Security Audit Report

Atlendis Protocol

Delivered: February 18th, 2022

Prepared for Atlendis Labs by

Summary

Runtime Verification, Inc. conducted a security audit on the Atlendis Protocol contracts. The audit was conducted from January 10th, 2022, to February 18th, 2022.

Several issues have been identified as follows:

- Implementation flaws: A1, A2, A3, A5, A6, A9, B5
- Potential security vulnerabilities: A4, A6, A7, A8, B04, B09

A number of additional suggestions have also been made, including:

- Input validations: A08, B05, B08
- Best practices: A01, A07, B09, B10
- Gas optimization: B07, B08
- Code readability: B01, B02

All the critical security issues have been addressed by Atlendis Labs. Details can be found in the Findings section as well as the Informative Findings section.

The code is generally well written and thoughtfully designed, following best practices.

Scope

The targets of the audit are the smart contract source files at git-commit-id `c6027d0b145d8ff3cffb2382acc9d3661f4cd9c3`.

The audit focused on the following core contracts and interfaces:

- BorrowerPools.sol
- PoolsSettingsManager.sol
- Position.sol
- lib/Errors.sol
- lib/PoolLogic.sol
- lib/Rounding.sol
- lib/Types.sol
- lib/WadRayMath.sol
- interfaces/IBorrowerManagement.sol
- interfaces/IBorrowerPools.sol
- interfaces/IPlatformManagement.sol
- interfaces/IPoolsParametersManagement.sol

- interfaces/IPosition.sol

The audit is limited in scope within the boundary of the Solidity contract only. Off-chain and client-side portions of the codebase, as well as deployment and upgrade scripts are *not* in the scope of this engagement but are assumed to be correct (see below).

Assumptions

The audit is based on the following assumptions and trust model.

- External token contracts conform to the ERC20 standard, especially that they *must* revert in failures. They do *not* allow any callbacks (e.g., ERC721 or ERC777) or any external contract calls. They do *not* implicitly update the balance of accounts (e.g., rebasing tokens or fees on transfers) other than explicit events of transfers, mints, or burns.
- For the authorizable contracts, their owners and authorized users are trusted to behave correctly and honestly, as described in the Security Model section.

Methodology

Although the manual code review cannot guarantee to find all possible security vulnerabilities as mentioned in the Disclaimer, we have followed the following approaches to make our audit as thorough as possible. First, we rigorously reasoned about the business logic of the contract, validating security-critical properties to ensure the absence of loopholes in the business logic and/or inconsistency between the logic and the implementation. To this end, we developed a high-level model of the central logical parts. Then, we carefully checked if the code is vulnerable to known security issues and attack vectors. Finally, we formally verified some properties of the arithmetic library with the Coq proof assistant.

Disclaimer

This report does not constitute legal or investment advice. The preparers of this report present it as an informational exercise documenting the due diligence involved in the secure development of the target contract only and make no material claims or guarantees concerning the contract's operation post-deployment. The preparers of this report assume no liability for any and all potential consequences of the deployment or use of this contract.

Smart contracts are still a nascent software arena, and their deployment and public offering carries substantial risk. This report makes no claims that its analysis is fully comprehensive and recommends always seeking multiple opinions and audits.

This report is also not comprehensive in scope, excluding a number of components critical to the correct operation of this system.

The possibility of human error in the manual review process is very real, and we recommend seeking multiple independent opinions on any claims which impact a large quantity of funds.

Security Model

The contracts are operated by multiple parties, and the secure operation of the contracts requires all the participating agents to behave correctly, honestly, and diligently.

Governance

The governance responsibility is to administer the contract. This includes granting and revoking borrowers access to their respective pools, creating pools, updating their parameters if needed, and freezing and unfreezing the protocol in case of malfunctioning. All these actions should be executed with extreme care since it’s effectively the assets of the lenders and borrowers at stake.

Borrowers

Borrowers are privileged accounts that can borrow assets from their respective pools. Borrowers are responsible for repaying the loans according to the terms and conditions of the pool parameters and off-chain agreements negotiated with Atlendis.

