Security Review Report
NM-0121 ATLENDIS

NETHERMIND
SECURITY

(Oct 13, 2023)

Contents

1 Executive Summary

This document presents the security review performed by Nethermind on the Atlendis V2 Contracts. Atlendis is a capital-efficient credit protocol where borrowers have dedicated liquidity pools to which lenders can provide assets at different interest rates. Each liquidity pool consists of a number of ticks, each representing a different interest rate for the lender. When lenders deposit, they create a position represented by an ERC721 token assigned to a tick of their choosing. During a borrow, the liquidity belonging to the lowest interest rate ticks is used first until the borrowed amount is fulfilled.

This assessment focuses on two newly added features on top of the core contracts: the rollover feature, the early repayment feature, and two sets of new contracts, including the rewards component and a migration component.

The audited code consists of approximately 1,400 lines of Solidity with well-written NatSpec documentation for each function. The Atlendis team provided extensive documentation to assist during the audit, and they also made significant efforts in testing by adopting various testing methods. **The audit was performed using** (a) manual analysis of the codebase, (b) automated analysis tools, (c) simulation of the smart contracts, and (d) creation of test cases. **Along this document, we report** 11 points of attention, two are classified as `Medium`, three are classified as `Low`, two are classified as `Informational` and four are classified as `Best Practices`. The issues are summarized in Fig. 1.

This document is organized as follows. Section 2 presents the files in the scope of this audit. Section 3 summarizes the issues. Section 4 presents the system overview. Section 5 discusses the risk rating methodology adopted for this audit. Section 6 details the issues. Section 7 discusses the documentation provided by the client for this audit. Section 8 presents the compilation, tests, and automated tests. Section 9 concludes the document.

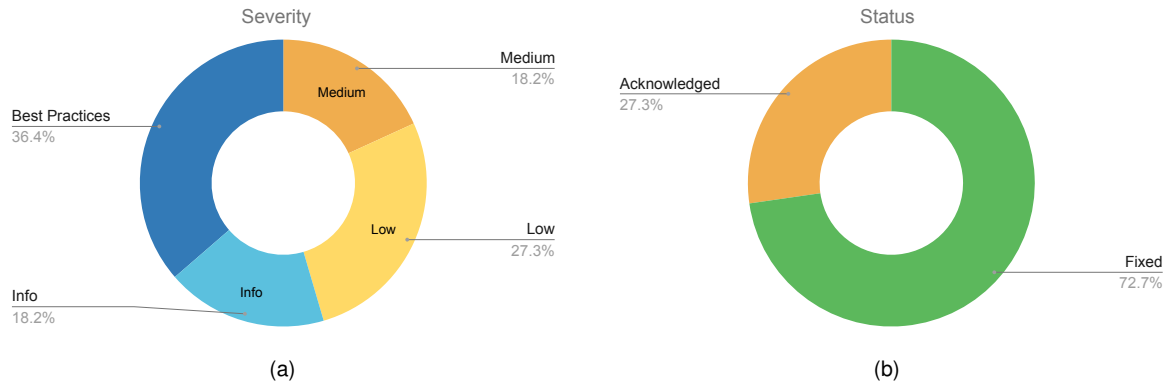

Fig. 1: Distribution of issues: Critical (0), **High** (0), **Medium** (2), **Low** (3), **Undetermined** (0), **Informational** (2), **Best Practices** (4). **Distribution of status: Fixed** (8), **Acknowledged** (3), **Mitigated** (0), **Unresolved** (0), **Partially Fixed** (0)

Summary of the Audit

Audit Type	Security Review
Initial Report	Oct. 2, 2023
Response from Client	Oct. 10, 2023
Final Report	Oct. 13, 2023
Methods	Manual Review, Automated Analysis
Repository	Atlendis V2
Commit Hash (Audit)	95b0c2fa73847581475d93e48763e0036a4ad39a
Documentation Assessment	High
Test Suite Assessment	High

2 Audited Files

	Contract	LoC	Comments	Ratio	Blank	Total
1	periphery/migration/V2Migration.sol	28	18	64.3%	8	54
2	periphery/migration/V1Migration.sol	43	17	39.5%	8	68
3	periphery/migration/interfaces/IPoolsController.sol	19	21	110.5%	1	41
4	periphery/migration/interfaces/IMigration.sol	8	10	125.0%	2	20
5	periphery/migration/interfaces/IPositionManager.sol	30	42	140.0%	5	77
6	rewards/RewardsManager.sol	244	94	38.5%	72	410
7	rewards/PositionStatus.sol	6	7	116.7%	1	14
8	rewards/IRewardsManager.sol	48	197	410.4%	32	277
9	rewards/modules/CustodianRewards.sol	106	73	68.9%	39	218
10	rewards/modules/ContinuousERC20Rewards.sol	131	67	51.1%	47	245
11	rewards/modules/TermERC20Rewards.sol	181	85	47.0%	54	320
12	rewards/modules/RewardsModule.sol	40	42	105.0%	17	99
13	rewards/modules/RateWeightedTermERC20Rewards.sol	40	38	95.0%	10	88
14	rewards/modules/interfaces/ITermERC20Rewards.sol	31	100	322.6%	14	145
15	rewards/modules/interfaces/ITermRewardsModule.sol	19	37	194.7%	4	60
16	rewards/modules/interfaces/IRateWeightedTermERC20Rewards.sol	6	19	316.7%	3	28
17	rewards/modules/interfaces/IContinuousERC20Rewards.sol	13	68	523.1%	10	91
18	rewards/modules/interfaces/IContinuousRewardsModule.sol	10	20	200.0%	2	32
19	rewards/modules/interfaces/ICustodianRewards.sol	19	45	236.8%	5	69
20	rewards/modules/interfaces/IRewardsModule.sol	14	69	492.9%	11	94
	Total	**1036**	**1069**	**103.2%**	**345**	**2450**

The audit scope also includes the following two pull requests: one for the rollover function and another for the early repayment function.

3 Summary of Issues

	Finding	Severity	Update
1	Early repayment amounts do not follow a fixed interest rate	Medium	Fixed
2	The function `updatePositionRate` can be used to obtain a larger reward for rate weighted ERC20 rewards	Medium	Fixed
3	Function `collectRewards` can lock a small amount of tokens in the `ContinuousERC20Rewards` module	Low	Acknowledged
4	Rounding direction used to calculate `earlyRepaymentAccrualReduction` can lead to insufficient repayments	Low	Fixed
5	Stakers cannot update stake with `ContinuousERC20Rewards` or `CustodianRewards` modules	Low	Acknowledged
6	Inconsistent check when reward is transferred out of different modules	Info	Fixed
7	Unfair reward distribution in the `CustodianRewards` module	Info	Acknowledged
8	Checks in `safeSubtract` functions are not consistent	Best Practices	Fixed
9	Incorrect NatSpec	Best Practices	Fixed
10	Missing reentrancy guards	Best Practices	Fixed
11	Optimizing repeated calculation to save gas	Best Practices	Fixed

4 System Overview

This audit focused on four new features: Loan rollovers, early loan repayments, migration contracts, and reward contracts.

