Security Review Report
NM-0085 Atlendis Labs

NETHERMIND

(May 16, 2023)

Contents

1 Executive Summary

This document presents the security review conducted by Nethermind on the Atlendis revolving credit line implementation. 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. Atlendis employs Know-Your-Customer (KYC) for both lenders and borrowers.

The audited code consists of 3,300 lines of Solidity with well-written NatSpec documentation for each function. The Atlendis team provided extensive documentation to assist during the audit. **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** 17 points of attention, where one is classified as `High`, four are classified as `Medium`, three are classified as `Low`, eight are classified as `Informational` and one is 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.

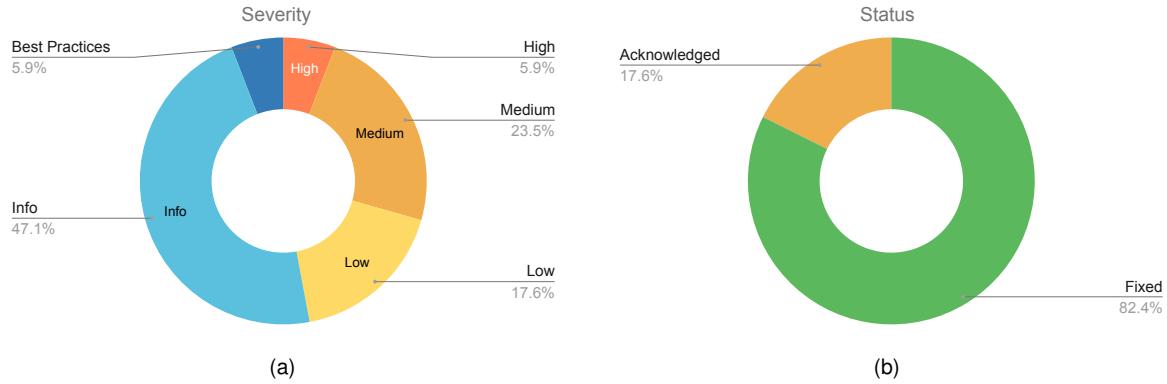

Fig. 1: Distribution of issues: Critical (0), **High** (1), **Medium** (4), **Low** (3), **Undetermined** (0), **Informational** (8), **Best Practices** (1). **Distribution of status: Fixed** (14), **Acknowledged** (3), **Mitigated** (0), **Unresolved** (0)

Summary of the Audit

Audit Type	Security Review
Initial Report	May 12, 2023
Response from Client	May 13, 2023
Final Report	May 16, 2023
Methods	Manual Review, Automated Analysis
Repository	Atlendis
Commit Hash (Initial Audit)	3bb71ee64c83eb3aeabcc1d85d5896a11b638f1f
Documentation	Whitepaper, Borrow/Lend process diagrams
Documentation Assessment	High
Test Suite Assessment	High

2 Audited Files

	Contract	LoC	Comments	Ratio	Blank	Total
1	FactoryRegistry.sol	68	37	54.4%	19	124
2	products/RevolvingCreditLines/RevolvingCreditLine.sol	53	35	66.0%	14	102
3	products/RevolvingCreditLines/RCLFactory.sol	99	28	28.3%	21	148
4	products/RevolvingCreditLines/RCLRewardsManager.sol	50	25	50.0%	10	85
5	products/RevolvingCreditLines/libraries/AccrualsHelpers.sol	37	35	94.6%	7	79
6	products/RevolvingCreditLines/libraries/LenderLogic.sol	656	296	45.1%	68	1020
7	products/RevolvingCreditLines/libraries/Errors.sol	34	5	14.7%	3	42
8	products/RevolvingCreditLines/libraries/BorrowerLogic.sol	317	156	49.2%	36	509
9	products/RevolvingCreditLines/libraries/DataTypes.sol	74	7	9.5%	10	91
10	products/RevolvingCreditLines/libraries/PositionHelpers.sol	296	150	50.7%	35	481
11	products/RevolvingCreditLines/libraries/RCLValidation.sol	21	29	138.1%	10	60
12	products/RevolvingCreditLines/libraries/RevolvingCreditLineDeployer.sol	15	6	40.0%	2	23
13	products/RevolvingCreditLines/modules/RCLGovernance.sol	110	56	50.9%	33	199
14	products/RevolvingCreditLines/modules/RCLState.sol	109	50	45.9%	21	180
15	products/RevolvingCreditLines/modules/RCLBorrowers.sol	85	31	36.5%	23	139
16	products/RevolvingCreditLines/modules/RCLLenders.sol	244	77	31.6%	56	377
17	products/RevolvingCreditLines/modules/interfaces/IRCLBorrowers.sol	9	36	400.0%	6	51
18	products/RevolvingCreditLines/modules/interfaces/IRCLGovernance.sol	33	118	357.6%	25	176
19	products/RevolvingCreditLines/modules/interfaces/IRCLState.sol	18	27	150.0%	4	49
20	products/RevolvingCreditLines/modules/interfaces/IRCLLenders.sol	33	103	312.1%	14	150
21	products/RevolvingCreditLines/interfaces/IRCLRewardsManager.sol	7	29	414.3%	4	40
22	products/RevolvingCreditLines/interfaces/IRevolvingCreditLine.sol	23	27	117.4%	5	55
23	libraries/PoolTimelockLogic.sol	40	21	52.5%	6	67
24	libraries/FixedPointMathLib.sol	78	53	67.9%	17	148
25	libraries/TimeValue.sol	14	15	107.1%	3	32
26	libraries/FundsTransfer.sol	35	27	77.1%	5	67
27	common/fees/PoolTokenFeesController.sol	129	94	72.9%	38	261
28	common/fees/IFeesController.sol	28	117	417.9%	21	166
29	common/roles-manager/Managed.sol	22	19	86.4%	7	48
30	common/roles-manager/StandardRolesManager.sol	53	42	79.2%	16	111
31	common/roles-manager/interfaces/IRolesManager.sol	8	32	400.0%	5	45
32	common/roles-manager/interfaces/IManaged.sol	10	29	290.0%	7	46
33	common/roles-manager/interfaces/IStandardRolesManager.sol	7	20	285.7%	4	31
34	common/custodian/CustodianTimelockLogic.sol	36	21	58.3%	5	62
35	common/custodian/PoolCustodian.sol	179	106	59.2%	72	357
36	common/custodian/IPoolCustodian.sol	47	171	363.8%	37	255
37	common/custodian/CustodianStorage.sol	19	12	63.2%	5	36
38	common/custodian/adapters/IAdapter.sol	9	33	366.7%	6	48
39	common/custodian/adapters/AaveV2V3Adapter.sol	53	33	62.3%	12	98
40	common/custodian/adapters/NeutralAdapter.sol	30	27	90.0%	8	65
41	common/custodian/adapters/extensions/IAaveLendingPool.sol	15	28	186.7%	3	46
42	interfaces/IFactoryRegistry.sol	34	67	197.1%	11	112
43	interfaces/IPool.sol	32	60	187.5%	14	106
44	interfaces/ITimelock.sol	12	40	333.3%	9	61
45	interfaces/IPositionDescriptor.sol	4	11	275.0%	1	16
46	interfaces/IProductFactory.sol	15	28	186.7%	4	47
	Total	**3300**	**2469**	**74.8%**	**742**	**6511**