Lenders

Lenders are permissionless accounts that can provide liquidity to pools to earn interest on deposits. Lenders are responsible for evaluating the risk from interacting with the contract.

Token Integration Matrix

The Atlendis protocol contracts can, in general, interact with arbitrary tokens. However, due to the wide range of tokens, the features and implementations also vary, effectively rendering some tokens incompatible with the contract or exposing them to unexpected behaviors. The following table summarizes some basic properties and well-known tokens' limitations that Atlendis is willing to support. Details of the issues can be found in the findings section of the report by following the footnotes. Please note that this table is a best effort. The information was extracted by skimming through the tokens' source codes, and are not the result of an in-depth analysis.

Short	Decimals	Infinite Approval[1]	Fee[2]	Rebasing	Callbacks[3]
DAI	18	Yes	No	No	No
USDC	6	No	No	No	No
USDT	6	Yes	Yes	No	No
WETH	18	Yes	No	No	Yes
WBTC	8	No	No	No	No
SUSHI	18	Yes	No	No	No
COMP	18	Yes	No	No	No
BAL	18	No	No	No	No
UNI	18	Yes	No	No	No
MATIC	18	No	No	No	Yes
YFI	18	No	No	No	No
CRV	18	Yes (as of V2)	No	No	No
AAVE	18	No	No	No	Yes

[1] See Risk of infinite approvals
[2] See A06: Incompability with fee-charging tokens
[3] See A07: Re-entrancy vulnerability in Position.withdraw

Risk of infinite approvals

When a pool receives tokens via a deposit or repay action, the tokens are forwarded to a third-party yield provider, which is currently only Aave. This is a two-step process: First, Aave is approved to transfer the funds. Second, Aave is instructed to pull in the tokens. The audited revision of the contract executed both steps on every deposit and repay action. This is a gas intensive approach, therefore Atlendis considered changing the implementation and grant Aave an infinite approval during pool creation and then only execute the second step during deposits and repays to save gas costs.

In general infinite approvals have been a recurring source of security vulnerabilities. In the case of Atlendis, the potential attack surface is minimized because there are no “idle” tokens. All deposits are either actively loaned out to a borrower or transferred to the third-party yield provider.

Still, there are risks to consider. Specifically, not every token implements an infinite approval scheme, for example, USDC and WBTC. These tokens will decrement the approved amount on every `transferFrom` call. Consequently, the approved amount may eventually be decreased to 0. At this point, the Atlendis contract would not be able to execute deposit, repay, and topUpMaintenanceFees actions successfully. However, this risk of an approved amount decreasing from 2**256 - 1 to 0 is mainly considered theoretical and not practical. However, in the presence of flash-mintable tokens, the risk can become real because users can get temporary access to arbitrary amounts of tokens and gain the necessary resources to decrease the approved amounts effectively.

Recommendation

Stay with the current exact approval scheme to avoid the risks mentioned above.

Status

Atlendis Labs decided against switching to the infinite approval scheme and stays with the current exact approach.

Library Deployment

Solidity libraries can be integrated in two ways: First, they can be embedded. The Solidity compiler will then essentially prepend the compiled library code to the contracts using it. Second, libraries can be independently deployed to the network, and the Solidity compiler will delegate calls made from the contracts to the library.

The PoolLibrary of the audited revision was meant to be embedded. However, the size of the compiled contract exceeded the code-size limit of the Ethereum network forbidding a successful deployment. Consequently, Atlendis decided to deploy the library separately and have the contracts link to it as a workaround.

We gave this change a lightweight review and couldn’t find any security-related issues.

Findings

A01: Unsafe transfer of ERC20 tokens

[Severity: High | Difficulty: Low | Category: Implementation flaws, Best Practice]

Description

Some tokens, e.g., USDT, do not conform to the ERC20 standard. In particular, the `transfer` and `transferFrom` functions do not return boolean values. Consequently, all transfers of USDT tokens initiated by the contract will eventually revert because the surrounding success checks will fail.

