call stacks to ensure that all contracts are easy to understand.

19. Governance is a single point of failure

Severity: **High**	Difficulty: **High**
Type: Access Controls	Finding ID: TOB-ATL-19
Target: `throughout the codebase`	

Description

Because the governance role is responsible for critical functionalities, it constitutes a single point of failure within the Loans and Revolving Credit Lines.

The role can perform the following privileged operations:

- Registering and deploying new products
- Setting economic parameters
- Entering the pool into a non-standard payment procedure
- Setting the phases of the Loans and Revolving Credit Lines
- Proposing and executing timelock operations
- Granting and revoking lender and borrower roles, respectively
- Changing the yield providers used by the Pool Custodian

These privileges give governance complete control over the protocol and critical protocol operations. This increases the likelihood that the governance account will be targeted by an attacker and incentivizes governance to act maliciously.

Exploit Scenario

Eve, a malicious actor, manages to take over the governance address. She then changes the yield providers to her own smart contract, effectively stealing all the funds in the Atlendis Labs protocol.

Recommendations

Short term, consider splitting the privileges across different addresses to reduce the impact of governance compromise. Ensure these powers and privileges are kept as minimal as possible.

Long term, document an incident response plan (see Appendix H) and ensure that the private keys for the multisig are managed safely (see Appendix G).

20. Pool is put in NON_STANDARD state only after executeTimelock() is called

Severity: **Informational**	Difficulty: Medium
Type: Timing	Finding ID: TOB-ATL-20
Target: RCLGovernance.sol	

Description

In the event of a protocol error, the RCLGovernance contract has the ability to start a non-standard payment procedure or a rescue operation. This will instead use the NonStandardPaymentModule to handle the repayment procedure, or allow the protocol to transfer all the funds to the timelock recipient.

Since this is a sensitive operation, the governance contract must go through a delay using a time-locked procedure. The first step of the procedure is to call the startNonStandardPaymentProcedure/startRescueProcedure function, which will call the initiate function on the Timelock contract. After a delay, the executeTimelock function will be called, and all the funds will be transferred to the timelock.recipient. The pool will then be set to the NON_STANDARD state, disabling any borrowing/lending functionality.

However, the pool will be set to the NON_STANDARD state only after the executeTimelock function has been called. This means that, during the delay between the call to initiate and the call to executeTimelock, the functionality of the pool contract will be the same. This could allow an attacker to exploit the contracts.

```
function executeTimelock() external onlyGovernance {
    timelock.execute();

    uint256 withdrawnAmount = CUSTODIAN.withdrawAllDeposits(timelock.recipient);

    if (timelock.timelockType == TimelockType.NON_STANDARD_REPAY) {
        INonStandardRepaymentModule(timelock.recipient).initialize(withdrawnAmount);
    }

    currentLoan.nonStandardRepaymentTimestamp = block.timestamp;
    orderBookPhase = DataTypes.OrderBookPhase.NON_STANDARD;

    emit TimelockExecuted(withdrawnAmount);
}
```

Figure 20.1: The executeTimelock function in RCLGovernance.sol#L115-L128

Exploit Scenario
An issue is discovered within the RCL products. As a result, governance proposes the rescue operation and calls `startRescueProcedure`. However, Eve notices the call and finds a bug allowing her to drain the lenders' funds. Because the pool is not in the `NON_STANDARD` state, Eve can steal all the assets from the protocol.

Recommendations
Short term, create a new phase that allows lenders to withdraw assets during an emergency, and put the pool in this phase when starting a time-locked procedure.

Long term, document an incident response plan that indicates the actions taken in the event of protocol failure.

21. Detached positions cannot be exited during subsequent loans

Severity: **Medium**	Difficulty: **Low**
Type: Denial of Service	Finding ID: TOB-ATL-21
Target: `TickLogic.sol`	

Description

If a position has been detached during a previous loan cycle and is borrowed in a subsequent loan cycle, that position cannot be exited.

The protocol allows positions that can be fully matched to `exit` the loan, withdrawing the full amount of their deposit and the accumulated accruals. However, a position that has been detached during a previous lending cycle will not be able to `exit`; instead, the function will revert with an arithmetic underflow:

```
tick.withdrawnAmounts.toBeAdjusted -= endOfLoanPositionValue;
```

Figure 21.1: `registerExit` function in `TickLogic.sol`

This is because the `prepareTickForNextLoan` function in the `TickLogic` contract is called on each repayment, which in turn sets the `tick.withdrawnAmounts` to zero:

```
delete tick.withdrawnAmounts;
```

Figure 21.2: `prepareTickForNextLoan` function in `TickLogic.sol`

Due to this error, the lender will be prevented from exiting their position and will have to wait for the loan to be repaid to withdraw their assets.

Exploit Scenario

A Revolving Credit Line is created for Alice with a maximum borrowable amount of 100 ether and a duration of 10 weeks.

1. Bob deposits 20 ether into the pool at a 10% rate, and Alice borrows 10 ether.
2. Bob assumes Alice will not borrow any more assets and detaches his position so he can utilize the assets elsewhere. He receives 10 ether from the position.
3. Alice repays the loan and again borrows 10 ether.
4. Charlie deposits 10 ether into the pool at a 10% rate.
5. Bob decides to exit his position. However, when he executes the `exit` function, it reverts with an arithmetic underflow.

Recommendations

Short term, consider revising how detached positions are considered in the system during multiple loan cycles.

Long term, improve unit test coverage and create a list of system properties that can be tested with smart contract fuzzing. For example, this issue could have been discovered by implementing a property test that checks that calling `exit` does not revert if all the preconditions are met.

22. Roles manager can never be updated

Severity: **High**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-ATL-22
Target: `Managed.sol`	

Description

The Atlendis Labs protocol uses role-based access control to manage permissions and privileges. The manager of the roles is responsible for granting and revoking roles; if necessary, the role manager can be transferred to another address by governance via the `updateRolesManager` function. However, the `updateRolesManager` function will always revert:

```
modifier onlyGovernance() {
    if (!rolesManager.isGovernance(msg.sender)) revert ONLY_GOVERNANCE();
    _;
}

/**
 * @inheritdoc IManaged
 */
function updateRolesManager(address _rolesManager) external onlyGovernance {
    if (rolesManager.isGovernance(msg.sender)) revert ONLY_GOVERNANCE();
    rolesManager = IRolesManager(_rolesManager);
    emit RolesManagerUpdated(address(rolesManager));
}
```

Figure 22.1:The `onlyGovernance` modifier and `updateRolesManager` function in `Managed.sol#L25-L37`

Exploit Scenario

The `PoolTokenFeesController` contract is deployed with an incorrect `rolesManager`. Governance notices the issue and attempts to call `updateRolesManager` with the correct `_rolesManager` address. However, the call will revert because the `if` statement in the `onlyGovernance` modifier and the `if` statement in `updateRolesManager` are contradictory.

Recommendations

Short term, remove the `if` statement, as the condition it checks is incorrect, and use just the `onlyGovernance` modifier instead.

Long term, improve unit test coverage and create a list of system properties that can be tested with smart contract fuzzing.

23. Risks with transaction reordering

Severity: **Informational**	Difficulty: **High**
Type: Timing	Finding ID: TOB-ATL-23
Target: `throughout the codebase`	

Description

Throughout the codebase, there are opportunities to reorder transactions to maximize the borrowable amounts of the borrower, grief lender actions, or front-run borrow actions.

- A user could monitor the mempool for a borrow action and deposit their own assets into the pool at a lower rate than other lenders. By doing so, the user could ensure that their position would earn the maximum amount of interest.
- A miner/validator could reorder transactions in a block such that their deposit is complete before the borrow action has executed, while the other lender's deposits could be moved to follow the borrow action. This would ensure that the miner/validator's position earns the maximum amount of interest.
- If a lender wants to exit or detach their position, a borrower could notice the transaction in the mempool and front-run them with another borrow transaction, preventing them from withdrawing some or all of their assets.