3 Summary of Issues

	Finding	Severity	Update
1	Attackers can front-run to cause borrowers to borrow or lenders to exit at a high rate	High	Acknowledged
2	Function `exit(...)` is vulnerable to ERC777 reentrancy attack	Medium	Fixed
3	Insufficient amount of tokens due to rounding issues in the `lateRepayFees` calculation	Medium	Fixed
4	Wrong formula to calculate `actualRate` in the function `registerExitFees(...)`	Medium	Fixed
5	Centralization risks	Low	Acknowledged
6	Rollover period can be used to lock lender funds temporarily	Low	Fixed
7	Switching yield provider poses delegatecall risk	Low	Fixed
8	Granting and revoking governance role	Info	Fixed
9	Setting `minDepositAmount = 0` could cause the position counting to behave incorrectly	Info	Fixed
10	Inconsistent `minDepositAmount` input validation	Info	Fixed
11	Incorrect `minDepositAmount` input validation in `validateDeployment`	Medium	Fixed
12	Shadowing local variables	Info	Fixed
13	The conditions in the `withdraw(...)` function's if-else block can be simplified	Info	Fixed
14	Usage of deprecated `safeApprove`	Info	Fixed
15	Zeroed `EXIT_FEES_INFLECTION_THRESHOLD` might lead to inaccurate exit fee calculation	Info	Fixed
16	`PoolTokenFeesController` deployment args must align with token decimals	Info	Acknowledged
17	Unused library import in `RCLBorrowers`	Best Practices	Fixed

4 System Overview

The audit is based on seven contracts:

a) **RevolvingCreditLine**
b) **RCLFactory**
c) **FactoryRegistry**
d) **StandardRolesManager**
e) **PoolCustodian**
f) **Adapter**
g) **PoolTokenFeesController**

Fig. 2 presents the interaction diagram of contracts.