Recommendation

Do not expect that transfer calls made to arbitrary ERC20 tokens will return boolean values. Consider using a library, e.g., OpenZeppelin's SafeTransferERC20, to work around the issue.

Status

Atlendis Labs addressed the issue.

A02: Discrepancy between getRepayAmount() and repay()

[Severity: Mid | Difficulty: Low | Category: Implementation flaws]

Description

There is a discrepancy between how `normalizedRepayAmount` is calculated in `BorrowerPools.getRepayAmount()` and `BorrowerPools.repay()`. Specifically, the `LATE_REPAY_FEE` and `PLATFORM_FEE` are added in different orders. Consequently, the computed amount will usually be higher in `repay()` than in `getRepayAmount()`.

Recommendation

We recommend factoring out this logic into a reusable function and having a single source of truth for the value. This also lowers the risk of introducing a mistake if the function needs to be changed in the future.

Status

Atlendis Labs addressed the issue.

A03: Unfair repayments due to maximum rate change

[Severity: High | Difficulty: High | Category: Implementation flaws]

Description

There is a possibility that lenders will not get paid back their total amount if the lender deposited into the highest tick and the max rate is changed during an ongoing loan.

See related: A05: Non-Borrowable deposits after rate changes

Scenario

1. Assume Bob is willing to borrow 100 tokens at a minimum rate of 1%, maximum rate of 10%, and a rate spacing of 1%.
2. Assume that 90 tokens have been deposited in 1%-9% ticks.
3. Alice is a lender willing to lend 10 tokens at 10%, so she deposits into the 10% tick. For simplicity, assume that no other tokens have been deposited into the 10% tick.
4. Bob borrows 100 tokes from his pool.
5. Governance changes Bob's maximum rate to 9%.
6. Bob wants to pay back his loan after 1 year during the usual repayment period.
    a. However, he will only need to pay back the tokens taken from the 1%-9% ticks (=90 tokens) plus interest.
    b. Alice is not paid back because the 10% tick is not considered in the `BorrowerPools.repay()` method because the new maximum rate is higher than her lending rate.

Recommendation

Notice that a similar change to the minimum rate will not lead to a similar corrupted situation because the `repay()`-method considers the lowest tick that contains deposits (`pool.state.lowerInterestRate`) and not the minimum rate (`pool.parameters.MIN_RATE`).

This observation suggests a solution for the maximum rate: Maintain a variable `pool.state.higherInterestRate` that keeps track of the highest tick that contains deposits. Set the upper bound of the loop in the repay method to this new variable instead of `pool.parameters.MAX_RATE`.

Status

Atlendis Labs addressed the issue by removing the possibility of updating the pool's minimum rate, maximum rate, and rate spacing parameters after its creation.

A04: Possible DoS due to unbounded loops

[Severity: High | Difficulty: High | Category: Security]

Description

`BorrowerPools.borrow()` and `BorrowerPools.repay()` suffer from a potential DoS vulnerability.

The `borrow()`-method contains a while loop that can potentially exceed the block gas limit.

Scenario

1. Assume Alice is a borrower. Her pool has a lower interest rate of 0%, maximum rate of 10%, rate spacing of 0.01%, the underlying asset token is wETH, and the maximum borrowable amount is 100 wETH.
2. Furthermore, assume that lenders have deposited 100 wETH only in the 5% tick for simplicity.
3. Bob is a malicious lender. He deposits tiny amounts of money (e.g., 1 Wei) into the lowest 100 ticks. Consequently, all ticks between 0% and 1% contain exactly 1 Wei.
4. When Alice wants to borrow 100 wETH, the borrowing algorithm will sell 100 bonds to Bob.
5. The while-loop of the `borrow()`-method would perform at least 100 iterations, potentially exceeding the block gas limit.
6. Alice transaction will revert.

Status

Atlendis Labs acknowledges the issue. No on-chain solution will be implemented. Instead, the involved parameters are manually evaluated off-chain before pool creation.