Since the time of the deposit has no influence on the order in which the deposits will be borrowed (as long as they are in the same interest rate and epoch), lenders are incentivized to deposit their assets as close to the borrow action as possible.

```
function deposit(
    DataTypes.Tick storage tick,
    DataTypes.Loan storage currentLoan,
    uint256 amount
) public {
    if ((currentLoan.maturity > 0) && (tick.latestLoanId == currentLoan.id)) {
        tick.newEpochsAmounts.toBeAdjusted += amount;
        tick.newEpochsAmounts.available += amount;
        tick.epochs[tick.currentEpochId].deposited += amount;
    } else {
        tick.baseEpochsAmounts.available += amount;
        tick.baseEpochsAmounts.adjustedDeposits += amount.div(tick.yieldFactor);
    }
}
```

Figure 23.1: The `deposit` function in `TickLogic.sol#L415-L428`

Exploit Scenario

A Revolving Credit Line is created for Alice with a maximum borrowable amount of 100 ether. Bob deposits 20 ether into the pool, and Alice borrows 10 ether. After some time has passed, Bob decides to detach his position to withdraw the unborrowed amount. Alice sees this transaction in the mempool and front-runs it so Bob's position ends up being fully borrowed, preventing him from withdrawing the assets.

Recommendations

Short term, analyze opportunities for transaction reordering within the current codebase and examine the potential implications of reordering state-changing actions.

Long term, thoroughly document the risks of transaction reordering, or refactor the codebase to mitigate them if possible.

24. Problematic approach to the handling precision errors	
Severity: **Informational**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-ATL-24
Target: `throughout the codebase`	

Description

Many functions in the TickLogic contract contain obscure precision checks. These checks may protect against a subset of unexpected behavior, but are problematic when considering the full range of inputs.

One such example is in the RCLBorrower contract's `borrow` function. After iterating over the ticks and updating respective amounts, the `RCLBorrower` contract checks the liquidity remaining after adjustment. If the amount is less than or equal to 10, the execution of the `borrow` function continues; otherwise, it halts. This effectively allows users to borrow an additional 10 units of the tokens:

```
while (remainingAmount > 0 && rate <= MAX_RATE) {
    (uint256 tickBorrowedAmount, , ) = TickLogic.borrow(
        ticks[rate],
        currentLoan,
        DataTypes.BorrowInput({totalAmountToBorrow: remainingAmount,
totalAccrualsToAllocate: 0, rate: rate})
    );
    remainingAmount -= tickBorrowedAmount;

    totalToBeRepaid +=
        tickBorrowedAmount +
        TickLogic.accrualsFor(tickBorrowedAmount, currentLoan.maturity -
block.timestamp, rate);

    rate += RATE_SPACING;
}
// @dev precision issue
if (remainingAmount > 10) revert RevolvingCreditLineErrors.RCL_NO_LIQUIDITY();
```

Figure 24.1: The `borrow` function in `TickLogic.sol#L115-L128`

When a user withdraws, similar logic applies to the withdrawal amounts, as defined in Figure 24.2. This allows users to withdraw up to 10 tokens for free.

```
} else {
    if (
        withdrawnAmount > tick.baseEpochsAmounts.available + 10 ||
        adjustedAmountToWithdraw > tick.baseEpochsAmounts.adjustedDeposits + 10
    ) revert RevolvingCreditLineErrors.RCL_NOT_ENOUGH_ADJUSTED_DEPOSITS();
    withdrawnAmount = tick.baseEpochsAmounts.available;
    tick.baseEpochsAmounts.available = 0;
    tick.baseEpochsAmounts.adjustedDeposits = 0;
}
```

Figure 24.2: The `withdraw` function in `TickLogic`

This check is also present in the `exit` function of the `TickLogic` contract, which gives users up to 10 free tokens upon exiting their position:

```
// @dev precision issue
if (endOfLoanBorrowEquivalent > 10) revert
RevolvingCreditLineErrors.RCL_NO_LIQUIDITY();
```

Figure 24.3: The `executeTimelock` function in `RCLGovernance.sol#L115-L128`

This attack can be significantly cheaper on layer two networks, where gas costs allow attackers to submit many transactions and profit from a series of deposits or withdrawals.

Recommendations

Short term, avoid comparing against magic values and always ensure that any deposits into the protocol round up to maximize the amount a user pays, and that any withdrawals or exits round down to minimize the amount a user receives. Solving precision errors through rounding will ensure that dust, or small amounts of imprecision, does not allow attackers to profit from interacting with the system.

Long term, analyze all formulas in the system to ensure that they follow accurate rounding directions, as explained in Appendix C: Rounding Recommendations.

25. Lenders with larger deposits earn less accruals if their position is only partially borrowed

Severity: **Medium**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-ATL-25
Target: `TickLogic.sol`	

Description

If a lender deposits more assets than the amount that ends up being borrowed from their position, they will receive fewer accruals than if they had deposited exactly the amount that ends up being borrowed.

The protocol allows lenders to deposit any amount of assets at any allowed interest rate, as long as the amount is larger than the minimum deposit amount. It uses the interest rate yield factor to calculate the amount of interest/accruals the lender has earned during the lending cycle. Whenever a borrow is triggered, a yield factor increase is calculated and added to the previous `yieldFactor`:

```
tick.yieldFactor += accrualsFor(
   amountBorrowed + accrualsAllocated,
   currentLoan.maturity - block.timestamp,
   rate
).div(tick.baseEpochsAmounts.adjustedDeposits);
```

Figure 25.1: `yieldFactor` increase in `borrowFromBase` in `TickLogic.sol`

However, the larger the amount in `tick.baseEpochsAmounts.adjustedDeposits`, the less the yield factor will increase due to rounding errors. This will lead to lenders earning less interest if their deposit is significantly larger than the borrower amount.

Exploit Scenario

An RCL is created for Alice with a maximum borrowable amount of 30 million USDC and a duration of 10 weeks.

Consider the following two cases:

1. Bob deposits 10 million USDC, and his position is fully borrowed. At the end of the loan, Bob has earned 191,780 USDC.
2. Bob deposits 30 million USDC, and his position is borrowed for 10 million USDC. At the end of the loan, Bob has earned 191,760 USDC.

Bob has earned 20 USDC less in case two than in case one.

Recommendations

Short term, investigate the cause of the rounding error and mitigate it. Thoroughly test the interest calculations using tokens with different decimal precisions and in various different scenarios.

Long term, analyze the system arithmetic for precision issues and decide on the rounding direction of each operation to minimize precision loss. Use smart contract fuzzing to test system invariants. This issue could have been discovered by implementing a property test that checks that the accrued interest of a lender is equal any time their position is borrowed for a certain amount, no matter the lender's deposited amount.

A. Vulnerability Categories

The following tables describe the vulnerability categories, severity levels, and difficulty levels used in this document.

Vulnerability Categories	
Category	Description
Access Controls	Insufficient authorization or assessment of rights
Auditing and Logging	Insufficient auditing of actions or logging of problems
Authentication	Improper identification of users
Configuration	Misconfigured servers, devices, or software components
Cryptography	A breach of system confidentiality or integrity
Data Exposure	Exposure of sensitive information
Data Validation	Improper reliance on the structure or values of data
Denial of Service	A system failure with an availability impact
Error Reporting	Insecure or insufficient reporting of error conditions
Patching	Use of an outdated software package or library
Session Management	Improper identification of authenticated users
Testing	Insufficient test methodology or test coverage
Timing	Race conditions or other order-of-operations flaws
Undefined Behavior	Undefined behavior triggered within the system

Severity Levels

Severity	Description
Informational	The issue does not pose an immediate risk but is relevant to security best practices.
Undetermined	The extent of the risk was not determined during this engagement.
Low	The risk is small or is not one the client has indicated is important.
Medium	User information is at risk; exploitation could pose reputational, legal, or moderate financial risks.
High	The flaw could affect numerous users and have serious reputational, legal, or financial implications.