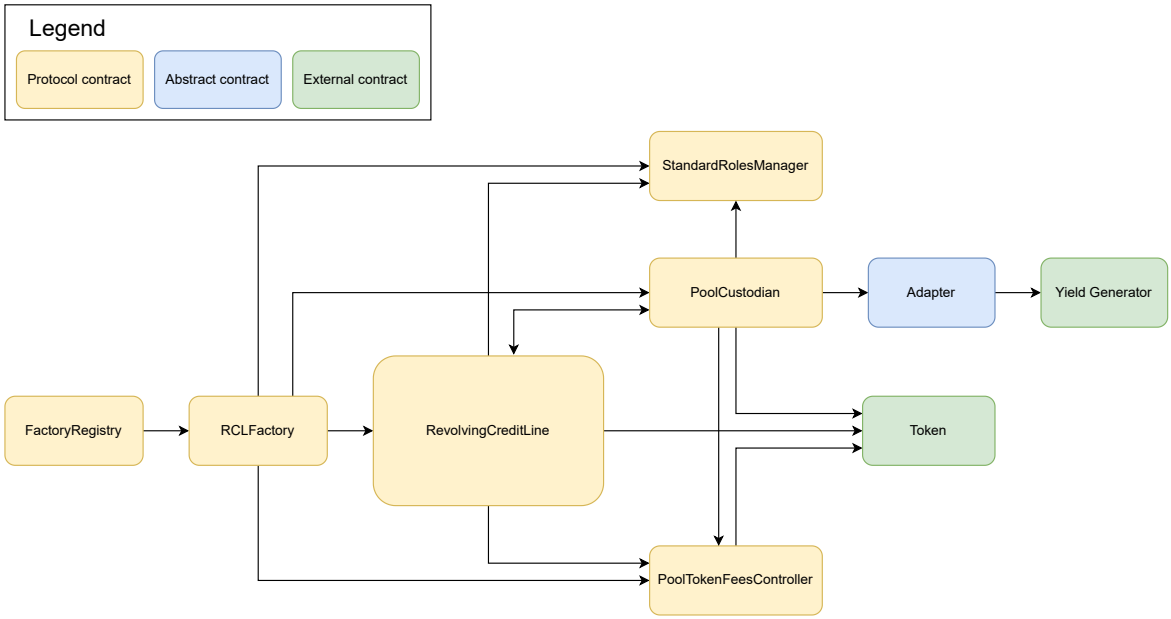

Fig. 2: Interaction Diagram of Contracts

The **RevolvingCreditLine** contract implements all the logic of the revolving credit line system. It is also the entry point of interaction, allowing lenders and borrowers to participate in lending and borrowing activities within the protocol. The contract will enable lenders to deposit funds into a pool, specifying the amount and the interest rate they desire for their funds. Borrowers can use these funds to borrow loans. The contract matches lenders' positions with borrowers based on the specified interest rates. Lenders can change the interest rate of their positions and withdraw their funds when not borrowed. On the other hand, Borrowers can borrow funds from the order book by matching positions with lenders at prevailing interest rates. They can borrow multiple times if the maximum borrowable amount is not reached and there are available funds. Borrowers are also responsible for repaying their loans to start a new loan cycle.

The **RCLFactory** contract is responsible for deploying Revolving Credit Line (RCL) contracts. The main purpose of the `RCLFactory` contract is to streamline the creation and initialization of a `RevolvingCreditLine` pool, ensuring consistency and standardized functionality.

The **FactoryRegistry** contract serves as a registry for product factories within the protocol. Its main purpose is to manage and utilize these product factories to deploy instances of products and register them within the system.

The **StandardRolesManager** contract handles all role management within a pool. Its main purpose is to define and manage the permissions and roles of different participants in each pool. It is specifically designed to define and manage the following roles:

- Lender: This role is assigned to participants authorized to provide liquidity to pools for borrowers. They can engage in actions such as depositing funds, changing interest rates, or withdrawing their positions.
- Borrower: This role is assigned to participants who are authorized to borrow. Borrowers have their own liquidity pools and can borrow from funds provided by lenders. They can engage in actions such as borrowing, repaying, and rolling loans over.
- Operator: By default, positions within the pool are not transferable unless the transfer is initiated by an operator. The operator role allows for transferring positions between different addresses or accounts, enabling flexibility in managing ownership and control over positions.

The **PoolCustodian** contract handles the custody of funds. It serves as a dedicated storage and management mechanism for the funds associated with a specific pool. Lenders can deposit their tokens into the custodian contract, and those funds can either remain in the contract itself or be further deposited into a third-party yield provider like `Aave` or `Compound` to generate additional yield.

The **Adapter** contract is a connector or interface between a `PoolCustodian` and a specific yield provider. Its primary purpose is to implement the necessary logic to facilitate actions such as depositing, withdrawing, and computing rewards associated with the yield provider. The derived contracts of Adapter include **NeutralAdapter** and **AaveV2V3Adapter** .

The **PoolTokenFeesController** contract is specifically designed to handle and manage fees associated with pool tokens within each pool. Its main purpose is to control and regulate the collection and distribution of fees incurred during various actions, such as borrowing, interest, withdrawal, and exit fees.

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 [High] Attackers can front-run causing borrowers to borrow or lenders to exit at a high rate

File(s): RCLLenders.sol

Description: The RevolvingLineCredit contract allows lenders to deposit funds at a specific rate, and borrowers can borrow them from the lowest to the highest rate. However, malicious lenders can change the rate at any time before their positions are borrowed, forcing the borrower to borrow at a higher rate by front-running the borrowing transaction. This issue also exists if lenders want to exit their position because when a borrowed position exits, it has to loop through every tick and remove funds, similar to the deposit process.

Attack Scenario

1. Alice (the attacker) adds $1M at the lowest rate tick. Bob (the borrower) sees this as a great term and calls borrow(...) to borrow $1M from Alice;
2. While Bob's transaction is in the mempool, Alice front-runs and updates the rate from the lowest rate to the highest rate;
3. When Bob's transaction is executed, he will borrow funds at the highest rate, which is not what Bob expected or intended;

Recommendation(s): Consider adding protection for the borrower in the function borrow(...). For example, consider allowing the borrower to specify the maximum tick they want to borrow from in this call.

Status: Acknowledged

Update from the client: The protocol is by the construction of a hybrid, built with both smart contracts and legal contracts. As a consequence, the users will have to comply with strict identity checks, and will explicitly not be allowed to perform such actions in the protocol. Additionally, addressing this would add more complexity to the protocol, which does not seem necessary considering the potential consequences at this time. This finding will be considered in further developments, as the Atlendis team will work on new iterations of the smart contracts.

6.2 [Medium] Function exit(...) is vulnerable to ERC777 reentrancy attack

File(s): RCLLenders.sol

Description: The RCLLenders contract contains an exit(...) function, which allows lenders to exit their position before maturity. This function calculates the number of tokens the lender will receive, updates the contract state, and sends the funds to the lender. If the lender wants to fully exit their position by withdrawing their entire borrowed amount (i.e., borrowedAmountToExit == type(uint256).max), the position is burned after the funds are transferred out.

However, this implementation does not follow the check-effect-interactions pattern and may be vulnerable to reentrancy attacks if the token is an ERC777 with a call hook on the receiver. An attacker can call exit(...) again using the call hook before the position is burned, allowing them to steal the funds. The code snippet for the exit(...) function is shown below:

```
////////////////////////////////////////////////////////
// @audit Funds are transferred out here
////////////////////////////////////////////////////////
FundsTransfer.chargedWithdraw({
    token: TOKEN,
    custodian: CUSTODIAN,
    recipient: msg.sender,
    amount: toBeWithdrawn - fees,
    fees: fees
});

////////////////////////////////////////////////////////////////////////////////
// @audit The position is burned after the funds are transferred out
////////////////////////////////////////////////////////////////////////////////
if (borrowedAmountToExit == type(uint256).max) {
    burn(positionId);
    emit Exited(positionId, true, unborrowedAmount, actualBorrowedAmountToExit, toBeWithdrawn - fees, fees);
} else {
    emit Exited(positionId, false, unborrowedAmount, actualBorrowedAmountToExit, toBeWithdrawn - fees, fees);
}
```

Recommendation(s): To prevent reentrancy attacks, consider following the check-effect-interactions pattern or using ReentrancyGuard.