The **rollover function** enables borrowers to extend their loans by starting a new loan cycle during the repayment of the previous one in the same transaction. Borrowers have choices such as rolling over the same amount with added interest and fees, rolling over with the full loan amount plus costs, borrowing more, or borrowing less. This feature's availability is subject to governance control, allowing oversight to prevent indefinite rollovers and ensure responsible lending practices.

The **early repayment function** allows borrowers to repay their loans before the scheduled repayment period begins, typically aimed at reducing the total interest paid by borrowers. When borrowers opt for early repayment, they settle their outstanding loan amount ahead of time, potentially saving on interest costs. The activation of the early repayment feature is subject to governance control. Early repayment is a beneficial tool for borrowers seeking to minimize the overall cost of their loans by settling them ahead of schedule.

The **migration contracts** ensure a seamless transition from the two existing pools of the protocol: v1 and v2, to a new pool. These contracts serve as a user-friendly utility, enabling users to effortlessly migrate their positions from the old pools to the updated one by invoking the `migrate(...)` function. This set consists of two contracts: `V1Migration` and `V2Migration`, each responsible for migrating user positions from v1 and v2 pools, respectively.

The **rewards contracts** allow users to stake their positions and earn various rewards. Users will interact with the reward manager to stake and unstake their positions, or claim their rewards. The reward manager will then call different modules to calculate and distribute the rewards. The reward modules are divided into two sets:

Continuous Rewards: Rewards are collected and distributed whenever users complete staking related actions.

- **Custodian Rewards**: Rewards that are collected from idle funds which have been placed in third party yield generator.
- **ERC20 Token Rewards**: Tokens are transferred to the module, and these tokens are distributed back to staked positions. The module distributes a fixed amount of tokens per second, and the rewards received by each staked position depend on the total amount of positions staked.

Term Rewards: Rewards are allocated at the time of staking for a chosen duration. After which, the rewards can be redeemed. However, if a user unstakes their position before the duration is complete they forfeit their rewards.

- **ERC20 Token**: Tokens are transferred to the module when a user stakes their position. The reward is computed based on factors such as the staked position amount, the chosen locking duration, and internal parameters of the module. These rewards become redeemable after the locking duration expires.
- **ERC20 Token with Rate Weight**: This is an extension of the above module. In this case, rewards computation also takes into account the rate of the position, in addition to the factors mentioned above. This rate-weighted approach provides a more nuanced way of calculating rewards based on the position's rate.

Fig. 2 presents the interaction diagram of rewards contracts.

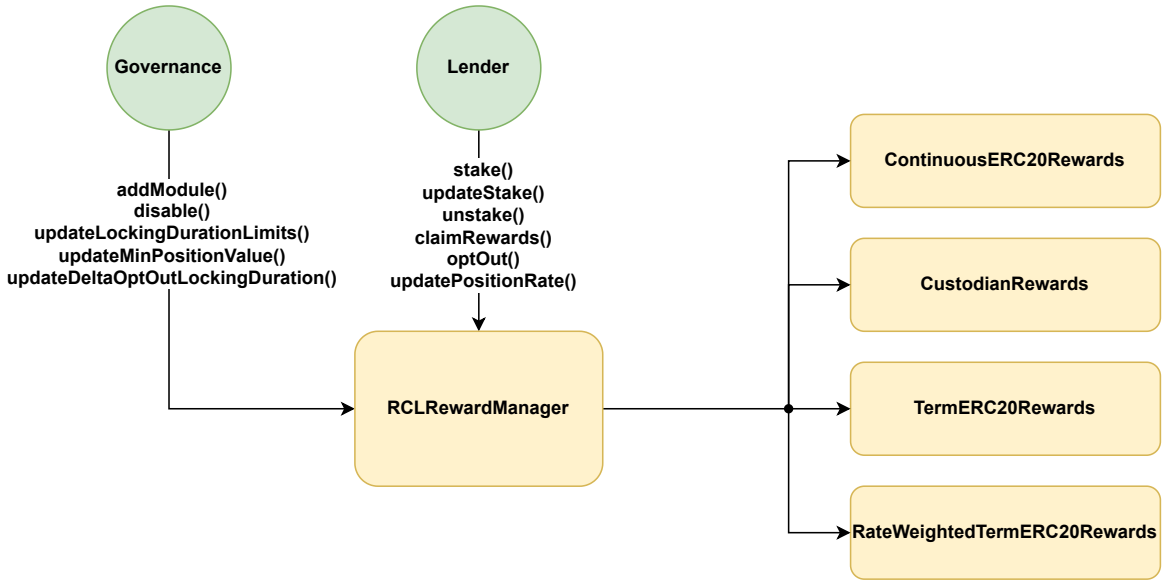

Fig. 2: Interaction Diagram of Rewards Contract

5 Risk Rating Methodology

The risk rating methodology used by Nethermind follows the principles established by the OWASP Foundation. The severity of each finding is determined by two factors: **Likelihood** and **Impact**.

Likelihood is a measure of how likely the finding is to be uncovered and exploited by an attacker. This factor will be one of the following values:

a) **High**: The issue is trivial to exploit and has no specific conditions that need to be met;

b) **Medium**: The issue is moderately complex and may have some conditions that need to be met;

c) **Low**: The issue is very complex and requires very specific conditions to be met.

When defining the likelihood of a finding other factors are also considered. These can include but are not limited to: Motive, opportunity, exploit accessibility, ease of discovery, and ease of exploit.

Impact is a measure of the damage that may be caused if the finding were to be exploited by an attacker. This factor will be one of the following values:

a) **High**: The issue can cause significant damage such as loss of funds or the protocol entering an unrecoverable state;

b) **Medium**: The issue can cause moderate damage such as impacts that only affect a small group of users or only a particular part of the protocol;

c) **Low**: The issue can cause little to no damage such as bugs that are easily recoverable or cause unexpected interactions that cause minor inconveniences.

When defining the impact of a finding other factors are also considered. These can include but are not limited to Data/state integrity, loss of availability, financial loss, and reputation damage. After defining the likelihood and impact of an issue, the severity can be determined according to the table below.

		Severity Risk		
Impact	**High**	Medium	High	Critical
	Medium	Low	Medium	High
	Low	Info/Best Practices	Low	Medium
	Undetermined	Undetermined	Undetermined	Undetermined
		Low	**Medium**	**High**
		Likelihood		

To address issues that do not fit a High/Medium/Low severity, Nethermind also uses three more finding severities: **Informational**, **Best Practices**, and **Undetermined**.

a) **Informational** findings do not pose any risk to the application, but they carry some information that the audit team intends to formally pass to the client;

b) **Best Practice** findings are used when some piece of code does not conform with smart contract development best practices;

c) **Undetermined** findings are used when we cannot predict the impact or likelihood of the issue.