A05: Non-Borrowable deposits after rate changes

[Severity: Mid | Difficulty: High | Category: Implementation flaws]

Description

The governance can change some pool parameters, including the maximum rate of the pool. If lenders already deposited into high rate ticks, and the government reduces the maximum rate later, the deposits cannot be loaned out. Notice, however, that lenders can still withdraw from these ticks, so no deposits are permanently lost.

See related: A03: Unfair repayments due to maximum rate change

Scenario

1. Bob is borrower, with pools parameters minRate = 0%, maxRate = 10%, rate spacing = 1%.
2. Alice deposits 100 tokens in the 8% tick of the pool.
3. Governance changes the maxRate to 5%.
4. Alice deposits are no longer available for Bob to borrow.

Status

Atlendis Labs addressed the issue by removing the possibility of updating the pool's minimum rate, maximum rate, and rate spacing parameters after its creation.

A06: Incompatibility with fee-charging tokens

[Severity: High | Difficulty: Low | Category: Security, Implementation flaws]

Description

Some ERC20 tokens, e.g., USDT, can charge fees on token transfers. In this case, a portion of the transferred amount is instead transferred to the fee-receiver. Consequently, the receiver specified in the ERC20.transfer method will not receive the total transferred amount.

The `Position.deposit` and `BorrowerPools.repay` methods might revert depending on when fees are charged. Additionally, `Position.withdraw` and `BorrowerPools.borrow` might lead to unexpected transferred amounts.

Scenario A: Failed deposit

1. Consider a token that charges a fee of 5%.
2. Alice deposits 100 tokens into one of Atlendis' pools.
    a. The amount gets temporarily transferred to the Atlendis contract. 5 tokens are charged as a fee. 95 tokens are actually received.
    b. Atlendis attempts to forward the original amount of 100 tokens to the third-party yield provider.
    c. The transaction will revert because only 95 tokens are temporarily available.

Scenario B: borrow amount too low

1. Consider a token that can change fees and currently charges 0%.
2. Assume Bob is a borrower. For simplicity, assume that the pool is currently empty.
3. Alice deposits 100 tokens into Bob’s pool.
    a. The total amount of 100 tokens is temporarily transferred to Atlendis' contract and subsequently forwarded to the third-party yield provider.
4. The token service provider increases its fees to 5%.
5. Bob attempts to borrow 100 tokens.
    a. Atlendis instructs the third-party yield provider to transfer 100 tokens to Bob.
    b. 5 tokens are charged as a fee. Consequently, Bob receives only 95 tokens.

Scenario C: Failed repay

1. Consider a token that can change fees and currently charges 0%.
2. Assume Bob is a borrower. For simplicity, assume that the pool is currently empty.
3. Alice deposits 100 tokens into the 10% tick of Bob’s pool.

    a. Atlendis receives the total amount of 100 tokens and forwards them to the third-party yield provider.
4. Bob takes a loan of 100 tokens.
5. The token service provider increases its fee to 5%.
6. Bob wants to repay his loan + interest, i.e., 110 tokens.
    a. The amount gets temporarily transferred to the Atlendis contract. 5.5 tokens are charged as a fee. 104.5 tokens are actually received.
    b. Atlendis attempts to forward the original amount of 110 tokens to the third-party yield provider.
    c. The transaction will revert because only 104.5 tokens are available.

Scenario D: withdraw amount too low

1. Consider a token that can change fees and currently charges 0%.
2. Assume Bob is a borrower. For simplicity, assume that the pool is currently empty.
3. Alice deposits 100 tokens into the 10% tick of Bob's pool.
4. Atlendis receives the total amount of 100 tokens and forwards them to the third-party yield provider.
5. Bob takes a loan of 100 tokens.
6. Bob wants to repay his loan + interest, i.e,. 110 tokens.
7. The total amount is temporarily transferred to Atlendis' contract and subsequently forwarded to the third-party yield provider.
8. The token service provider increases its fee to 5%.
9. Alice wants to withdraw:
    a. Atlendis instructs the third-party yield provider to transfer 110 tokens (deposit + interest) to Alice.
    b. 5.5 tokens are charged as a fee. Consequently, Alice receives only 104.5 tokens.