Difficulty Levels

Difficulty	Description
Undetermined	The difficulty of exploitation was not determined during this engagement.
Low	The flaw is well known; public tools for its exploitation exist or can be scripted.
Medium	An attacker must write an exploit or will need in-depth knowledge of the system.
High	An attacker must have privileged access to the system, may need to know complex technical details, or must discover other weaknesses to exploit this issue.

B. Code Maturity Categories

The following tables describe the code maturity categories and rating criteria used in this document.

Code Maturity Categories	
Category	Description
Arithmetic	The proper use of mathematical operations and semantics
Auditing	The use of event auditing and logging to support monitoring
Authentication / Access Controls	The use of robust access controls to handle identification and authorization and to ensure safe interactions with the system
Complexity Management	The presence of clear structures designed to manage system complexity, including the separation of system logic into clearly defined functions
Cryptography and Key Management	The safe use of cryptographic primitives and functions, along with the presence of robust mechanisms for key generation and distribution
Decentralization	The presence of a decentralized governance structure for mitigating insider threats and managing risks posed by contract upgrades
Documentation	The presence of comprehensive and readable codebase documentation
Front-Running Resistance	The system's resistance to front-running attacks
Low-Level Manipulation	The justified use of inline assembly and low-level calls
Testing and Verification	The presence of robust testing procedures (e.g., unit tests, integration tests, and verification methods) and sufficient test coverage

Rating Criteria	
Rating	Description
Strong	No issues were found, and the system exceeds industry standards.
Satisfactory	Minor issues were found, but the system is compliant with best practices.
Moderate	Some issues that may affect system safety were found.

Weak	Many issues that affect system safety were found.
Missing	A required component is missing, significantly affecting system safety.
Not Applicable	The category is not applicable to this review.
Not Considered	The category was not considered in this review.
Further Investigation Required	Further investigation is required to reach a meaningful conclusion.

C. Rounding Recommendations

Atlendis Labs uses fixed-point arithmetic. The current strategy leads to rounding in incorrect directions, allowing attackers to benefit from dust, or small amounts of imprecision, and to misprice or steal assets. These issues indicate a need to test the system in greater depth (TOB-ATL-9, TOB-ATL-10). Atlendis Labs implements a series of rounding calculations that involve giving users free funds under precision errors. We always recommend applying a rounding direction (up or down) such that the pool benefits. The sections below are examples based on our assumptions of expected behavior in the pool, and Atlendis Labs should execute further analysis on these directions and implement accordingly.

Determining Rounding Direction

To determine how to apply rounding (whether up or down), consider the result of the expected output.

For example, the formula for calculating installment amount is determined by the following formula:

$$\text{installment amount} = \frac{loanAmount \times rate(1+rate)^{\#paymentPeriods}}{(1+rate)^{\#paymentPeriods} - 1}$$

In order to benefit the pool, the installment payment amount must tend toward a higher value (↑), as this amount is used in calculations to determine how much a user should pay the pool.

The rounding direction of the numerator indicates that the numerator should be as large as possible. In order to further increase the installment amount, the following criteria must apply:

- `loanAmount` must round up.
- `rate(1+rate)` must round up.
- `#paymentPeriods` must round up.

The rounding direction of the denominator indicates that this number should be as small as possible, such that the installment amount increases.

- `(1+rate)` must round down.
- `#paymentPeriods` must round down.

Combining the above arrows in rounding suggests that the mathematics in the following formula should round in the specified directions:

$$\text{installment amount } \uparrow = \frac{loanAmount \uparrow \times rate \uparrow (1 + rate) \uparrow^{\#paymentPeriods}}{(1 + rate) \downarrow^{\#paymentPeriods \downarrow} \downarrow -1} \uparrow$$

The analyses of formulas such as the installment amount should result in the creation of functions that apply floor and ceiling rounding to each operation. In the above example, this should result in the addition and use of four new functions: `mulUp`, `mulDown`, `divUp`, and `divDown`. (Note that, due to the use of the Taylor series for exponentiation, this will be encompassed within the aforementioned functions.) Considering the rounding direction in this way will ensure that the installment amounts that a user pays back to the pool always round up, thereby causing any imprecision loss to increase the pool's balance rather than the user's balance.

Rounding Subformulas

(1-x) vs (x-1) Rounding

Several operations require to compute $(1-x)$ or $(x-1)$. The following rules can be followed to apply the rounding:

- $(1-x)$
    - $(1-x) \uparrow$ requires $x \downarrow$
    - $(1-x) \downarrow$ requires $x \uparrow$
- $(x-1)$
    - $(x-1) \uparrow$ requires $x \uparrow$
    - $(x-1) \downarrow$ requires $x \downarrow$

Similar rounding techniques can be applied in all of the system's formulas to ensure that rounding always occurs in the direction that benefits the pool. Some system formulas are highlighted below that similarly identify the rounding direction across the variables. Many of these are implemented with the assumption that no fee calculations are present, in order to focus on pure analysis of the mathematical formulas.

Bullet Loans

Bullet Loan Repayment

The following formula calculates the amount that a user should pay back for borrowing `borrowedAmount` from the protocol.

$$\text{repayment Amount } \uparrow = \frac{\text{borrowed amount} \uparrow}{\text{discountFactor (rate, loan duration)} \downarrow} \downarrow$$

$$\text{discountFactor } \downarrow = \frac{\text{ONE} \downarrow}{\text{ONE } +(\text{ rate} \cdot \text{loanDuration })\uparrow}$$

Borrowing Rate Calculation

The following formula calculates the interest rate at which users borrow their funds.

$$borrowRate \uparrow = \frac{1}{amountBorrowed \downarrow} \min \uparrow_{amountBorrowedInTick_i} \uparrow \sum \uparrow tickRate \times_{amountBorrowedInTick_i}$$

Installment Loans

Installment Loan Repayment

The following formula calculates the amount a user should pay back during an installment loan of `loanAmount`.

$$\text{installment amount } \uparrow = \frac{loanAmount \uparrow \times rate \uparrow (1 + rate) \uparrow^{\#paymentPeriods}}{(1 + rate) \downarrow^{\#paymentPeriods \downarrow} \downarrow -1} \uparrow$$

Late Repayment Penalty Fee

The following formula calculates how much a user should pay in late/penalty fees for their monthly payment amount.

$$penaltyfee = \text{ monthly payment } \uparrow \times \left(\frac{\uparrow \text{ late repay fee}}{\text{365 days (a year)} \downarrow} \right)^{\#dayslate \uparrow}$$

Borrowing Fee

The following formula calculates the split of fees for the borrower fee into fees for the treasury and fees for the borrower.

$$\text{fee To Treasury} \uparrow = \text{borrow amount} \uparrow \times \text{ BORROW FEE} \uparrow$$

$$\text{borrowerAmount} \downarrow = \text{borrow amount} \downarrow \times (1 \text{ BORROW FEE } \uparrow) \downarrow$$

`

Rounding Results – Revolving Credit Lines

Borrow Rate

The following formula calculates the interest rate for the borrower.

$$borrowRate \uparrow = \frac{1}{amountBorrowed \downarrow} \min \uparrow_{amountBorrowedInTick_i} \uparrow \sum \uparrow tickRate \times_{amountBorrowedInTick_i}$$

Lender Position Value – Withdrawing after maturity

The following formula calculates the amount the position is worth when a lender attempts to withdraw after maturity.

$$\text{lender position } \downarrow = \text{ initial deposit } \downarrow \times \left(1 + \frac{\text{tick } \%}{12 \text{ months}} \downarrow\right)^{\# \text{months} \downarrow}$$