6 Issues

6.1 [Medium] Early repayment amounts do not follow a fixed interest rate

File(s): `RCLBorrowers.sol`

Description: When a lender exits a position, liquidity from a position at a different tick may take its place, which will change the calculated interest rate. Atlendis has designed its repayment system such that the interest amount is calculated at the time of the borrow so that borrowers will know the exact amount that will be owed at the end of their loan.

With the introduced logic surrounding early repayments, a fixed interest rate throughout the loan can no longer be guaranteed. The interest amount to be paid for an early repayment is calculated on the current distribution of positions on ticks rather than relying on the static interest rate that was calculated at the start of the loan.

As a lender, it is possible to manipulate the interest rate of an early repayment with the following steps:

1. Lender has borrowed position within the higher ticks for a given loan;
2. Borrower sends a transaction to the mempool to complete an early repayment;
3. Lender frontruns this transaction and exits their position, unborrowed funds from lower ticks take its place;
4. Borrower early repayment executes, and the `earlyRepaymentReduction` is calculated based on the current lower interest rate, leading to a smaller reduction than expected;

Recommendation(s): Consider allowing only one of these features (early repayments or exiting positions during a loan) at a time.

Status: Fixed

Update from the client: We are fully aware of this issue, and ran simulations to estimate the range of the rate difference depending on the time and amount of exits. Our conclusion after a deeper analysis is that the changes required to mitigate the issue might be sizeable, and would necessitate a deeper rework of our data structures to be solved. For that reason, we will clearly state with borrowers that to get fixed rate rates as expected, the pool can either offer early repayment or secondary market capabilities, but not both at the time. We implemented a toggle function that can deactivate exits if need be. PR: https://github.com/Atlendis/priv-contracts-v2/pull/714

6.2 [Medium] The function `updatePositionRate` can be used to obtain a larger reward for rate weighted ERC20 rewards

File(s): `RewardsManager.sol`

Description: The contract `RewardsManager` allows lenders to update the rate of their staking position by calling the function `updatePositionRate(...)`. This function only updates the rate through a call to `POSITION_MANAGER.updateRate(...)`, without updating the reward in any module.

This poses an issue because the `RateWeightedTermERC20Rewards` module relies on the position's current tick rate to calculate the reward at staking time. Consequently, an attacker could create a position with a low rate, stake it in the `RateWeightedTermERC20Rewards` module to obtain a larger reward, and then call `updatePositionRate(...)` to change the rate of their position. This allows the attacker to continue receiving a large reward from the `RateWeightedTermERC20Rewards` module and also benefit from a high-interest rate from borrowers.

Recommendation(s): Consider checking or updating the reward when updating the rate of a position.

Status: Fixed

Update from the client: The issue was acknowledged and fixed, rate weighted term erc20 rewards module now does not allow for position update rate. PR: https://github.com/Atlendis/priv-contracts-v2/pull/730

6.3 [Low] Function `collectRewards` can lock a small amount of tokens in the `ContinuousERC20Rewards` module

File(s): `ContinuousERC20Rewards.sol`

Description: In the `collectRewards(...)` function, `rewardsSinceLastUpdate` is calculated and then divided by total `deposits` to calculate `earningsPerDeposit`. However, although `earningsPerDeposit` is rounded down, `rewardsSinceLastUpdate` is not updated accordingly, leading to a small amount of rewards being locked in the contract.

Consider the scenario where `rewardsSinceLastUpdate = 101` and `deposits = 100`. For simplicity, we remove the `RAY` calculation in the example. In this case, because of rounding down, `earningsPerDeposit` should increase by `1`, making the total reward needed `100`. However, `pendingRewards` still adds `101`, reserving `1` extra token wei. These rewards will be locked in the contract forever.

```
uint256 contractBalance = TOKEN.balanceOf(address(this));
uint256 rewardsSinceLastUpdate = pendingRewards + maximumRewardsSinceLastUpdate <= contractBalance
    ? maximumRewardsSinceLastUpdate
    : contractBalance - pendingRewards;

// @audit `earningsPerDeposit` is rounded down but `rewardsSinceLastUpdate` is not updated accordingly
earningsPerDeposit += rewardsSinceLastUpdate.div(deposits, RAY);
pendingRewards += rewardsSinceLastUpdate;
lastUpdateTimestamp = block.timestamp;
```

Recommendation(s): Consider recalculating the amount of rewards that should be added to `pendingRewards` from the `earningsPerDeposit` increase.

Status: Acknowledged

Update from the client: The changes required to address this finding properly would imply inefficient arithmetic operations that would add complexity and gas consumption to our contracts. We feel that these changes are not worth the very minimal amounts of tokens that could be locked because of this issue - the total amount of locked tokens would be in the tens of weis at most. For these reasons, we will not address this finding.

6.4 [Low] Rounding direction used to calculate `earlyRepaymentAccrualReduction` can lead to insufficient repayments

File(s): BorrowerLogic.sol

Description: When borrowers repay a loan early, the function `registerEarlyRepaymentAccrualReduction(...)` calculates the `earlyRepaymentAccrualReduction` based on the duration and the rate. This amount is deducted when calculating the final amount that the borrower needs to repay.

The value `earlyRepaymentAccrualReduction` is rounded up, meaning that when it is subtracted from the total amount, the final repayment amount might be less than expected.

```
function registerEarlyRepaymentAccrualReduction(
    DataTypes.Tick storage tick,
    uint256 timeDelta,
    uint256 rate
) public returns (uint256 earlyRepaymentAccrualReduction) {
    ////////////////////////////////////////////////////////////////
    // @audit This should round down instead of up
    //        All `accrualsForUp` should be `accrualsForDown` instead
    ////////////////////////////////////////////////////////////////
    if (tick.baseEpochsAmounts.borrowed != 0) {
        earlyRepaymentAccrualReduction += AccrualsHelpers.accrualsForUp(
            tick.baseEpochsAmounts.borrowed,
            timeDelta,
            rate
        );
        tick.yieldFactor -= AccrualsHelpers.calculateYieldFactorDecrease(
            tick,
            AccrualsHelpers.accrualsForUp(tick.baseEpochsAmounts.borrowed, timeDelta, rate)
        );
    }
    if (tick.newEpochsAmounts.borrowed != 0) {
        uint256 accrualsReduction = AccrualsHelpers.accrualsForUp(tick.newEpochsAmounts.borrowed, timeDelta, rate);
        // ...
    }
    if (tick.detachedAmounts.borrowed != 0) {
        uint256 accrualsReduction = AccrualsHelpers.accrualsForUp(tick.detachedAmounts.borrowed, timeDelta, rate);
        // ...
    }
}
```

Recommendation(s): Consider rounding down instead of rounding up when calculating `earlyRepaymentAccrualReduction` in the function `registerEarlyRepaymentAccrualReduction(...)`.