Recommendation

Check the actual token balance before and after each transfer, and change the logic of the contract to account for the exact transferred amounts.

Status

Acknowledged by Atlendis Labs. No on-chain solution will be implemented. Instead, no fee-charging tokens will be integrated.

A07: Re-entrancy vulnerability in Position.withdraw

[Severity: High | Difficulty: Low | Category: Security, Best Practice]

Description

Some tokens allow the senders and receivers of token transfers to execute arbitrary code, e.g., tokens implementing the ERC777 or ERC1363 interfaces. Attackers can exploit the callback functions to execute re-entrancy attacks.

The Position.withdraw() function is vulnerable to re-entrancy attacks and can be exploited by a malicious lender to withdraw deposits that are currently actively loaned out to a borrower.

Scenario

1. Assume Alice deposited 100 tokens to the 10% tick of Bob's pool.
    a. position.adjustedAmount = 100
    b. position.remainingBonds = 0
2. Bob takes a loan. 50% of the 10% tick are actively loaned out.
3. Alice attempts to withdraw her unused tokens:
    a. 50 unused tokens are transferred to Alice.
    b. Notice that Alice's position has not yet been updated.
    c. Alice re-enters into withdraw:
    d. Another 50 tokens are transferred to Alice.
    e. After the re-entrancy call returned, Alice's position is updated, the new state is:
    f. position.adjustedBalance = 50
    g. position.remainingBonds = 50
    h. After the original withdraw call returned, Alice's position is updated again:
    i. position.adjustedBalance = 0
    j. position.remaingBonds = 50

Recommendation

A. Follow checks-effects-interactions-Pattern.
B. Use an anti-reentrancy-library, e.g., from OpenZeppelin.

Status

Atlendis Labs addressed the issue by following the checks-effects-interactions-pattern.

A08: Missing sanity checks for borrowerHash

[Severity: High | Difficulty: High | Category: Security, Input validation]

Description

Pools are not directly identified by their borrower's addresses but by some borrower hash that is assigned to a borrower address by the governance. Consequently, to lookup the pool corresponding to a specific borrower address, two steps are performed: First, the borrower hash is looked up in mapping (address => borrowerHash). Second, the pool is looked up in a mapping (borrowerHash => Pool). If two borrowers are given the same hash, they will access the same pool.

Scenario

1. Assume Bob is a borrower. The governance assigned him borrowerHash = 0x01.
2. Further, assume that lenders have deposited 100 tokens into Bob's pool.
3. Alice is a new borrower. The governance assigns her the same borrowerHash 0x01. This might be accidental, due to a mistake made by the governance, or maliciously because the governance account has been compromised.
4. Alice can now borrow 100 tokens from Bob's pool.

Recommendation

Make sure, that borrwerHashes are unique among all borrowers. Additionally add a sanity check to PoolsSettingsManager.allow to forbid `borrowerHash = 0` and `borrowerAddress = address(0)`.

Status

Atlendis Labs addressed the issue.

A09: Borrowable amounts may be too high

[Severity: High | Difficulty: Low | Category: Implementation flaws]

Description

Under certain circumstances, it is possible that a borrower can borrow amounts that are too high, i.e., the borrowable amount is higher than the available deposits of the pool. The amounts are then effectively taken from other pools with the same underlying asset.