Exit Amount Formula

When users exit the system, the value calculated to send users should always round down.

$$\text{exitAmount} \downarrow = \text{borrow return} \downarrow (1 + \text{ rate} \downarrow) \downarrow (\% \text{ of loan complete})$$

Registering Borrowing Fee Calculation

As the fees calculate the amount of money to keep in the pool, this amount should be maximized by rounding up. This ensures that, in the case of precision errors, the fee amount skews toward the pool instead of the user.

$$fees \uparrow \ = \ amount \uparrow \ \times \ BORROWING\ FEES\ RATE \ (\uparrow)$$

Exit Fee Calculation

$$\text{if } \frac{t}{T} \leq 0.8$$

$$\text{exit fee } \uparrow = \text{ initial fee } \uparrow \times \left(\frac{\text{final fee end first cliff}\uparrow - \text{ initial fee} \downarrow}{\text{first cliff duration} \downarrow}\right) \times \left(\frac{\text{current time } t \uparrow}{maturity \downarrow} \uparrow\right)$$

$$\text{otherwise:}$$
$$\text{exit fee} \uparrow = \text{final fee end first cliff} + \left(\frac{\text{final fee end second cliff}\uparrow - \text{final fee end first cliff}\downarrow}{\text{final fee end second cliff}\downarrow}\right) \times \left(\frac{\text{current time } t\uparrow}{\text{maturity}\downarrow} \uparrow \text{ - first cliff duration} \downarrow\right) \uparrow$$

Adjusting Withdrawal Amounts – `TickLogic.withdraw`

The calculation for the adjusted amount to withdraw should round up, as it will decrease the `adjustedDeposits` relative to a tick. By using default truncation, this can round down to 0, which can result in an unchanged `adjustedDeposit` and adjusted available funds.

$$adjusted\ amount\ to\ withdraw \uparrow \ = \frac{withdrawn\ amount\uparrow}{yield\ factor\downarrow}$$

Continuous Rewards Module

Fixed Distribution Rate

$$\text{daily stake rewards} \downarrow = (\text{amount to be distributed per day}) \downarrow \times \left(\frac{\text{stakes position valued}\downarrow}{\text{total positions}\uparrow} \right) \downarrow$$

Term Rewards – Distribution Rate as a function of Locking Period

$$\text{position rewards} \downarrow = \text{position value} \downarrow * \text{distribution rate} \downarrow * \frac{\text{multiplier} \downarrow}{\text{base multiplier}\uparrow} \downarrow * \text{locking duration} \downarrow$$

where multiplier should round up, thereby:

$$\text{multiplier} = \text{base multiplier} \uparrow + (\text{max multiplier} \uparrow - \text{base multiplier}\downarrow) \uparrow * \frac{(\text{locking duration} \uparrow - \text{min locking duration} \downarrow)\uparrow}{(\text{max locking duration} \downarrow - \text{min locking duration} \uparrow)\downarrow} \uparrow$$

Term Rewards – Distribution Rate as a function of Locking Period and Lending Rate

where the multiplier corresponds to:

$$\text{multiplier} = \text{base multiplier} \uparrow + (\text{max multiplier} - \text{base multiplier}) \uparrow * \frac{(\text{locking duration} \uparrow - \text{min locking duration} \downarrow)\uparrow}{(\text{max locking duration} \downarrow - \text{min locking duration} \uparrow)\downarrow} \uparrow * \frac{(\text{max rate} \uparrow - \text{rate} \downarrow)}{\text{max rate} \downarrow} \uparrow$$

D. Simplification Recommendations

To simplify the codebase, we recommend making the following changes.

Standardize structures across the system

Currently, the code tracks data structures inside the base epoch differently from those outside of the base epoch. The base epochs track values based on variables, whereas future epochs track against a specific epoch structure. Wherever possible, we recommend standardizing data structures to ensure consistency across the codebase.

Move validation functions for variables closer to contracts using them

Certain instances in the codebase use validation functions that are outside of the core of the function. These include functions in `RCLLender` or `RCLBorrower` that use a helper function like `validateExit` inside the `TickLogic` contract. This design of functions makes it harder to track the validation that each function is performing, and makes the codebase harder to understand.

Split the tick logic into multiple contracts

The `TickLogic` contract is a monolithic helper contract that controls functions for registering operations, executing operations, and validation. The function should be split up into individual contracts to increase readability.

Calculate amounts based on current values instead of calculating adjustments

The accounting in the RCL contracts is implemented with the assumption that amounts are constant. As this does not hold through the duration of a tick, the codebase must re-adjust to account for "realistic" values. This computation becomes difficult to reason about, as each scenario requires corner cases and further adjustments.

Simplify the Revolving Credit Lines (RCLs)

Many of this audit's findings relate to the RCLs. These findings resulted in unexpected behavior in the contracts due to the complex number of operations that borrowers and lenders can achieve during a loan cycle. In many examples, the mispricing or miscalculation by various functions had effects that were not immediately clear from the initial triage.

As a result, we recommend simplifying the RCL contracts to start with smaller iterations, and continue to build out the system once each feature is complete and thoroughly tested through unit and integration testing and fuzzing. As such, we are providing recommendations on simplifying the RCL contracts through introducing the following suggested versions. Features of each are expected to compound off the others.

Version 1

- Introduce a smaller expiration period for RCL products.
- Attribute all deposits and amounts to a single epoch.
- Implement maximum borrow limits for borrowers and lending limits for lenders.
- Only allow lenders to execute full withdrawals of their funds.
- Allow borrowers to revolve their credit once over the duration of the loan period.

Version 2

- Increase the expiration period for RCL products .
- Build infrastructure that handles the depositing and lending over two epochs.
- Increase maximum borrow limits and lending limits for lenders.
- Allow lenders to partially withdraw their funds over multiple epochs.
- Allow borrowers to revolve their credit twice over the duration of the loan period.

Version 3

- Implement additional support for multiple epochs, slowly increasing the number of epochs and ticks allowed.
- Increase the maximum borrow limit and lending limit to the intended value.
- Allow borrowers to revolve their credit multiple times over the duration of the loan period.
- Allow lenders to detach their positions.

E. System Invariant Recommendations

During the security review, we implemented Echidna tests to fuzz the Revolving Credit Lines. While Echidna helped find directly exploitable issues in the codebase, we identified other invariants that the fuzzing suite can be extended to include in order to find other unintended behavior. We also recommend extending the fuzzing tests to allow Echidna to create dynamic pools for testing.

Global system invariants

- [ ] The unborrowed/available amount on a Pool must be at most equal to the deposited amount in the Pool Custodian.
- [ ] The total borrowed amount should be equal to the sum of the borrowed amounts of all positions in the pool.

Access Controls

- [ ] Each pool contract has an associated role manager.
- [ ] Each pool has an associated fee controller.
- [ ] Governance address can withdraw fees from the Fee Controller.
- [ ] Governance can change the `RoleManager` contract.
- [ ] Governance can change the fee controller.
- [ ] Governance can change internal parameters of a pool.
- [ ] Governance can add reward modules.
- [ ] Governance cannot remove reward modules.

Revolving Credit Lines

- [ ] Calls to `currentAccruals` after a pool's maturity should always revert.
- [ ] An opted out position can never be reversed.
- [ ] With a total borrowed amount X and a duration Y, the total accruals should equal Z, no matter if the amount was borrowed at once or via multiple smaller amounts.

Reward Modules

- [ ] Once a new module is added, a user can update their stake to apply the generated rewards, without timelock.
- [ ] If a user has staked their position, they can update the rate whenever they want.
- [ ] If a user has staked their position, they can opt out of RCL loans.

Continuous Rewards Module

- [ ] A user receives collected rewards once they are claimed or unstaked.
- [ ] Specific to continuous ERC20 rewards: Pending rewards of the rewards module must be less than or equal to the balance of the token (i.e., the token associated with the reward module) in the rewards module contract address.