Status: Fixed

Update from the client: The issue is acknowledged and was fixed, resolving flaky unit tests at the same time. PR: https://github.com/Atlendis/priv-contracts-v2/pull/715

6.5 [Low] Stakers cannot update stake with `ContinuousERC20Rewards` or `CustodianRewards` modules

File(s): `ContinuousERC20Rewards.sol`, `CustodianRewards.sol`

Description: In the reward modules contract, when users want to update their stake, the function `stake(...)` is called to override the position data. However, in `ContinuousERC20Rewards` and `CustodianRewards`, if a position is already staked, the function simply returns without updating any value of the position.

```
function stake(
    uint256 positionId,
    address owner,
    uint256 rate,
    uint256 positionValue
) public onlyRewardsManager rewardsCollector {
    StakedPosition storage stakedPosition = stakedPositions[positionId];
    if (isStaked(positionId)) return; // @audit Cannot `_updateStake()`
    ...
}
```

Recommendation(s): Consider modifying the logic to update the position even if it is already staked.

Status: Acknowledged

Update from the client: This is intentional, the benefits of the feature - restaking rewards - are not compelling enough to justify the added complexity, given the expected amount of expected rewards. For these reasons, we will not address this finding.

6.6 [Info] Inconsistent check when reward is transferred out of different modules

File(s): `ContinuousERC20Rewards.sol`

Description:

In some reward modules, it is possible for a reward amount of zero to be transferred out when unstaking, while for some other modules, it is not. This inconsistency may cause a problem if the ERC20 reward token reverts on zero-value transfers. For example, if a new module that has this weird reward token is added while the position is already staked, when the position is unstaked, the reward for the new module will be zero and may cause a revert.

```
// @audit `ContinuousERC20Rewards.sol`
function unstake(uint256 positionId, address owner) public onlyRewardsManager rewardsCollector {
    ...
    // @audit Could revert if `TOKEN` reverts on zero value transfers
    TOKEN.safeTransfer(owner, positionRewards);
    ...
}

// @audit `TermERC20Rewards.sol`
function unstake(uint256 positionId, address owner) public onlyRewardsManager {
    ...
    if (positionRewards > 0) {
        TOKEN.safeTransfer(owner, positionRewards);
    }
    ...
}
```

Recommendation(s): Consider using a consistent check when transferring rewards to avoid potential issues with ERC20 reward tokens by adding a zero amount check to `unstake(...)` in `ContinuousERC20Rewards.sol`.

Status: Fixed

Update from the client: Issue was acknowledged and fixed. PR : https://github.com/Atlendis/priv-contracts-v2/pull/731

6.7 [Info] Unfair reward distribution in the `CustodianRewards` module

File(s): `CustodianRewards.sol`

Description: In an Atlendis pool, when users' funds are deposited but not borrowed, they can be deposited into an external protocol to earn additional yield. For example, during the time borrowers repay their loans, all the funds can be deposited into an Aave pool. These accrued yields are then distributed as rewards in the `CustodianRewards` module.

Even though the funds of all users are used to earn yield, only stakers will receive these rewards, which is unfair. Since users are the ones taking the risk when depositing their funds into an external protocol, if they choose not to stake, their funds should not be used.

Recommendation(s): Consider reviewing the mechanism of the `CustodianRewards` module to ensure fairness between staker and non-staker users.

Status: Acknowledged

Update from the client: This mechanism was implemented this way for simplicity and gas consumption reasons. The rewards distribution is indeed not representative of what each individual lender should get, but they will always have the possibility to stake their position to get their share when such a rewards program is deployed. For these reasons, we won't make any changes following this finding as the mechanism is implemented as expected.

6.8 [Best Practices] Checks in `safeSubtract` functions are not consistent

File(s): `LenderLogic.sol`

Description: The `LenderLogic` contract has multiple `safeSubstract` functions that are used to update detached, epoch, base epoch, and new epoch amounts in a safe way. To avoid underflow reverts, a check verifies that the retrieved value is lower than the value to be retrieved from. If the value to be retrieved is greater, a zero value is set. The following code is an example of such a check:

```
1 tick.newEpochsAmounts.borrowed = borrowedAmountToWithdraw > tick.newEpochsAmounts.borrowed
2     ? 0
3     : tick.newEpochsAmounts.borrowed - borrowedAmountToWithdraw;
```

However, this safe pattern is not applied to all `safeSubstract` functions. This check is not applied for instance for `tick.newEpochsAmounts.toBeAdjusted` and `tick.newEpochsAmounts.optedOut` in the function `safeSubstractNewEpochsAmounts(...)`.

Recommendation(s): Consider adding checks in `safeSubstract` functions to avoid underflow reverts.

Status: Fixed

Update from the client: The issue was acknowledged and fixed, all safe subtract functions were aligned on the same model, implementing the underflow checks. PR: https://github.com/Atlendis/priv-contracts-v2/pull/729

6.9 [Best Practices] Incorrect NatSpec

File(s): LenderLogic.sol, AccrualHelpers.sol, BorrowerLogic.sol

Description: NatSpec blocks describe the specifications of the functions. The following incorrectness has been identified:

- `positionOwner` from `validateWithdraw()` function;

```
/**
 * ...
 * @param positionOwner Minimum amount of funds that must be held in a position
 * ...
 */
function validateWithdraw(
    DataTypes.Position storage position,
    DataTypes.Tick storage tick,
    DataTypes.Loan storage referenceLoan,
    DataTypes.Loan storage currentLoan,
    uint256 amountToWithdraw,
    uint256 minDepositAmount,
    address positionOwner    //@audit positionOwner is the address of the owner
) {...}
```

- `dev` information from the `accrualsForDown()` function;

```
/**
 * ...
 * @dev Rounds the result up
 * ...
 */
function accrualsForDown(
    uint256 amount,
    uint256 timeDelta,
    uint256 rate
) public view returns (uint256 accruals) {
    // @audit mulDown rounds down the accruals, the result is not rounded up
    accruals = ((amount * timeDelta * rate) / 365 days).mulDown(1);
}
```

- `notice` section for the `registerDetachedInterestFees()` function;

```
/**
 * @notice Logic to persist the repayment fees for new epochs
 * ...
 */
function registerDetachedInterestFees(
    DataTypes.Tick storage tick,
    IFeesController feesController
) public returns (
    uint256 fees
) {...}
```

Recommendation(s): Modify the NatSpec specifications to match the code's utility.

Status: Fixed

Update from the client: The parameter in the first natspec was removed after implementing the force withdraw position for revoked lender feature. The other natspecs were fixed. PR1: https://github.com/Atlendis/priv-contracts-v2/pull/719 PR2: https://github.com/Atlendis/priv-contracts-v2/pull/733

6.10 [Best Practices] Missing reentrancy guards

File(s): RCLLenders.sol

Description: The Atlendis protocol does not protect itself from reentrancy. Since there are no expected user behaviors where reentrancy is necessary, adding reentrancy guards can help to prevent unexpected behavior.