Scenario

1. Assume there is an ongoing loan. For simplicity, assume that Aave's liquidity ratio does not increase during the scenario. Also, for simplicity, assume that all deposits explicitly mentioned in this scenario are made to the 0% tick. The `jellyfiLiquidityRatio` and `yieldProviderLiquidityRatio` are currently 10. There are currently no pending amounts in the tick.
2. Alice deposits 100 tokens to the tick.
    a. `tick.adjustedPendingAmount = normalizedAmount / tick.yieldProviderLiquidityRaio = 100 / 10 = 10.`
3. The ongoing loan is repaid. For simplicity, assume that all deposits but Alice's are withdrawn.
    a. The `jellyfiLiquidtioRatio` becomes 20.
    b. `bondIssuanceMultiplier = yieldProviderLiquidityRatio / jellyfiLiquidityRatio = 10 / 20 = 0.5.`
    c. `tick.totalAmount = tick.pendingAmount * bondIssuaceMultiplier = 10 * 0.5 = 5`
4. Bob deposits 100 tokens to the tick.
    a. `tick.totalAmount = tick.totalAmount + (normalizedAmount / jellyFiLiquidityRaio) / bondIssuanceMultiplier = 5 + (100 / 20) / 0.5 = 15`
5. Borrower attempts to borrow 300 tokens.
    a. `normalizedUsedAmount = tick.adjustedRemainingAmount * jellyFiLiquidityRaio = 15 * 20 = 300`

Recommendation

Change the order-book-keeping logic to account for the correct amounts of deposits. In particular, a deposit adjusted amount should not depend on earlier deposits and repayments.

Status

Atlendis Labs addressed the issue.

Informative Findings

B01: Duplicate function getTickBondPrice()

[Severity: Low | Difficulty: - | Category: Code readability]

Description

The PoolLogic library and BorrowerPools contract implement the same getTickBondPrice() function.

The library is meant to be embedded directly and not separately deployed and linked against. Then PoolLibrary.getTickBondPrice should be declared internal. The function won't be externally callable either way (notice that the contract version is externally callable). Moreover, that's currently the only non-internal function of the library, and declaring it internal would clarify that the library is intended to be embedded.

Having two implementations of the same functions increases the risk of mistakes if the functions need to be changed in the future.

Recommendation

We suggest that the contract calls the library function internally. Another benefit is getting rid of the magic constants in die contract implementation. It's also possible declaring the contract version external instead of public since it is never used internally.

Status

Atlendis Labs addressed the issue by removing the function from the BorrowerPools contract.

B02: Simplify expressions

[Severity: Low | Difficulty: - | Category: Code readability]

Description

The following expression in `PoolLogic.includePendingDepositsForTick` can be simplified to enhance readability and save some gas costs.

```
RAY.rayMul(tick.yieldProviderLiquidityRatio)
   .rayDiv(tick.jellyFiLiquidityRatio)
```

The simplified variant:

```
tick.yieldProviderLiquidityRatio.rayDiv(tick.jellyFiLiquidityRatio)
```

Status

Atlendis Labs addressed the issue.

B03: Add a timelock to deposits

[Severity: Low | Difficulty: - | Category: Security]

Description

It should never be profitable for a lender to deposit and withdraw within the same transaction. Any profits would stem from some miscalculation, e.g., from rounding errors. Although we didn’t find a concrete scenario to exploit the contract this way, we suggest adding a timelock to deposits to forbid withdrawals during the same transaction as a defensive programming technique.

Status

Atlendis Labs addressed the issue.

B04: Invariant breaks of rate spacings

[Severity: Mid | Difficulty: High | Category: Implementation flaws, Input Validation]

Description

The governance can change the minimum and maximum rates of a pool after creation. Setting the minimum rate is only possible when the following conditions are fulfilled:

```
minRate < pools[borrower].parameters.MAX_RATE)
minRate % pools[borrower].parameters.RATE_SPACING == 0
```

Similarly, to update the maxim rate the following conditions must be satisfied:

```
maxRate > pools[borrower].parameters.MIN_RATE
maxRate % pools[borrower].parameters.RATE_SPACING == 0
```

These sanity checks prevent dysfunctional pools that do not accept deposits. For reference, here are the checks that are performed during deposits:

```
rate >= pool.parameters.MIN_RATE
rate <= pool.parameters.MAX_RATE
rate % pool.parameters.RATE_SPACING == 0
```

However, the sanity checks of the setters are not performed during pool creation. Hence it is possible to create pools that cannot accept any deposits.

See related: B06 Unnecessary strong invariants

Recommendation

Add the sanity checks from the setters to `PoolsSettingsManager.createNewPool` function.

Status

Atlendis Labs addressed the issue.