Status: Fixed

Update from the client: Updated logic to apply the check effect pattern according to the recommendation.

Update from Nethermind: Solved in https://github.com/Atlendis/priv-contracts-v2/pull/612.

6.3 [Medium] Insufficient amount of tokens due to rounding issues in the `lateRepayFees` calculation

File(s): BorrowerLogic.sol

Description: The function `repay(...)` will loop through every tick to calculate the late repayment fee for each tick. But in the end, the total amount of the late repayment fee is calculated using the total borrowed value instead of summing up the fee in each individual tick.

The issue is the fee calculation rounding up, so if there are `n` ticks, the potential amount increased is $2 \times n$ token weis because each tick rounds up twice. While the total fee is rounded up only once, the pool might potentially lack $2 \times n - 1$ token weis.

```
function repay(
    ...
) public returns (uint256 fees, uint256 lateRepayFees) {
    ...
    while (currentInterestRate <= maxRate) {
        DataTypes.Tick storage tick = ticks[currentInterestRate];
        if (tick.latestLoanId == currentLoan.id) {
            if (block.timestamp > currentLoan.maturity) {
                registerLateRepaymentAccruals(tick, block.timestamp - currentLoan.maturity, lateInterestFeesRate);
            }
            ...
        }
        currentInterestRate += spacing;
    }
    ////////////////////////////////////////////////////////////
    // @audit Rounding up once using `totalBorrowed`
    //        While it's rounded up twice in each tick in `registerLateRepaymentAccruals()`
    ////////////////////////////////////////////////////////////
    lateRepayFees = block.timestamp > currentLoan.maturity
        ? AccrualsHelpers.accrualsForUp(totalBorrowed, block.timestamp - currentLoan.maturity, lateInterestFeesRate)
        : 0;
}

function registerLateRepaymentAccruals(
    ...
) public {
    if (tick.baseEpochsAmounts.borrowed != 0) {
        tick.yieldFactor += AccrualsHelpers.calculateYieldFactorIncrease(tick, timeDelta, rate);
    }
    if (tick.newEpochsAmounts.borrowed != 0) {
        uint256 newEpochsAccruals = AccrualsHelpers.accrualsForUp(tick.newEpochsAmounts.borrowed, timeDelta, rate);
        tick.newEpochsAmounts.toBeAdjusted += newEpochsAccruals;
    }
    if (tick.detachedAmounts.borrowed != 0) {
        uint256 detachedAccruals = AccrualsHelpers.accrualsForUp(tick.detachedAmounts.borrowed, timeDelta, rate);
        tick.detachedAmounts.toBeAdjusted += detachedAccruals;
    }
}
```

Recommendation(s): Consider calculating the total late repayment fees by summing up the late repayment fee in each tick.

Status: Fixed

Update from the client: Updated logic to compute fees only a single time and use the result of the calculation twice, as opposed to computing the value in place for each use. Fixed the rounding according to the finding as a result.

Update from Nethermind: Solved in https://github.com/Atlendis/priv-contracts-v2/pull/588.

6.4 [Medium] Wrong formula to calculate `actualRate` in the function `registerExitFees(...)`

File(s): PoolTokenFeesController.sol

Description: Within the `PoolTokenFeesController` contract, the function `registerExitFees(...)` calculates the amount of fees users must pay based on the `amount` and `timeUntilMaturity`. The exit fee increases as the loan approaches maturity to discourage users from exiting right before maturity. However, the formula for cases in which `loanAdvancement > EXIT_FEES_INFLECTION_THRESHOLD` is currently implemented incorrectly in the codebase. This results in users having to pay more fees than expected.

```
if (loanAdvancement > EXIT_FEES_INFLECTION_THRESHOLD) {
    ////////////////////////////////////////////////////
    // @audit It should be divided by `1 - EXIT_FEES_INFLECTION_THRESHOLD`
    //        instead of `EXIT_FEES_INFLECTION_THRESHOLD`
    ////////////////////////////////////////////////////
    actualRate =
        EXIT_FEES_INFLEXION_RATE -
        (EXIT_FEES_INFLEXION_RATE - EXIT_FEES_MIN_RATE)
            .mul(loanAdvancement - EXIT_FEES_INFLECTION_THRESHOLD)
            .div(EXIT_FEES_INFLECTION_THRESHOLD);
} else {
    actualRate =
        EXIT_FEES_MAX_RATE -
        (EXIT_FEES_MAX_RATE - EXIT_FEES_INFLEXION_RATE).mul(loanAdvancement).div(
            EXIT_FEES_INFLECTION_THRESHOLD
        );
}
```

Recommendation(s): Consider changing the denominator of the formula to `ONE - EXIT_FEES_INFLECTION_THRESHOLD` instead of `EXIT_FEES_INFLECTION_THRESHOLD`.

Status: Fixed

Update from the client: Updated logic to fix the issue, as well as clarify the terms and their use. For example, a 90% loan advancement now means that the loan is 90% done, while it meant that it was 10% done before.

Update from Nethermind: Solved in https://github.com/Atlendis/priv-contracts-v2/pull/599.

6.5 [Low] Centralization risks

File(s): src/*

Description: Atlendis relies on governance to manage some parts of the protocol through governance-only functions. While governance usually implies a decentralized voting system, in this case, the governance address is a multi-sig contract controlled by the Atlendis team. This places a reasonable amount of trust in the Atlendis team to manage the protocol correctly and act in the user's best interest.