As an example, when a lender deposits into the protocol an NFT will be minted representing their position ID. A malicious lender associated with a contract address is able to execute their own logic when the NFT is minted through the `_checkOnERC721Received()` callback. This NFT is minted before sending the funds to the custodian contract. A malicious `lender` is able to call other entry points from the protocol with the NFT before sending the funds to the protocol.

```
function deposit(
    uint256 rate,
    uint256 amount,
    address to
) external onlyLender onlyInPhase(DataTypes.PoolPhase.OPEN) returns (uint256 positionId) {
    // ...

    ////////////////////////////////////////////////////////////////////
    //@audit lender could get control flow back before sending the funds
    ////////////////////////////////////////////////////////////////////
    _safeMint(to, positionId);
    CUSTODIAN.deposit(amount, msg.sender);

    // ...
}
```

We would like to note that with the code at the time of this security review, we were not able to identify any direct security issues related to reentrancy, however since reentrancy is an unexpected user interaction we recommend preventing it entirely.

Recommendation(s): Consider implementing reentrancy protection on all entry points within the protocol.

Status: Fixed

Update from the client: The issue was acknowledged and fixed, reentrancy locks were introduced in functions that can trigger position transfers, such as deposit, withdraw and exit. PR: https://github.com/Atlendis/priv-contracts-v2/pull/728

6.11 [Best Practices] Optimizing repeated calculation to save gas

File(s): BorrowerLogic.sol

Description: In the function `registerEarlyRepaymentAccrualReduction(...)`, the helper function `accrualsForUp(...)` is called twice with the exact same parameters. This redundancy can be optimized by saving the result of the first call to a variable and using that variable instead of recalculating the value.

```
////////////////////////////////////////
// @audit Repeated `accrualsForUp()` calls
////////////////////////////////////////
earlyRepaymentAccrualReduction += AccrualsHelpers.accrualsForUp(
    tick.baseEpochsAmounts.borrowed,
    timeDelta,
    rate
);
tick.yieldFactor -= AccrualsHelpers.calculateYieldFactorDecrease(
    tick,
    AccrualsHelpers.accrualsForUp(tick.baseEpochsAmounts.borrowed, timeDelta, rate)
);
```

Recommendation(s): Consider saving the result of the first call to `accrualsForUp(...)` in a variable and using that variable instead of recalculating it to save gas.

Status: Fixed

Update from the client: These calculations were changed after solving the preceding finding (rounding down), and they are not repeated anymore. As such, this issue is not valid anymore.

7 Documentation Evaluation

Software documentation refers to the written or visual information that describes the functionality, architecture, design, and implementation of software. It provides a comprehensive overview of the software system and helps users, developers, and stakeholders understand how the software works, how to use it, and how to maintain it. Software documentation can take different forms, such as user manuals, system manuals, technical specifications, requirements documents, design documents, and code comments. Software documentation is critical in software development, enabling effective communication between developers, testers, users, and other stakeholders. It helps to ensure that everyone involved in the development process has a shared understanding of the software system and its functionality. Moreover, software documentation can improve software maintenance by providing a clear and complete understanding of the software system, making it easier for developers to maintain, modify, and update the software over time. Smart contracts can use various types of software documentation. Some of the most common types include:

- Technical whitepaper: A technical whitepaper is a comprehensive document describing the smart contract's design and technical details. It includes information about the purpose of the contract, its architecture, its components, and how they interact with each other;
- User manual: A user manual is a document that provides information about how to use the smart contract. It includes step-by-step instructions on how to perform various tasks and explains the different features and functionalities of the contract;
- Code documentation: Code documentation is a document that provides details about the code of the smart contract. It includes information about the functions, variables, and classes used in the code, as well as explanations of how they work;
- API documentation: API documentation is a document that provides information about the API (Application Programming Interface) of the smart contract. It includes details about the methods, parameters, and responses that can be used to interact with the contract;
- Testing documentation: Testing documentation is a document that provides information about how the smart contract was tested. It includes details about the test cases that were used, the results of the tests, and any issues that were identified during testing;
- Audit documentation: Audit documentation includes reports, notes, and other materials related to the security audit of the smart contract. This type of documentation is critical in ensuring that the smart contract is secure and free from vulnerabilities.

These types of documentation are essential for smart contract development and maintenance. They help ensure that the contract is properly designed, implemented, and tested, and they provide a reference for developers who need to modify or maintain the contract in the future.

The `Atlendis` team has shared a thorough whitepaper about their protocol, which covers the main aspects of the protocol and provides detailed information about user actions and associated fees. Specifically for this audit, the four newly added features are documented in the README file. Moreover, the team conducted a comprehensive code walkthrough and maintained open communication to address any inquiries or concerns raised by the Nethermind auditors.

8 Test Suite Evaluation

8.1 Contracts Compilation Output

```
> forge build
[] Compiling...
[] Compiling 306 files with 0.8.13
[] Solc 0.8.13 finished in 127.14s
Compiler run successful
```