Terms Rewards Module

- ☐ Rewards cannot be redeemed before locking duration (exclusive).
- ☐ If a user unstakes their position before locking duration has passed, the user receives no rewards.
- ☐ In RCL, the user can configure specific duration intervals.
- ☐ In loans, the locking duration is the remaining time until the end of the book building phase plus the duration of the loan.
- ☐ Pending rewards of the rewards module must be less than or equal to the balance of the token (i.e., the token associated with the reward module) in the rewards module contract address.

Functional pre- and post-conditional invariants

LoanBase

`LoanBase.getPosition` (applicable to bullet, coupon bullet, and installment loans):

- ☐ A pool in the `BOOK_BUILDING` phase should return a position status of AVAILABLE.
- ☐ A pool in the `ISSUED` phase, if funds have not been borrowed, should return a position status of `UNAVAILABLE`.
- ☐ A pool in the `ISSUED` phase, if funds have been borrowed, should return a position status of `BORROWED`.
- ☐ A pool that meets none of the conditions specified above should return a position status of `UNAVAILABLE`.

LoanLender

`LoanLender.deposit(rate, amount, to)`

Preconditions:

- ☐ The pool must be in the `BOOK_BUILDING` phase.
- ☐ `block.timestamp` must be within `CREATION TIMESTAMP` + `BOOK_BUILDING_PERIOD_DURATION`.
- ☐ The amount to deposit must exceed `minDepositAmount`.
- ☐ The provided rate a lender wants to deposit at is bound between `[MIN_RATE, MAX_RATE]`.
- ☐ The delta of the new rate and the `MIN_RATE` of the pool must be divisible by `RATE_SPACING`.

Postconditions:

- ☐ The `LoanLogic.tick.depositedAmount` should increase by the amount deposited.
- ☐ `positionId` should increment by 1.

- ☐ Deposits should increase by `amount`.
- ☐ A new token should be minted to the address provided by lenders.
- ☐ `positions[positionId]` should be initialized with this position, where the following apply:
    - ☐ The `depositedAmount`, `rate`, `depositBlockNumber` aligns with the incoming arguments.
    - ☐ `unborrowedAmountWithdrawn = 0`
    - ☐ `numberOfPaymentsWithdrawn = 0`
- ☐ Token balance of the Pool Custodian increases by `amount`.
- ☐ Token balance of the lender decreases by `amount`.

`LoanLender.updateRate(), RCLLenders.updateRate()`

Preconditions:

- ☐ The caller must be the holder of the position.
- ☐ The pool must be in the BOOK_BUILDING phase.
- ☐ block.timestamp must be within `CREATION TIMESTAMP` + `BOOK_BUILDING_PERIOD_DURATION`.
- ☐ The provided rate a lender wants to deposit at is bound between `[MIN_RATE, MAX_RATE]`.

Postconditions:

- ☐ The old tick's deposited amount should decrease.
- ☐ The new tick's deposited amount should increase.
- ☐ Retrieving `positions[positionId].rate` should yield the new rate.
- ☐ Updating the rate will force an internal withdrawal and an immediate deposit. This should NOT break any pre/post condition for these operations.
- ☐ The amount deposited should not change during the update of the rate.

`LoanLender.withdraw(positionId)`

Preconditions:

- ☐ The caller must be the owner of the position.
- ☐ Deposit and withdrawal cannot occur within the same block.
- ☐ The pool phase must not be in `ORIGINATION`.
- ☐ The pool phase must not be in `NON_STANDARD`.
- ☐ The pool must have sufficient funds to execute withdrawals.
- ☐ The Pool Custodian must have sufficient deposits to cover the withdrawn amount.

Postconditions:

- ☐ If the pool phase is in `BOOK_BUILDING`, `tick.depositedAmount` should be decreased by the withdrawn amount.
- ☐ If the `unborrowedAmount = 0`, the position NFT should be burned.
- ☐ If the payments are complete, position NFT should be burned.
- ☐ If the pool phase is `CANCELED`, position NFT should be burned.
  Amount = `unborrowedAmount + paymentsToWithdraw  - fees`
- ☐ The pool custodian deposited balance should decrease by the withdrawn amount.
- ☐ The token balance of the Pool Custodian should decrease by the withdrawn amount.
- ☐ The token balance of the `msg.sender` should increase by the withdrawn amount.

`LoanLender.withdraw (positionId, amount)`

Preconditions

- ☐ Only possible within the `BOOK_BUILDING_PHASE`.
- ☐ The caller must be the owner of the position.
- ☐ The position deposit block number must not be equivalent to the current block number.
- ☐ The amount to withdraw must be less than the deposited amount.
- ☐ The amount to withdraw must be greater than the minimum deposited amount.
- ☐ The position's deposited amount, subtracted by the withdrawal amount, must be larger than or equal to the minimum deposit amount

Postconditions

- ☐ The tick for the position's rate should decrease by the amount to be withdrawn.
- ☐ The NFT must not be burned.
- ☐ Otherwise, merely decrease the `position.depositedAmount`.
- ☐ Decrease the deposit amount.

LoanBorrower

`LoanBorrower.borrow()`

Preconditions

- ☐ The block timestamp must be within origination phase start and duration.
- ☐ Deposits must not be zero.
- ☐ The `borrowedAmount` must be within the `MIN_ORIGINATION_AMOUNT` and the `TARGET_ORIGINATION_AMOUNT`.
  - ☐ If deposits < `MIN_ORIGINATION_AMOUNT`, all deposits can be borrowed.
- ☐ Borrowed amount must not exceed the deposit amount.
- ☐ After iterating ticks, the loan must have sufficient funds to execute the `borrow`.

Postconditions

- ☐ The pool phase is `ISSUED`.
- ☐ The borrow timestamp must be updated.
- ☐ The `totalCurrentBorrowed` must be updated.
- ☐ The `totalBorrowedStatic` must be updated.
- ☐ Borrowing fees must be registered within `FeeController`.
- ☐ The token balance of the borrower must have increased by `borrowedAmount -` fees.
- ☐ The token balance of the Pool Custodian must have decreased by `borrowedAmount` - fees.
- ☐ If `borrowedAmount` < `MIN_ORIGINATION_AMOUNT`, all deposits are borrowed.

`LoanBorrower.repay()`

Preconditions

- ☐ The loan must be issued.
- ☐ The timestamp must be between [borrow timestamp + loan duration - repayment period duration, borrow timestamp + loan duration].
- ☐ A later timestamp is subject to late repayment fees.
- ☐ The timestamp must precede the repay period duration.

Postconditions

- ☐ The token balance of `PoolCustodian` should increase by amount and fees.
- ☐ The custodian must receive approval for the amount repaid.

Revolving Credit Lines

`RCLBorrowers.borrow(address to, uint256 amount):`

Preconditions:

- ☐ `currentLoan.maturity` must be less than 0.
- ☐ `Block.timestamp` must be less than `currentLoan.maturity.`
- ☐ The amount + total borrowed amount must not exceed the maximum borrowable limit.
- ☐ The amount to borrow must not be set to 0.
- ☐ The `to` address must be a borrower's address.
- ☐ The caller must be a borrower.
- ☐ `OrderBookPhase` must be `OPEN`.

Postconditions:

- ☐ The borrowed amount must decrease by the remaining amount.

`

- ☐ `RCLBorrowers.totalBorrowed` should increase by `amount`.
- ☐ `RCLBorrowers.totalBorrowedWithSecondary` should increase by `amount`.
- ☐ The fees controller must have received borrowing fees.
    - ☐ `PoolTokenFeesController.dueFees` must have increased by `fee` amount.
    - ☐ `PoolTokenFeesController.totalFees` must have increased by `fee`.
- ☐ The token balance of the user must have increased by `amount`.
- ☐ The token balance of the Pool Custodian must have decreased by `amount` + `fees`.