While some governance-only functions employ timelock features to give users time to react to potential protocol changes, many functions allow for instant protocol parameter changes. It is possible for governance to cause some significant damage to the protocol and its users. Some of the centralization risks are:

- Removing lender role once a deposit has been made;
- Initiating a rollover period and removing the borrower role from the corresponding borrower;
- Initiating the non-standard repayment procedure unnecessarily;
- Initiating the rescue procedure unnecessarily;

Recommendation(s): Consider adding a timelock feature to governance-only functions, which can change important protocol parameters, actions, or roles.

Status: Acknowledged

Update from the client: By construction, the Atlendis protocol is a hybrid between smart contracts and legal contracts, and as a consequence relies heavily on governance to enforce on chain what was agreed off chain. For these reasons, governance has to maintain full control over the roles in the platform, in case users would not comply with the legal requirements at any time during a loan and roles have to be managed dynamically. However, in some cases, such as switching yield provider, non standard repayment procedure and updating rollover period, governance can leave a choice to users that do not agree with the decision to exit from the pool if they want. For these reasons, in these cases a timelock was implemented according to recommendation.

Update from Nethermind: Rollover delay implemented in https://github.com/Atlendis/priv-contracts-v2/pull/615

6.6 [Low] Rollover period can be used to lock lender funds temporarily

File(s): RCLState.sol

Description: To allow borrowers to "roll over" their loans, a `rolloverPeriod` is tracked in the contract that can be used to specify a time window where lenders cannot withdraw their funds. This is necessary to guarantee an available amount and rate for the borrower. A borrower can use this feature to temporarily prevent lender withdrawals until governance intervenes by simply not borrowing after requesting for a loan rollover. The modifier that prevents rollover withdrawals is shown below:

```
modifier onlyLoanNotInRolloverPeriod() {
    if (rolloverPeriod > 0 && currentLoan.maturity > 0 && block.timestamp > currentLoan.maturity - rolloverPeriod)
        revert RevolvingCreditLineErrors.RCL_LOAN_IN_ROLLOVER_PERIOD();
    _;
}
```

Assuming that governance has accepted the borrower's request for a rollover, the `rolloverPeriod` will be non-zero. The check to see if the current time is within the rollover period time window (`block.timestamp > currentLoan.maturity - rolloverPeriod`) has a lower limit but no upper limit. As long as the `rolloverPeriod` is non-zero and the borrower doesn't update their current loan maturity, lenders cannot withdraw.

To address this, governance can intervene by setting the `rolloverPeriod` to zero. However, there still may be a reasonable wait time when lenders cannot withdraw.

Recommendation(s): Consider adding an upper bound check to the rollover period check, such that if a lender requests a rollover period and then decides not to rollover their loan, users will still be able to withdraw after some time rather than needing to wait for governance to unlock their funds.

Status: Fixed

Update from the client: Updated logic so that the rollover period stops at maturity.

Update from Nethermind: Solved in PR.

6.7 [Low] Switching yield provider poses `delegatecall` risk

File(s): PoolCustodian.sol

Description: The function `startSwitchYieldProviderProcedure` is used to begin switching the yield provider. As a safeguard for users, this process is constrained by a timelock. However, when starting the switching process, a `delegatecall` is immediately executed using the `newAdapter` logic. This defeats the point of the `timelock`, as potentially arbitrary logic can be executed by the governance immediately rather than having to wait for the timelock. The code is shown below:

```
function startSwitchYieldProviderProcedure(
   ...
) external onlyGovernance {
    if (delay < SWITCH_YP_MIN_TIMELOCK_DELAY) revert TIMELOCK_DELAY_TOO_SMALL();

    if (!IAdapter(newAdapter).supportsInterface(type(IAdapter).interfaceId)) revert ADAPTER_NOT_SUPPORTED();

    ////////////////////////////////////////////////////////////////////////////////
    // @audit A delegatecall to an arbitrary address is executed at the start of timelock
    ////////////////////////////////////////////////////////////////////////////////
    bytes memory returnData = adapterDelegateCall(
        newAdapter,
        abi.encodeWithSignature('supportsToken(address)', newYieldProvider)
    );
    ...
}
```

Recommendation(s): The `delegatecall` to check if the adapter is supported can be placed in the function `executeTimelock(...)`. This will allow the check to occur after the timelock window rather than immediately.

Status: Fixed

Update from the client: Moved the validation logic to the execution part of the timelock so that the arbitrary logic is executed after the timelock delay. That way, lenders that do not agree with the update can exit if they want.

Update from Nethermind: Solved in https://github.com/Atlendis/priv-contracts-v2/pull/604.

6.8 [Info] Granting and revoking governance role

File(s): StandardRolesManager.sol

Description: Granting and revoking the governance role (or the DEFAULT_ADMIN_ROLE) is done using the functions batchGrantRole(...) and batchRevokeRole(...). As the governance role is crucial to the protocol's operation, it would be better to ensure that at least one active EOA/contract has that role at all times. Errors can happen while transferring the role to a new address, Ex: instead of calling the grant function first, the call is initiated to the revoke first. Another possible scenario is while revoking multiple addresses with this role, the caller's address is also included.

Recommendation(s): Consider implementing a mechanism to ensure that at least one has the governance role at any time. When proposing a new governance role, a set-then-claim approach is recommended to ensure that an incorrect address isn't set as governance.

Status: Fixed

Update from the client: Updated logic to follow the recommendation, do not allow governance to renounce to the role or revoke themselves, and implement a two-step process to grant governance role.