8.2 Tests Output

The relevant output is presented below.

```
Running 1 test for test/unit/rewards/ContinuousERC20Rewards.t.sol:ContinuousERC20RewardsDeploymentTest
Test result: ok. 1 passed; 0 failed; finished in 1.13ms

Running 3 tests for test/unit/rewards/TermERC20Rewards.t.sol:TermERC20RewardsDeploymentTest
Test result: ok. 3 passed; 0 failed; finished in 2.84ms

Running 3 tests for test/unit/rewards/RewardsManager.t.sol:RewardsManagerDeploymentTest
Test result: ok. 3 passed; 0 failed; finished in 1.21ms

Running 18 tests for test/unit/rewards/RewardsManagerGovernance.t.sol:RewardsManagerGovernanceTest
Test result: ok. 18 passed; 0 failed; finished in 6.86ms

Running 4 tests for test/unit/rewards/RateWeightedTermERC20Rewards.t.sol:RateWeightedTermERC20RewardsTest
Test result: ok. 4 passed; 0 failed; finished in 206.59ms

Running 8 tests for test/unit/rewards/BulletRewardsManager.t.sol:BulletRewardsManagerTest
Test result: ok. 8 passed; 0 failed; finished in 582.01ms

Running 23 tests for test/unit/rewards/TermERC20Rewards.t.sol:TermERC20RewardsTest
Test result: ok. 23 passed; 0 failed; finished in 703.37ms

Running 19 tests for test/unit/rewards/RewardsManager.t.sol:RewardsManagerTest
Test result: ok. 19 passed; 0 failed; finished in 760.32ms

Running 16 tests for test/unit/rewards/ContinuousERC20Rewards.t.sol:ContinuousERC20RewardsTest
Test result: ok. 16 passed; 0 failed; finished in 759.77ms

Running 8 tests for test/unit/rewards/CustodianRewards.t.sol:CustodianRewardsTest
Test result: ok. 8 passed; 0 failed; finished in 584.44ms

Running 6 tests for test/unit/rewards/RCLRewardsManagerPositionUpdate.t.sol:RCLRewardsManagerTest
Test result: ok. 6 passed; 0 failed; finished in 762.01ms

Running 5 tests for test/unit/periphery/migration/V1Migration.t.sol:V1MigrationTestContract
Test result: ok. 5 passed; 0 failed; finished in 130.49ms

Running 3 tests for test/unit/periphery/migration/V2Migration.t.sol:V2MigrationTestContract
Test result: ok. 3 passed; 0 failed; finished in 189.37ms

Running 4 tests for test/unit/products/revolving-credit-line/RCLLenders.transferFrom.t.sol:RCLLenderTranserFromTest
Test result: ok. 4 passed; 0 failed; finished in 8.86ms

Running 29 tests for test/unit/products/revolving-credit-line/RCLGovernance.t.sol:RCLGovernanceTest
Test result: ok. 29 passed; 0 failed; finished in 144.98ms

Running 16 tests for test/unit/products/revolving-credit-line/RCLLenders.deposit.t.sol:RCLLenderDepositTest
Test result: ok. 16 passed; 0 failed; finished in 1.05s

Running 16 tests for test/unit/products/revolving-credit-line/RCLLenders.updateRate.t.sol:RCLLenderUpdateRateTest
Test result: ok. 16 passed; 0 failed; finished in 2.20s
```

```
Running 6 tests for test/unit/products/revolving-credit-line/RCLBorrowers.rollover.t.sol:RCLBorrowerRolloverTest
Test result: ok. 6 passed; 0 failed; finished in 734.09ms

Running 12 tests for test/unit/products/revolving-credit-line/RCLFactory.t.sol:RCLFactoryTest
Test result: ok. 12 passed; 0 failed; finished in 10.40ms

Running 24 tests for test/unit/products/revolving-credit-line/RCLLenders.opt-out.t.sol:RCLLenderOptOutTest
Test result: ok. 24 passed; 0 failed; finished in 3.55s

Running 26 tests for test/unit/products/revolving-credit-line/RCLBorrowers.borrow.t.sol:RCLBorrowerBorrowTest
Test result: ok. 26 passed; 0 failed; finished in 7.87s

Running 42 tests for test/unit/products/revolving-credit-line/RCLLenders.withdraw.t.sol:RCLLenderWithdrawTest
Test result: ok. 19 passed; 0 failed; finished in 9.72s

Running 32 tests for test/unit/products/revolving-credit-line/RCLLenders.exit.t.sol:RCLLendersExitTest
Test result: ok. 32 passed; 0 failed; finished in 7.04s

Running 37 tests for test/unit/products/revolving-credit-line/RCLLenders.detach.t.sol:RCLLenderDetachTest
Test result: ok. 37 passed; 0 failed; finished in 8.51s

Running 44 tests for test/unit/products/revolving-credit-line/RCLBorrowers.repay.t.sol:RCLBorrowerRepayTest
Test result: ok. 44 passed; 0 failed; finished in 9.71s

Running 1 test for test/functional/periphery/migration/V1MigrationFlunaPool.t.sol:V1MigrationFlunaPool
Test result: ok. 1 passed; 0 failed; finished in 6.44s

Running 1 test for
↪  test/functional/periphery/migration/V2MigrationAfterBorrowWithFees.t.sol:V2MigrationAfterBorrowWithFees
Test result: ok. 1 passed; 0 failed; finished in 14.51s

Running 1 test for test/functional/products/revolving-credit-line/GasMeasurements-MultipleTicksExit.t.sol:
    GasMeasurementMultipleTicksExit
Test result: ok. 1 passed; 0 failed; finished in 10.89s

Running 1 test for test/functional/products/revolving-credit-line/GasMeasurements-SingleTickDetachOptOut.t.sol:
    GasMeasurementSingleTickDetachOptOut
Test result: ok. 1 passed; 0 failed; finished in 10.89s

Running 1 test for test/functional/products/revolving-credit-line/GasMeasurements-MultipleTicksBorrow.t.sol:
    GasMeasurementMultipleTicksBorrow
Test result: ok. 1 passed; 0 failed; finished in 24.64ms

Running 1 test for test/functional/products/revolving-credit-line/RolloverFullLoan.t.sol:RolloverFullLoan
Test result: ok. 1 passed; 0 failed; finished in 11.26s

Running 1 test for
↪  test/functional/products/revolving-credit-line/AdditionalBorrowerFundsRequiredWithVoluntaryRepay.t.sol:
    AdditionalBorrowerFundsRequiredWithVoluntaryRepay
Test result: ok. 1 passed; 0 failed; finished in 11.26s

Running 1 test for
↪  test/functional/products/revolving-credit-line/AdditionalBorrowerFundsRequiredForOptedOutPositions.t.sol:
    AdditionalBorrowerFundsRequiredForOptedOutPositions
Test result: ok. 1 passed; 0 failed; finished in 11.26s

Running 1 test for
↪  test/functional/products/revolving-credit-line/IntraTickExitsMultipleCycles.t.sol:IntraTickExitsMultipleCycles
Test result: ok. 1 passed; 0 failed; finished in 11.26s

Running 1 test for test/functional/products/revolving-credit-line/RolloverLoanWithAdditionalBorrow.t.sol:
    RolloverLoanWithAdditionalBorrow
Test result: ok. 1 passed; 0 failed; finished in 15.67ms

Running 1 test for test/functional/products/revolving-credit-line/NoBorrowerFundsRequiredWithVoluntaryRepay.t.sol:
    NoBorrowerFundsRequiredWithVoluntaryRepay
Test result: ok. 1 passed; 0 failed; finished in 16.40ms

Running 1 test for test/functional/products/revolving-credit-line/RolloverLoanWithFees.t.sol:RolloverLoanWithFees
Test result: ok. 1 passed; 0 failed; finished in 465.36ms
```

```
Running 1 test for test/functional/products/revolving-credit-line/RolloverLoanWithFeesBorrowerNoFundsTransferred.t.sol:
    RolloverLoanWithFeesBorrowerNoFundsTransferred
Test result: ok. 1 passed; 0 failed; finished in 7.24ms

Running 1 test for
↪  test/functional/products/revolving-credit-line/SecondaryBuyingBaseEpochExits.t.sol:SecondaryBuyingBaseEpochExits
Test result: ok. 1 passed; 0 failed; finished in 7.88ms

Running 1 test for test/functional/products/revolving-credit-line/TwoLoanCyclesWithExits.t.sol:TwoLoanCyclesWithExits
Test result: ok. 1 passed; 0 failed; finished in 568.87ms

Running 1 test for test/functional/products/revolving-credit-line/ThreeLoanCyclesWithSignalling.t.sol:
    ThreeLoanCyclesWithOptOutling
Test result: ok. 1 passed; 0 failed; finished in 10.39ms

Running 1 test for test/functional/products/revolving-credit-line/EarlyRepayWithExit.t.sol:EarlyRepayWithExit
Test result: ok. 1 passed; 0 failed; finished in 11.89s

Running 1 test for test/functional/products/revolving-credit-line/SwitchYieldProvider.t.sol:SwitchYieldProvider
Test result: ok. 1 passed; 0 failed; finished in 1.19s

Running 1 test for test/functional/products/revolving-credit-line/SimpleRescue.t.sol:SimpleCycleRescue
Test result: ok. 1 passed; 0 failed; finished in 1.02s

Running 1 test for test/functional/products/revolving-credit-line/ThreeLoanCyclesWithExitsOptOutsWithdrawals.t.sol:
    ThreeLoanCyclesWithExitsOptOutsWithdrawals
Test result: ok. 1 passed; 0 failed; finished in 592.01ms

Running 1 test for test/functional/products/revolving-credit-line/FullLoanCycleWithEarlyRepaysDetachAndFees.t.sol:
    FullLoanCycleWithEarlyRepaysDetachAndFees
Test result: ok. 1 passed; 0 failed; finished in 12.69s

Running 1 test for test/functional/products/revolving-credit-line/ThreeLoanCyclesWithFees.t.sol:ThreeLoanCyclesWithFees
Test result: ok. 1 passed; 0 failed; finished in 637.24ms

Running 1 test for test/functional/products/revolving-credit-line/SecondaryBuyingNewEpochExits.t.sol:
    SecondaryBuyingNewEpochExits
Test result: ok. 1 passed; 0 failed; finished in 1.31s

Running 1 test for test/functional/products/revolving-credit-line/ThreeLoanCyclesWithFurtherBorrows.t.sol:
    ThreeLoanCyclesWithFurtherBorrows
Test result: ok. 1 passed; 0 failed; finished in 2.03s

Running 1 test for test/functional/products/revolving-credit-line/ThreeLoanCyclesWithMultipleTicksAndEpochs.t.sol:
    ThreeLoanCyclesWithMultipleTicksAndEpochs
Test result: ok. 1 passed; 0 failed; finished in 3.40s

Running 1 test for test/functional/products/revolving-credit-line/StakingTermERC20AndRateWeightedTermERC20.t.sol:
    StakingTermERC20AndRateWeightedTermERC20
Test result: ok. 1 passed; 0 failed; finished in 5.55s

Running 1 test for test/functional/products/revolving-credit-line/Staking-ContinuousERC20&Custodian.t.sol:
    StakingContinuousERC20AndCustodian
Test result: ok. 1 passed; 0 failed; finished in 12.04s
```