`RCLBorrowers.repay()`

Preconditions

- ☐ `currentLoan.maturity` must not be zero.
- ☐ `Block.timestamp` must be greater or equal to the `currentLoan.maturity - REPAYMENT_PERIOD`.
- ☐ The caller must be a borrower.
- ☐ `OrderBookPhase` must be `OPEN`.

Postconditions

- ☐ If a borrower repays past `currentLoan.maturity`, the borrower must pay the `LATE_REPAYMENT_FEE_RATE`.
- ☐ Zero out `currentLoan.maturity`, `totalBorrowed`, `totalBorrowedWithSecondary`, `totalToBeRepaid`, and `totalRepaidWithSecondary`.

Pool Custodian

`PoolCustodian.deposit`

Preconditions

- ☐ The caller must be a pool.

Postconditions

- ☐ `depositedBalance` must increase.
- ☐ The token balance of the Pool Custodian must increase.
- ☐ The token balance of the pool should decrease.
- ☐ Calls by a non-pool address should always revert.
- ☐ Calls by a pool address should always succeed.

`PoolCustodian.withdraw()`

Preconditions

- ☐ The pool is the caller
- ☐ The deposited balance must be able to cover the withdrawal amount.

Postconditions

- ☐ The deposited balance decreases by the withdrawn amount.
- ☐ The token balance of the Pool Custodian decreases.
- ☐ The token balance of `to` increases.

`PoolCustodian.withdrawAllDeposits()`

Preconditions

- ☐ The pool is the caller.

Postconditions

- ☐ `depositedBalance` should be equivalent to 0.
- ☐ The token balance of the Pool Custodian should decrease by the full deposit amount.
- ☐ The token balance of `to` should increase by the full deposit amount.

`PoolCustodian.collectRewards()`

Postconditions

- ☐ `pendingRewards` increases by the collected amount.
- ☐ `generatedRewards` increases by the collected amount.

`PoolCustodian.withdrawRewards()`

Preconditions

- ☐ The caller must be the rewards operator.
- ☐ There must be sufficient pendingRewards to cover the `withdrawal amount`.

Postconditions

- ☐ The pending reward decreases by the withdrawn amount.
- ☐ The token balance of the Pool Custodian decreases by the withdrawn amount.
- ☐ The token balance of the Pool Custodian increases by the withdrawn amount.

`PoolCustodian.startSwitchYieldProviderProcedure`

Preconditions

- ☐ The provided delay must be greater than `SWITCH_YP_MIN_TIMELOCK_DELAY`.
- ☐ `newAdapter` must pass `supportsInterface` check

Postconditions

- ☐ The timelock delay must have started for the new yield provider.

F. Code Quality Recommendations

Trail of Bits recommends the following steps to enhance code quality.

- **Make use of helper functions or modifiers to enable code reuse across the system.**
  - `PoolTimelockLogic.sol#L50`
  - `LoanLender.sol#L114`
  - `LoanLender.sol#L52 and RewardsManager.sol#L85`
  - `Withdraw` and `withdrawAllDeposits()`
- **Move constant values to a helper library to de-duplicate identical variables.**
  - `RCLGovernance.sol#L28-29`
- **Consider refactoring `toggleRestrictions` to take an incoming Boolean argument.**
- **Standardize the use of modifiers for access controls across the system.**
  - Currently, `PoolCustodian.sol#L211` uses an `if` statement for access controls, whereas the entire system uses modifiers.
- **Move validation functions closer to where the variables are being used in the code.** In `LoanLender`, the `validateRate` function is in place in the `LoanLender` contract; however all RCL operations are put into `TickLogic`. This distribution of functions makes the codebase slightly more difficult to reason about.
- **Use a consistent naming convention for constant, immutable, and regular state variables.**
- **Consider refactoring struct deletion to use a single delete in places where the entire struct type variable is being deleted.**
  - `CustodianTimelockLogic.sol#L57-L60`
  - `PoolTimelockLogic.sol#L62-L65`
- **Remove redundant return statements to enhance readability.**
  - `LoanLogic.sol#L268`

G. Multisignature Wallet Best Practices

Consensus requirements for sensitive actions, such as spending the funds in a wallet, are meant to mitigate the risk of the following actions:

- Any one person's judgment overruling the others',
- Any one person's mistake causing a failure, and
- The compromise of any one person's credentials causing a failure.

In a 2-of-3 multisignature Ethereum wallet, for example, the execution of a "spend" transaction requires the consensus of two individuals in possession of two of the wallet's three private keys. For this model to be useful, it must fulfill the following requirements:

1. The private keys must be stored or held separately, and access to each one must be limited to a different individual.
2. If the keys are physically held by third-party custodians (e.g., a bank), multiple keys should not be stored with the same custodian. (Doing so would violate requirement #1.)
3. The person asked to provide the second and final signature on a transaction (i.e., the co-signer) ought to refer to a pre-established policy specifying the conditions for approving the transaction by signing it with his or her key.
4. The co-signer also ought to verify that the half-signed transaction was generated willfully by the intended holder of the first signature's key.

Requirement #3 prevents the co-signer from becoming merely a "deputy" acting on behalf of the first signer (forfeiting the decision-making responsibility to the first signer and defeating the security model). If the co-signer can refuse to approve the transaction for any reason, the due-diligence conditions for approval may be unclear. That is why a policy for validating transactions is needed. A verification policy could include the following:

- A protocol for handling a request to co-sign a transaction (e.g., a half-signed transaction will be accepted only via an approved channel)
- An allowlist of specific Ethereum addresses allowed to be the payee of a transaction
- A limit on the amount of funds spent in a single transaction, or in a single day

Requirement #4 mitigates the risks associated with a single stolen key. For example, say that an attacker somehow acquired the unlocked Ledger Nano S of one of the signatories.

A voice call from the co-signer to the initiating signatory to confirm the transaction would reveal that the key had been stolen and that the transaction should not be co-signed. If the signatory were under an active threat of violence, he or she could use a “duress code” (a code word, a phrase, or another signal agreed upon in advance) to covertly alert the others that the transaction had not been initiated willfully, without alerting the attacker.

H. Incident Response Recommendations

In this section, we provide recommendations around the formulation of an incident response plan.

Identify who (either specific people or roles) is responsible for carrying out the mitigations (deploying smart contracts, pausing contracts, upgrading the front end, etc.).

- Specifying these roles will strengthen the incident response plan and ease the execution of mitigating actions when necessary.

Document internal processes for situations in which a deployed remediation does not work or introduces a new bug.

- Consider adding a fallback scenario that describes an action plan in the event of a failed remediation.

Clearly describe the intended process of contract deployment.

Consider whether and under what circumstances Atlendis Labs will make affected users whole after certain issues occur.

- Some scenarios to consider include an individual or aggregate loss, a loss resulting from user error, a contract flaw, and a third-party contract flaw.

Document how Atlendis Labs plans keep up to date on new issues, both to inform future development and to secure the deployment toolchain and the external on-chain and off-chain services that the system relies on.

- For each language and component, describe noteworthy sources for vulnerability news. Subscribe to updates for each source. Consider creating a special private Discord channel with a bot that will post the latest vulnerability news; this will help the team keep track of updates all in one place. Also consider assigning specific team members to keep track of the vulnerability news of a specific component of the system.

Consider scenarios involving issues that would indirectly affect the system.

Determine when and how the team would reach out to and onboard external parties (auditors, affected users, other protocol developers, etc.) during an incident.

- Some issues may require collaboration with external parties to efficiently remediate them.

Define contract behavior that is considered abnormal for off-chain monitoring.

- Consider adding more resilient solutions for detection and mitigation, especially in terms of specific alternate endpoints and queries for different data as well as status pages and support contacts for affected services.