6.9 [Info] Setting `minDepositAmount = 0` could cause the position counting to behave incorrectly

File(s): RCLGovernance.sol

Description: In each tick of the RevolvingCreditLines, there are multiple epochs, and in each epoch, the positionsCount is stored to help reset the state of the epoch when there are no positions left. When users withdraw a position, the positionsCount is only decreased if they set the amount to type(uint256).max, which is a full withdrawal. An attacker can still withdraw the full amount without updating the positionsCount by specifying the exact value of the position as the amount. This results in the position being fully withdrawn but the positionsCount not being updated.

```
////////////////////////////////////////////////////////////////////////
// @audit isFullWithdraw is true only when amount is type(uint256).max
////////////////////////////////////////////////////////////////////////
bool isFullWithdraw = expectedWithdrawnAmount == type(uint256).max;
...
safeSubstractNewEpochsAmounts({
    tick: tick,
    shouldDecreasePositionCount: isFullWithdraw,
    borrowedAmountToWithdraw: 0,
    availableAmountToWithdraw: withdrawnAmount,
    toBeAdjustedAmountToWithdraw: withdrawnAmount,
    optedOutAmountToWithdraw: 0
});
```

However, when withdrawing, the remaining amount of the position is checked to be larger than minDepositAmount, so the issue only happens when governance sets minDepositAmount = 0.

Recommendation(s): Consider adding a check to ensure that minDepositAmount cannot be set to too small values.

Status: Fixed

Update from the client: As a result of NM-10 and NM-11, minDepositAmount can never be set to zero.

Update from Nethermind: Solved in https://github.com/Atlendis/priv-contracts-v2/pull/602.

6.10 [Info] Inconsistent `minDepositAmount` input validation

File(s): RCLGovernance.sol

Description: The value minDepositAmount can be set during deployment in RCLState.constructor(...) and changed by governance with RCLGovernance.setMinDepositAmount(...). The constructor has input validation for minDepositAmount, but setMinDepositAmount does not validate inputs.

Recommendation(s): Ensure input validation for minDepositAmount is consistent throughout the constructor and all setters. For completeness, it should be noted that there is another setter LoanGovernance.updateMinDepositAmount(...), which is out-of-scope for this audit that is also missing input validation.

Status: Fixed

Update from the client: Updated logic so that minDepositAmount can never be set to zero in any case in the pool.

6.11 [Medium] Incorrect `minDepositAmount` input validation in `validateDeployment`

File(s): RCLValidation.sol

Description: When deploying a contract inheriting from `RCLState`, a variety of parameters are packed and provided as `bytes` in the `parametersConfig` argument. This argument is then passed to the function `RCLValidation.validateDeployment(...)` to validate some of the packed values. One of these validated parameters is `minDepositAmount`, which is checked to ensure that the minimum deposit amount cannot be zero. The function is shown below:

```
function validateDeployment(bytes memory parametersConfig) public pure {
    if (parametersConfig.length != 256) revert INVALID_PRODUCT_PARAMS();

    (, uint256 minRate, uint256 maxRate, uint256 rateSpacing, , uint256 minDepositAmount, ) = abi.decode(
        parametersConfig,
        (uint256, uint256, uint256, uint256, uint256, uint256, uint256)
    );

    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 (minDepositAmount == 0) revert INVALID_ZERO_MIN_DEPOSIT_AMOUNT();
}
```

The `minDepositAmount` argument is loaded from the 6th packed uint256, and validation is completed on that value. However, in the `RCLState` constructor, after validation, the `minDepositAmount` is set from the 8th packed uint256 instead. The 6th packed uint256 contains the value for `LOAN_DURATION`. As a result, the input validation for the minimum deposit amount is done on the loan duration, and the minimum deposit amount has no input validation at all, allowing a `minDepositAmount` of zero. A snippet from the `RCLState` constructor is shown below:

```
constructor(
    ...
) Managed(rolesManager) {
    RCLValidation.validateDeployment(parametersConfig);

    ...

    (
        maxBorrowableAmount,
        MIN_RATE,
        MAX_RATE,
        RATE_SPACING,
        REPAYMENT_PERIOD,
        LOAN_DURATION, // @audit The minimum deposit amount validation is done on this value instead
        LATE_REPAYMENT_FEE_RATE,
        minDepositAmount // @audit `minDepositAmount` is the 8th packed uint256, not the 6th
    ) = abi.decode(parametersConfig, (uint256, uint256, uint256, uint256, uint256, uint256, uint256, uint256));

    ...

}
```

Recommendation(s): Ensure the input validation and values read between `RCLState.constructor` and `RCLValidation.validateDeployment` are consistent.

Status: Fixed

Update from the client: Updated validation logic to validate the right parameter at pool deployment.

Update from Nethermind: Solved in https://github.com/Atlendis/priv-contracts-v2/pull/587.

6.12 [Info] Shadowing local variables

File(s): AaveV2V3Adapter.sol

Description: The function supportsToken(...) in AaveV2V3Adapter consumes an input parameter named yieldProvider, while a state variable with the same name is defined in CustodianStorage, which AaveV2V3Adapter inherits. Shadowing can lead to unexpected behavior and bugs, making it difficult to determine which variable is referred to in different parts of the code.

```
////////////////////////////////////
// @audit shadowing variable
////////////////////////////////////
function supportsToken(address yieldProvider) external view returns (bool) {
    return IAaveLendingPool(yieldProvider).getReserveNormalizedIncome(address(token)) >= RAY;
}
```

Recommendation(s): Consider renaming the input parameter used in the function supportsToken(...) to avoid shadowing the state variable yieldProvider inherited from CustodianStorage.

Status: Fixed

Update from the client: Updated the variable name according to the recommendation.

Update from Nethermind: Solved in https://github.com/Atlendis/priv-contracts-v2/pull/601.