8.3 Code Coverage

```
> forge coverage
```

The relevant output is presented below.

File	% Statements	% Branches	% Funcs
common/rewards/RewardsManager.sol	96.18% (126/131)	92.50% (37/40)	86.36% (19/22)
common/rewards/modules/ContinuousERC20Rewards.sol	100.00% (66/66)	100.00% (14/14)	77.78% (7/9)
common/rewards/modules/CustodianRewards.sol	100.00% (60/60)	78.57% (11/14)	71.43% (5/7)
common/rewards/modules/RateWeightedTermERC20Rewards.sol	100.00% (9/9)	100.00% (2/2)	100.00% (2/2)
common/rewards/modules/RewardsModule.sol	85.71% (6/7)	50.00% (1/2)	66.67% (2/3)
common/rewards/modules/TermERC20Rewards.sol	100.00% (77/77)	100.00% (24/24)	93.33% (14/15)
libraries/FixedPointMathLib.sol	61.11% (11/18)	100.00% (0/0)	50.00% (7/14)
libraries/FundsTransfer.sol	100.00% (11/11)	50.00% (2/4)	100.00% (5/5)
libraries/TimeValue.sol	0.00% (0/3)	100.00% (0/0)	0.00% (0/1)
periphery/migration/V1Migration.sol	100.00% (16/16)	100.00% (4/4)	100.00% (1/1)
periphery/migration/V2Migration.sol	100.00% (8/8)	100.00% (0/0)	100.00% (1/1)
products/RevolvingCreditLines/RCLFactory.sol	100.00% (29/29)	100.00% (6/6)	100.00% (4/4)
products/RevolvingCreditLines/RCLRewardsManager.sol	100.00% (49/49)	100.00% (10/10)	100.00% (7/7)
products/RevolvingCreditLines/RevolvingCreditLine.sol	100.00% (10/10)	100.00% (0/0)	100.00% (2/2)
products/RevolvingCreditLines/libraries/AccrualsHelpers.sol	75.00% (6/8)	50.00% (2/4)	100.00% (4/4)
products/RevolvingCreditLines/libraries/BorrowerLogic.sol	100.00% (162/162)	80.00% (40/50)	23.08% (3/13)
products/RevolvingCreditLines/libraries/LenderLogic.sol	99.10% (221/223)	95.92% (94/98)	72.00% (18/25)
products/RevolvingCreditLines/libraries/PositionHelpers.sol	98.46% (128/130)	89.29% (50/56)	81.82% (9/11)
products/RevolvingCreditLines/libraries/RCLValidation.sol	100.00% (12/12)	100.00% (10/10)	100.00% (1/1)
products/RevolvingCreditLines/libraries/RclTimelockLogic.sol	0.00% (0/19)	0.00% (0/12)	0.00% (0/3)
products/RevolvingCreditLines/modules/RCLBorrowers.sol	100.00% (72/72)	86.11% (31/36)	100.00% (10/10)
products/RevolvingCreditLines/modules/RCLGovernance.sol	100.00% (50/50)	92.86% (13/14)	100.00% (13/13)
products/RevolvingCreditLines/modules/RCLLenders.sol	99.07% (107/108)	96.67% (29/30)	100.00% (15/15)
products/RevolvingCreditLines/modules/RCLState.sol	92.86% (13/14)	100.00% (4/4)	80.00% (4/5)

9 About Nethermind

Nethermind is a Blockchain Research and Software Engineering company. Our work touches every part of the web3 ecosystem - from layer 1 and layer 2 engineering, cryptography research, and security to application-layer protocol development. We offer strategic support to our institutional and enterprise partners across the blockchain, digital assets, and DeFi sectors, guiding them through all stages of the research and development process, from initial concepts to successful implementation.

We offer security audits of projects built on EVM-compatible chains and Starknet. We are active builders of the Starknet ecosystem, delivering a node implementation, a block explorer, a Solidity-to-Cairo transpiler, and formal verification tooling. Nethermind also provides strategic support to our institutional and enterprise partners in blockchain, digital assets, and decentralized finance (DeFi). In the next paragraphs, we introduce the company in more detail.