Combine issues and determine whether new detection and mitigation scenarios are needed.

Perform periodic dry runs of specific scenarios in the incident response plan to find gaps and opportunities for improvement and to develop muscle memory.

- Document the intervals at which the team should perform dry runs of the various scenarios. For scenarios that are more likely to happen, perform dry runs more regularly. Create a template to be filled in after a dry run to describe the improvements that need to be made to the incident response.

I. Token Integration Checklist

The following checklist provides recommendations for interactions with arbitrary tokens. Every unchecked item should be justified, and its associated risks understood. For an up-to-date version of the checklist, see `crytic/building-secure-contracts`.

For convenience, all Slither utilities can be run directly on a token address, such as the following:

```
slither-check-erc 0xdac17f958d2ee523a2206206994597c13d831ec7 TetherToken --erc erc20
slither-check-erc 0x06012c8cf97BEaD5deAe237070F9587f8E7A266d KittyCore --erc erc721
```

To follow this checklist, use the below output from Slither for the token:

```
slither-check-erc [target] [contractName] [optional: --erc ERC_NUMBER]
slither [target] --print human-summary
slither [target] --print contract-summary
slither-prop . --contract ContractName # requires configuration, and use of Echidna
and Manticore
```

General Considerations

- ❏ **The contract has a security review.** Avoid interacting with contracts that lack a security review. Check the length of the assessment (i.e., the level of effort), the reputation of the security firm, and the number and severity of the findings.

- ❏ **You have contacted the developers.** You may need to alert their team to an incident. Look for appropriate contacts on `blockchain-security-contacts`.

- ❏ **They have a security mailing list for critical announcements.** Their team should advise users (like you!) when critical issues are found or when upgrades occur.

Contract Composition

- ❏ **The contract avoids unnecessary complexity.** The token should be a simple contract; a token with complex code requires a higher standard of review. Use Slither's `human-summary` printer to identify complex code.

- ❏ **The contract uses `SafeMath`.** Contracts that do not use `SafeMath` require a higher standard of review. Inspect the contract by hand for `SafeMath` usage.

- ❏ **The contract has only a few non-token-related functions.** Non-token-related functions increase the likelihood of an issue in the contract. Use Slither's `contract-summary` printer to broadly review the code used in the contract.

- ❏ **The token has only one address.** Tokens with multiple entry points for balance updates can break internal bookkeeping based on the address (e.g., `balances[token_address][msg.sender]` may not reflect the actual balance).

Owner Privileges

- ❏ **The token is not upgradeable.** Upgradeable contracts may change their rules over time. Use Slither's `human-summary` printer to determine whether the contract is upgradeable.

- ❏ **The owner has limited minting capabilities.** Malicious or compromised owners can abuse minting capabilities. Use Slither's `human-summary` printer to review minting capabilities, and consider manually reviewing the code.

- ❏ **The token is not pausable.** Malicious or compromised owners can trap contracts relying on pausable tokens. Identify pausable code by hand.

- ❏ **The owner cannot blacklist the contract.** Malicious or compromised owners can trap contracts relying on tokens with a blacklist. Identify blacklisting features by hand.

- ❏ **The team behind the token is known and can be held responsible for abuse.** Contracts with anonymous development teams or teams that reside in legal shelters require a higher standard of review.

ERC20 Tokens

ERC20 Conformity Checks

Slither includes a utility, `slither-check-erc`, that reviews the conformance of a token to many related ERC standards. Use `slither-check-erc` to review the following:

- ❏ **`Transfer` and `transferFrom` return a Boolean.** Several tokens do not return a Boolean on these functions. As a result, their calls in the contract might fail.

- ❏ **The `name`, `decimals`, and `symbol` functions are present if used.** These functions are optional in the ERC20 standard and may not be present.

- ❏ **`Decimals` returns a `uint8`.** Several tokens incorrectly return a `uint256`. In such cases, ensure that the value returned is below 255.

- ❏ **The token mitigates the known ERC20 race condition.** The ERC20 standard has a known ERC20 race condition that must be mitigated to prevent attackers from stealing tokens.

Slither includes a utility, `slither-prop`, that generates unit tests and security properties that can discover many common ERC flaws. Use `slither-prop` to review the following:

- ❏ **The contract passes all unit tests and security properties from `slither-prop`.** Run the generated unit tests and then check the properties with Echidna and Manticore.

Risks of ERC20 Extensions

The behavior of certain contracts may differ from the original ERC specification. Conduct a manual review of the following conditions:

- ❏ **The token is not an ERC777 token and has no external function call in `transfer` or `transferFrom`.** External calls in the transfer functions can lead to re-entrancies.
- ❏ **`Transfer` and `transferFrom` should not take a fee.** Deflationary tokens can lead to unexpected behavior.
- ❏ **Potential interest earned from the token is taken into account.** Some tokens distribute interest to token holders. This interest may be trapped in the contract if not taken into account.

Token Scarcity

Reviews of token scarcity issues must be executed manually. Check for the following conditions:

- ❏ **The supply is owned by more than a few users.** If a few users own most of the tokens, they can influence operations based on the tokens' repartition.
- ❏ **The total supply is sufficient.** Tokens with a low total supply can be easily manipulated.
- ❏ **The tokens are located in more than a few exchanges.** If all the tokens are in one exchange, a compromise of the exchange could compromise the contract relying on the token.
- ❏ **Users understand the risks associated with a large amount of funds or flash loans.** Contracts relying on the token balance must account for attackers with a large amount of funds or attacks executed through flash loans.
- ❏ **The token does not allow flash minting.** Flash minting can lead to substantial swings in the balance and the total supply, which necessitate strict and comprehensive overflow checks in the operation of the token.

ERC721 Tokens

ERC721 Conformity Checks

The behavior of certain contracts may differ from the original ERC specification. Conduct a manual review of the following conditions:

- ❏ **Transfers of tokens to the `0x0` address revert.** Several tokens allow transfers to `0x0` and consider tokens transferred to that address to have been burned; however, the ERC721 standard requires that such transfers revert.
- ❏ **`safeTransferFrom` functions are implemented with the correct signature.** Several token contracts do not implement these functions. A transfer of NFTs to one of those contracts can result in a loss of assets.
- ❏ **The `name`, `decimals`, and `symbol` functions are present if used.** These functions are optional in the ERC721 standard and may not be present.
- ❏ **If it is used, `decimals` returns a `uint8(0)`.** Other values are invalid.
- ❏ **The `name` and `symbol` functions can return an empty string.** This behavior is allowed by the standard.
- ❏ **The `ownerOf` function reverts if the `tokenId` is invalid or is set to a token that has already been burned.** The function cannot return `0x0`. This behavior is required by the standard, but it is not always properly implemented.
- ❏ **A transfer of an NFT clears its approvals.** This is required by the standard.
- ❏ **The token ID of an NFT cannot be changed during its lifetime.** This is required by the standard.

Common Risks of the ERC721 Standard

To mitigate the risks associated with ERC721 contracts, conduct a manual review of the following conditions:

- ❏ **The onERC721`Received` callback is taken into account.** External calls in the transfer functions can lead to re-entrancies, especially when the callback is not explicit (e.g., in `safeMint` calls).

❏ **When an NFT is minted, it is safely transferred to a smart contract.** If there is a minting function, it should behave similarly to `safeTransferFrom` and properly handle the minting of new tokens to a smart contract. This will prevent a loss of assets.

❏ **The burning of a token clears its approvals.** If there is a burning function, it should clear the token's previous approvals.

J. Fix Review Results

When undertaking a fix review, Trail of Bits reviews the fixes implemented for issues identified in the original report. This work involves a review of specific areas of the source code and system configuration, not comprehensive analysis of the system.

From April 11 to April 13, 2023, Trail of Bits reviewed the fixes and mitigations implemented by the Atlendis Labs team for the issues identified in this report. We reviewed each fix to determine its effectiveness in resolving the associated issue.