6.13 [Info] The conditions in the withdraw(...) function's if-else block can be simplified

File(s): LenderLogic.sol

Description: In the function withdraw(...), the logic differs for when the tick is not borrowed and when a loan is ongoing, so they are separated by an if-else block. However, the condition to check currentLoan.maturity > 0 in the else branch could be removed since it is guaranteed to be larger than 0 in the if checks.

```
if (currentLoan.maturity == 0 || tick.latestLoanId < currentLoan.id) {
    ...
else {
    ////////////////////////////////////////////////////////////////////////
    // @audit currentLoan.maturity is always > 0 in this branch
    ////////////////////////////////////////////////////////////////////////
    if (currentLoan.maturity > 0 && tick.currentEpochId != position.epochId)
        revert RevolvingCreditLineErrors.RCL_LOAN_RUNNING();
    ...
}
```

Recommendation(s): Consider removing the condition currentLoan.maturity > 0.

Status: Fixed

Update from the client: Simplified the targeted condition in the withdraw function according to the recommendation.

Update from Nethermind: Solved in https://github.com/Atlendis/priv-contracts-v2/pull/613.

6.14 [Info] Usage of deprecated safeApprove

File(s): AaveV2V3Adapter.sol, FundsTransfer.sol

Description: The function safeApprove(...) is now deprecated, and its usage is discouraged, as mentioned in this comment.

Recommendation(s): As per OpenZeppelin documentation, "whenever possible, use safeIncreaseAllowance and safeDecreaseAllowance instead".

Status: Fixed

Update from the client: Replaced safeApprove with safeIncreaseAllocation according to the recommendation.

Update from Nethermind: Solved in https://github.com/Atlendis/priv-contracts-v2/pull/603.

6.15 [Info] Zeroed EXIT_FEES_INFLECTION_THRESHOLD might lead to inaccurate exit fee calculation

File(s): PoolTokenFeesController.sol

Description: In the function registerExitFees(...), the actualRate is calculated only if EXIT_FEES_INFLECTION_THRESHOLD is a non-zero value. If, by choice or mistake, that variable is not set during the contract creation, it will lead to the wrong calculation of fees during a position exit.

```
function registerExitFees(uint256 amount, uint256 timeUntilMaturity) external onlyPool returns (uint256 fees) {
    uint256 loanAdvancement = (ONE * timeUntilMaturity) / LOAN_DURATION;
    ////////////////////////////////////////////////////////////////////////////
    // @audit The actualRate default value could be set as EXIT_FEES_MIN_RATE
    //        instead of zero.
    ////////////////////////////////////////////////////////////////////////////
    uint256 actualRate = 0;
    if (EXIT_FEES_INFLECTION_THRESHOLD != 0) {
        if (loanAdvancement > EXIT_FEES_INFLECTION_THRESHOLD) {
            actualRate =
                EXIT_FEES_INFLEXION_RATE -
                (EXIT_FEES_INFLEXION_RATE - EXIT_FEES_MIN_RATE)
                    .mul(loanAdvancement - EXIT_FEES_INFLECTION_THRESHOLD)
                    .div(EXIT_FEES_INFLECTION_THRESHOLD);
        } else {
            actualRate =
                EXIT_FEES_MAX_RATE -
                (EXIT_FEES_MAX_RATE - EXIT_FEES_INFLEXION_RATE).mul(loanAdvancement).div(
                    EXIT_FEES_INFLECTION_THRESHOLD
                );
        }
    }
    fees = amount.mulUp(actualRate);
    dueFees += fees;
    totalFees += fees;

    emit ExitFeesRegistered(TOKEN, fees, actualRate);
}
```

Recommendation(s): Either assign actualRate to EXIT_FEES_MIN_RATE if constant fees are levied when no inflexion threshold exists. Or, instead of the check:

```
if (EXIT_FEES_INFLECTION_THRESHOLD != 0) {
    if (loanAdvancement > EXIT_FEES_INFLECTION_THRESHOLD) {
    ...
    } else {
    ...
    }
}
```

Consider changing it like this:

```
if (EXIT_FEES_INFLECTION_THRESHOLD != 0  && loanAdvancement > EXIT_FEES_INFLECTION_THRESHOLD) {
...
} else {
...
}
```

Status: Fixed

Update from the client: Changed the logic so that a zero exit fees inflection threshold computes the exit fees as a simple slope between the inflection rate and max rate.

Update from Nethermind: Fixed in https://github.com/Atlendis/priv-contracts-v2/pull/599

6.16 [Info] `PoolTokenFeesController` deployment args must align with token decimals

File(s): `PoolTokenFeesController.sol`

Description: The contract `PoolTokenFeesController` sets the following immutable values at deployment:

```
uint256 public immutable WITHDRAWAL_FEES_RATE;
uint256 public immutable EXIT_FEES_INFLECTION_THRESHOLD;
uint256 public immutable EXIT_FEES_MIN_RATE;
uint256 public immutable EXIT_FEES_INFLEXION_RATE;
uint256 public immutable EXIT_FEES_MAX_RATE;
uint256 public immutable BORROWING_FEES_RATE;
uint256 public immutable INTEREST_FEES_RATE;
```

These values are consistently 18 decimals, as seen in the test file `RCL.fees.t.sol`, and are used to calculate fee pricing. When the pool's token is 18 decimals, calculations are correct. However, calculations may need to be corrected when using a token with decimals other than 18 (Ex: USDC with six decimals). To demonstrate this, consider the following snippets from the function `registerExitFees(...)`:

```
uint256 loanAdvancement = (ONE * timeUntilMaturity) / LOAN_DURATION;
actualRate =
                    EXIT_FEES_INFLEXION_RATE -
                   (EXIT_FEES_INFLEXION_RATE - EXIT_FEES_MIN_RATE)
                    .mul(loanAdvancement - EXIT_FEES_INFLECTION_THRESHOLD)
                    .div(EXIT_FEES_INFLECTION_THRESHOLD);
```

The value `ONE` is determined by the number of decimals in the token, as shown below:

```
ONE = 10**ERC20(TOKEN).decimals();
```

This leads to the variable `loanAdvancement` being six decimals. When this value is used to calculate the argument for the `FixedPointMathLib` multiply function, an overflow revert will occur since `loanAdvancement` is six decimals, but `EXIT_FEES_INFLECTION_THRESHOLD` is 18 decimals.