Blockchain Security: At Nethermind, we believe security is vital to the health and longevity of the entire Web3 ecosystem. We provide security services related to Smart Contract Audits, Formal Verification, and Real-Time Monitoring. Our Security Team comprises blockchain security experts in each field, often collaborating to produce comprehensive and robust security solutions. The team has a strong academic background, can apply state-of-the-art techniques, and is experienced in analyzing cutting-edge Solidity and Cairo smart contracts, such as ArgentX and StarkGate (the bridge connecting Ethereum and StarkNet). Most team members hold a Ph.D. degree and actively participate in the research community, accounting for 240+ articles published and 1,450+ citations in Google Scholar. The security team adopts customer-oriented and interactive processes where clients are involved in all stages of the work.

Blockchain Core Development: Our core engineering team, consisting of over 20 developers, maintains, improves, and upgrades our flagship product - the Nethermind Ethereum Execution Client. The client has been successfully operating for several years, supporting both the Ethereum Mainnet and its testnets, and now accounts for nearly a quarter of all synced Mainnet nodes. Our unwavering commitment to Ethereum's growth and stability extends to sidechains and layer 2 solutions. Notably, we were the sole execution layer client to facilitate Gnosis Chain's Merge, transitioning from Aura to Proof of Stake (PoS), and we are actively developing a full-node client to bolster Starknet's decentralization efforts. Our core team equips partners with tools for seamless node set-up, using generated docker-compose scripts tailored to their chosen execution client and preferred configurations for various network types.

DevOps and Infrastructure Management: Our infrastructure team ensures our partners' systems operate securely, reliably, and efficiently. We provide infrastructure design, deployment, monitoring, maintenance, and troubleshooting support, allowing you to focus on your core business operations. Boasting extensive expertise in Blockchain as a Service, private blockchain implementations, and node management, our infrastructure and DevOps engineers are proficient with major cloud solution providers and can host applications in-house or on clients' premises. Our global in-house SRE teams offer 24/7 monitoring and alerts for both infrastructure and application levels. We manage over 5,000 public and private validators and maintain nodes on major public blockchains such as Polygon, Gnosis, Solana, Cosmos, Near, Avalanche, Polkadot, Aptos, and StarkWare L2. Sedge is an open-source tool developed by our infrastructure experts, designed to simplify the complex process of setting up a proof-of-stake (PoS) network or chain validator. Sedge generates docker-compose scripts for the entire validator set-up based on the chosen client, making the process easier and quicker while following best practices to avoid downtime and being slashed.

Cryptography Research: At Nethermind, our Cryptography Research team is dedicated to continuous internal research while fostering close collaboration with external partners. The team has expertise across a wide range of domains, including cryptography protocols, consensus design, decentralized identity, verifiable credentials, Sybil resistance, oracles, and credentials, distributed validator technology (DVT), and Zero-knowledge proofs. This diverse skill set, combined with strong collaboration between our engineering teams, enables us to deliver cutting-edge solutions to our partners and clients.

Smart Contract Development & DeFi Research: Our smart contract development and DeFi research team comprises 40+ world-class engineers who collaborate closely with partners to identify needs and work on value-adding projects. The team specializes in Solidity and Cairo development, architecture design, and DeFi solutions, including DEXs, AMMs, structured products, derivatives, and money market protocols, as well as ERC20, 721, and 1155 token design. Our research and data analytics focuses on three key areas: technical due diligence, market research, and DeFi research. Utilizing a data-driven approach, we offer in-depth insights and outlooks on various industry themes.

Our suite of L2 tooling: Warp is Starknet's approach to EVM compatibility. It allows developers to take their Solidity smart contracts and transpile them to Cairo, Starknet's smart contract language. In the short time since its inception, the project has accomplished many achievements, including successfully transpiling Uniswap v3 onto Starknet using Warp.

- **Voyager** is a user-friendly Starknet block explorer that offers comprehensive insights into the Starknet network. With its intuitive interface and powerful features, Voyager allows users to easily search for and examine transactions, addresses, and contract details. As an essential tool for navigating the Starknet ecosystem, Voyager is the go-to solution for users seeking in-depth information and analysis;
- **Horus** is an open-source formal verification tool for StarkNet smart contracts. It simplifies the process of formally verifying Starknet smart contracts, allowing developers to express various assertions about the behavior of their code using a simple assertion language;
- **Juno** is a full-node client implementation for Starknet, drawing on the expertise gained from developing the Nethermind Client. Written in Golang and open-sourced from the outset, Juno verifies the validity of the data received from Starknet by comparing it to proofs retrieved from Ethereum, thus maintaining the integrity and security of the entire ecosystem.

Learn more about us at nethermind.io.

General Advisory to Clients

As auditors, we recommend that any changes or updates made to the audited codebase undergo a re-audit or security review to address potential vulnerabilities or risks introduced by the modifications. By conducting a re-audit or security review of the modified codebase, you can significantly enhance the overall security of your system and reduce the likelihood of exploitation. However, we do not possess the authority or right to impose obligations or restrictions on our clients regarding codebase updates, modifications, or subsequent audits. Accordingly, the decision to seek a re-audit or security review lies solely with you.

Disclaimer

This report is based on the scope of materials and documentation provided by you to Nethermind in order that Nethermind could conduct the security review outlined in **1. Executive Summary** and **2. Audited Files**. The results set out in this report may not be complete nor inclusive of all vulnerabilities. Nethermind has provided the review and this report on an as-is, where-is, and as-available basis. You agree that your access and/or use, including but not limited to any associated services, products, protocols, platforms, content, and materials, will be at your sole risk. Blockchain technology remains under development and is subject to unknown risks and flaws. The review does not extend to the compiler layer, or any other areas beyond the programming language, or other programming aspects that could present security risks. This report does not indicate the endorsement of any particular project or team, nor guarantee its security. No third party should rely on this report in any way, including for the purpose of making any decisions to buy or sell a product, service or any other asset. To the fullest extent permitted by law, Nethermind disclaims any liability in connection with this report, its content, and any related services and products and your use thereof, including, without limitation, the implied warranties of merchantability, fitness for a particular purpose, and non-infringement. Nethermind does not warrant, endorse, guarantee, or assume responsibility for any product or service advertised or offered by a third party through the product, any open source or third-party software, code, libraries, materials, or information linked to, called by, referenced by or accessible through the report, its content, and the related services and products, any hyperlinked websites, any websites or mobile applications appearing on any advertising, and Nethermind will not be a party to or in any way be responsible for monitoring any transaction between you and any third-party providers of products or services. As with the purchase or use of a product or service through any medium or in any environment, you should use your best judgment and exercise caution where appropriate. FOR AVOIDANCE OF DOUBT, THE REPORT, ITS CONTENT, ACCESS, AND/OR USAGE THEREOF, INCLUDING ANY ASSOCIATED SERVICES OR MATERIALS, SHALL NOT BE CONSIDERED OR RELIED UPON AS ANY FORM OF FINANCIAL, INVESTMENT, TAX, LEGAL, REGULATORY, OR OTHER ADVICE.