In summary, of the 25 issues described in this report, Atlantis labs has resolved 20 issues, has partially resolved two issues, has one issue requiring further investigation, and has not resolved the remaining two issues. For additional information, please see the Detailed Fix Review Results below.

ID	Title	Severity	Status
1	Borrower can drain lender assets by withdrawing the cancellationFee multiple times	High	Resolved
2	Incorrect fee calculation on withdrawal can lead to DoS of withdrawals or loss of assets	High	Resolved
3	Lack of zero address checks	High	Resolved
4	Problematic approach to data validation	Medium	Resolved
5	Borrower can skip the last coupon payment	Medium	Resolved
6	Initialization functions can be frontrun	Informational	Partially Resolved
7	The lenders' unborrowed deposits can be locked up by a borrower	Medium	Unresolved
8	optOut can be called multiple times	High	Resolved

9	Risks with deflationary, inflationary or rebasing tokens	Medium	Partially Resolved
10	Rounding down when computing fees will benefit users	Low	Further investigation required
11	Lenders can prevent each other from earning interest	Medium	Resolved
12	Incorrect calculation in getPositionRepartition can lock a user's position	Medium	Resolved
13	Detached positions are incorrectly calculated	Medium	Resolved
14	Borrower can reduce lender accruals	Medium	Resolved
15	Borrower can start a lending cycle before deposits are made	Informational	Resolved
16	Documentation and naming conventions can be improved	Informational	Resolved
17	Missing validation in detach	Informational	Resolved
18	Contract architecture is overcomplicated	Informational	Resolved
19	Governance is a single point of failure	High	Resolved
20	Pool is put in NON_STANDARD state only after executeTimelock() is called	Informational	Resolved
21	Detached positions cannot be exited during subsequent loans	Medium	Resolved
22	Roles manager can never be updated	High	Resolved

23	Risks with transaction reordering	Informational	Unresolved
24	Problematic approach to the handling precision errors	Informational	Resolved
25	Lenders with larger deposits earn less accruals if their position is only partially borrowed	Medium	Resolved

Detailed Fix Review Results

TOB-ATL-1: Borrower can drain lender assets by withdrawing the cancellationFee multiple times
Resolved in PR 504. The `cancellationFeeEscrow` variable is now correctly set to 0 during withdrawal.

TOB-ATL-2: Incorrect fee calculation on withdrawal can lead to DoS of withdrawals or loss of assets
Resolved in PR 515. The `PoolTokenFeesController` contract now uses the token's decimals for precision instead of the fixed constant `10**18`.

TOB-ATL-3: Lack of zero-address checks
Resolved in PR 513. The codebase now has zero address checks when initializing variables. Furthermore, there are appropriate ERC165 `supportsInterface` checks for appropriate contracts.

TOB-ATL-4: Lack of zero-address checks
Resolved in PR 526. There are now two separate validation libraries that are called during contract construction in the Loan and RCL products, respectively.

TOB-ATL-5: Borrower can skip the last coupon payment
Resolved in PR 523. The conditional statement was updated to include the case where `referenceTimestamp == loanMaturity+repaymentPeriodDuration`.

TOB-ATL-6: Initialization function can be frontrun
Partially resolved. The documentation has been updated to reflect the fact that initialization functions can be front-run in PR 546, but the factory is still vulnerable to front-running.

TOB-ATL-7: Lenders' unborrowed deposits can be locked up by a borrower
Unresolved. Atlendis Labs provided the following context for this finding's fix status:

Fixing this would break our current data structures use, necessitating deeper changes, for an issue whose impact is limited. Lenders will have ways to get their deposit back if the case happens, like exiting, or waiting for the loan to end by opting their position out.

TOB-ATL-8: optOut can be called multiple times
Resolved in PR 505. The `validateOptOut` function now correctly reverts if `position.optOutLoanId>0`.

TOB-ATL-9: Risks related to deflationary, inflationary, or rebasing tokens
Partially resolved. Atlendis Labs has updated the documentation regarding deflationary, inflationary, or rebasing tokens in PR 506; however, the code has not been updated to explicitly disallow these tokens.

TOB-ATL-10: Rounding down when computing fees benefits users
Further investigation required. In PR 529, the rounding directions are now specified as rounding up/down depending on the operation to round in favor of the protocol. In particular, the registration of fees is rounded up, and most protocol inflows/outflows are also rounded in the appropriate direction. The change to determining the rounding direction is a great step forward.

However, some operations must be specifically rounded in the opposite direction (e.g., rounding borrows up) to prevent underflows. In addition, the arithmetic operations are influential throughout the codebase and are a critical component of the system. We were timeboxed in our review of all the possible errors that could occur due to precision loss, and we recommend implementing further testing, especially stateful fuzz testing, to ensure intended behavior.

TOB-ATL-11: Lenders can prevent each other from earning interest
Resolved in PR 506. The `updateRate` function will now revert if a position has been `optedOut`.

TOB-ATL-12: Incorrect calculation in getPositionRepartition can lock a user's position
Resolved in PR 510. If the position has been borrowed in the previous cycle and will end up not being borrowed in the next cycle, the `getPositionRepartition` function now correctly returns the `borrowedAmount` as 0.

TOB-ATL-13: Detached positions are incorrectly calculated
Resolved in PR 522. When fees are registered, the difference between a detached position's adjusted amount and the detached amount borrowed are now registered as accruals.

TOB-ATL-14: Borrower can reduce lender accruals
Resolved in PR 512. The yield factor now uses RAY precision, which limits the loss due to truncation.

TOB-ATL-15: Borrower can start a lending cycle before deposits are made
Resolved in PR 509. The `borrow` function now validates that the `remainingAmount` must be larger than the `minDepositAmount`, preventing nonzero borrows from occurring.

TOB-ATL-16: Documentation and naming conventions can be improved
Resolved in PR 515. Variable names throughout the codebase have been updated to remove ambiguity and clearly illustrate their purpose.

TOB-ATL-17: Missing validation in detach
Resolved in PR 507. The `validateDetach` function now correctly checks that the position is at least partially borrowed.

TOB-ATL-18: Contract architecture is overcomplicated
Resolved in PR 541. The TickLogic library has been split up into various dedicated logic contracts for ticks, positions, and borrower/lender actions.

TOB-ATL-19: Governance is a single point of failure
Resolved in PR 546. The documentation has been updated to reflect the powers and privileges as well as the risks associated with governance.

TOB-ATL-20: Pool is put in NON_STANDARD state only after executeTimelock() is called
Resolved in PR 520. The pool is set to the `CLOSED` state during a non-standard repayment procedure to allow only withdrawals and repayments.

TOB-ATL-21: Detached positions cannot be exited during subsequent loans
Resolved in PR 511. The `registerExit` will now update state variables corresponding to an exit if a detached position is in the same epoch as the current loan. This then allows a detached position to be exited in a subsequent loan.

TOB-ATL-22: rolesManager can never be updated
Resolved in PR 519. The `if` statement is removed, so the call to `updateRolesManager` will now correctly update the `rolesManager`.

TOB-ATL-23: Risks with transaction reordering
Unresolved. Atlendis Labs provided the following context for this issue:

> *Note that issue 23 is not addressed in the documentation as of today, as the case described in the issue is trivial, does not bring harm to the user, and is basically how the protocol should behave anyways.*

TOB-ATL-24: Problematic approach to handling precision errors
Resolved in PR 529. The rounding directions are now specified for every operation, and arbitrary checks for precision loss throughout the codebase have been removed. We recommend continuing to test these rounding directions through fuzzing campaigns.

`

TOB-ATL-25: Lenders with larger deposits earn less accruals if their position is only partially borrowed
Resolved in PR 512. The yield factor now uses RAY precision, which limits the precision loss that could occur due to a division by a larger deposit.