Recommendation(s): When deploying each pool, ensure that the `PoolTokenFeesController` constructor arguments are scaled appropriately for the token decimals.

Status: Acknowledged

Update from the client: Noted, verified that our deployment scripts comply with the remark.

6.17 [Best Practices] Unused library import in `RCLBorrowers`

File(s): `RCLBorrowers.sol`

Description: The contract `RCLBorrowers` imports the library `TimeValue`. However, no functionality from `TimeValue` is used.

Recommendation(s): Consider removing the unused library in `RCLBorrowers.sol`.

Status: Fixed

Update from the client: Removed the unused import as recommended.

Update from Nethermind: Solved in https://github.com/Atlendis/priv-contracts-v2/pull/618.

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 plays a critical role in software development, as it enables effective communication between developers, testers, users, and other stakeholders involved in the software development process. 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 that describes the design and technical details of the smart contract. 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 provided extensive documentation in the form of a PDF Whitepaper, and PNG images about user flow on different actions and protocol lifecycles. The whitepaper encompasses a broad overview of the protocol modules and detailed insights into users' actions and associated fees. Additional documents about the user action flow helped understand each state change under the hood, during an action. 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 145 files with 0.8.13
[] Solc 0.8.13 finished in 101.14s
Compiler run successful
```

8.2 Tests Output

```
> forge test --match-contract RCL
[] Compiling...
No files changed, compilation skipped

Running 4 tests for test/unit/products/revolving-credit-line/RCLLenders.transferFrom.t.sol:RCLLenderTranserFromTest
Test result: ok. 4 passed; 0 failed; finished in 6.33ms

Running 18 tests for test/unit/products/revolving-credit-line/RCL.fees.t.sol:RCLFeesTest
Test result: ok. 18 passed; 0 failed; finished in 40.75ms

Running 41 tests for test/unit/products/revolving-credit-line/RCLLenders.detach.t.sol:RCLLendersDetachTest
Test result: ok. 41 passed; 0 failed; finished in 43.88ms

Running 36 tests for test/unit/products/revolving-credit-line/RCLLenders.exit.t.sol:RCLLendersExitTest
Test result: ok. 36 passed; 0 failed; finished in 57.61ms

Running 36 tests for test/unit/products/revolving-credit-line/RCLLenders.withdraw.t.sol:RCLLendersWithdrawTest
Test result: ok. 36 passed; 0 failed; finished in 35.48ms

Running 12 tests for test/unit/products/revolving-credit-line/RCLFactory.t.sol:RCLFactoryTest
Test result: ok. 12 passed; 0 failed; finished in 8.80ms

Running 32 tests for test/unit/products/revolving-credit-line/RCLGovernance.t.sol:RCLGovernanceTest
Test result: ok. 32 passed; 0 failed; finished in 9.72ms

Running 46 tests for test/unit/products/revolving-credit-line/RCLBorrowers.repay.t.sol:RCLBorrowersRepayTest
Test result: ok. 46 passed; 0 failed; finished in 60.17ms

Running 6 tests for test/unit/rewards/RCLRewardsManagerPositionUpdate.t.sol:RCLRewardsManagerTest
Test result: ok. 6 passed; 0 failed; finished in 615.31ms

Running 16 tests for test/unit/products/revolving-credit-line/RCLLenders.deposit.t.sol:RCLLenderDepositTest
Test result: ok. 16 passed; 0 failed; finished in 627.24ms

Running 15 tests for test/unit/products/revolving-credit-line/RCLLenders.updateRate.t.sol:RCLLenderUpdateRateTest
Test result: ok. 15 passed; 0 failed; finished in 1.85s

Running 23 tests for test/unit/products/revolving-credit-line/RCLLenders.opt-out.t.sol:RCLLenderOptOutTest
Test result: ok. 23 passed; 0 failed; finished in 2.19s

Running 26 tests for test/unit/products/revolving-credit-line/RCLBorrowers.borrow.t.sol:RCLBorrowerBorrowTest
Test result: ok. 26 passed; 0 failed; finished in 3.12s
```

8.3 Code Coverage

```
> forge coverage
```

The relevant output is presented below. Please note that the low code coverage for `src/libraries` may be due to an issue with Foundry where internal libraries are not accurately tracked (See here)

Directory	Line Coverage		Functions	
src	**80.0 %**	12 / 15	**60.0 %**	3 / 5
src/common/custodian	**80.4 %**	74 / 92	**75.0 %**	18 / 24
src/common/custodian/adapters	**100.0 %**	28 / 28	**100.0 %**	13 / 13
src/common/fees	**100.0 %**	41 / 41	**100.0 %**	11 / 11
src/common/rewards	**92.2 %**	83 / 90	**83.3 %**	20 / 24
src/common/rewards/modules	**100.0 %**	170 / 170	**83.3 %**	30 / 36
src/common/roles-manager	**77.8 %**	14 / 18	**81.8 %**	9 / 11
src/libraries	**38.5 %**	15 / 39	**66.7 %**	10 / 15
src/products/RevolvingCreditLines	**93.2 %**	55 / 59	**90.9 %**	10 / 11
src/products/RevolvingCreditLines/libraries	**99.5 %**	410 / 412	**66.0 %**	35 / 53
src/products/RevolvingCreditLines/modules	**99.4 %**	164 / 165	**97.1 %**	33 / 34

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.

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.