B05: Unnecessary strong invariants

[Severity: Low | Difficulty: - | Category: Design]

Description

The PoolsSettingsManager maintains the following invariants of the minimum rate, maximum rate and rate spacing parameters:

```
minRate % pools[borrower].parameters.RATE_SPACING == 0
maxRate % pools[borrower].parameters.RATE_SPACING == 0
```

These invariants intend to prohibit ticks that cannot be deposited into. However, the invariants seem unnecessary strong in that they prohibit some feasible pool parameter combinations, e.g., `minRate = 1, maxRate = 3, rateSpacing = 2` and `minRate = 2, maxRate = 8, rateSpacing = 3`

See related: B05: Invariant breaks of rate spacings

Recommendation

The invariants cannot be replaced by the weaker invariant:

```
(maxRate - minRate) % pools[borrower].parameters.RATE_SPACING == 0
```

The latter invariant is less restrictive in that it permits the pool parameter combinations from above. Notice that changing the clauses would also affect other parts of the code, for example, `BorrowerPools.deposit()` and `BorrowerPools.updateRate()`.

Status

Atlendis Labs addressed the issue.

B06: Move tick-fields to the pool state to save gas costs

[Severity: Low | Difficulty: - | Category: Gas optimization]

Description

The following two fields of the Tick-struct can be moved to the PoolState-struct.

This would save a considerable amount of storage (hence gas costs) depending on the rate spacing and bounds of the pool.

`Tick.currentBondsIssuanceIndex`

- This value should be equal among all ticks, considering the min and max rates are never changed.
- Even if min or max rates are changed, there is no conflict as long as the index strictly increases during repay.

`Tick.yieldProviderLiquidityRatio`

- The value only depends on the yield provider, underlying token, and the time requested from the yield provider.
- The value does not depend on the ratio or any other tick fields.
- If moved to the pool level, it could be set during pool initialization (instead of tick initialization) and during fee collection (as before).
- Having this variable globally per pool would also reduce the number of external calls made to the yield provider.

Status

Atlendis Labs addressed the issue.

B07: Add sanity checks to Position.deposit

[Severity: Low | Difficulty: - | Category: Usability]

Description

Deposits of 0 tokens are not useful and should be considered a mistake by the lender.

Add a sanity check to `Position.deposit()` and revert if the normalized deposited amount is `0`.

Status

Atlendis Labs addressed the issue.

B08: WAD and RAY math rounding errors

[Severity: Low | Difficulty: - | Category: Security, Best practice]

Description

The contracts make heavy use of an arithmetic library developed by Aave. The library simulates fixed-point arithmetic on the Solidity level. Specifically, the library enables simulating arithmetic with 18 (Wad) and 27 (Ray) digit precision values. When expressions involve quantities of different precision, it is recommended to convert between them explicitly. While we didn't identify any severe rounding error that stems from an omitted conversion, we still encourage Atlendis Labs to follow the library's instructions and make conversions explicit.

Of course, explicit conversions introduce an overhead in terms of gas costs. In some cases, however, it is possible to simplify the conversion by replacing it with a less gas intensive but semantically equivalent expression. In particular, we proved that the following expressions are equal:

```
a.wadToRay().rayMul(b).rayToWad() = a.wadMul(b).rayToWad()
```

The term on the left-hand side is arguably the most expressive and readable version, but the term on the right-hand side consumes less gas. The proof of the equivalence is given in Appendix A.

Status

Atlendis Labs addressed the issue.

B09: Risk of freezing the contract

[Severity: - | Difficulty: - | Category: -]

Description

The protocol supports an emergency feature that allows the governance account to freeze the protocol in case of unforeseen malfunctions. When the protocol is frozen, most functions will no longer be callable until the governance unfreezes the contract again. However, some functions are time-critical. For example, a borrower needs to repay an ongoing loan in the pre-specified period. If he fails to repay on time, he will get charged a penalty fee. If the borrower cannot refund because the protocol is frozen, he would still need to pay the penalty, despite not influencing the circumstances that led to the unfortunate scenario.

Status

Atlendis Labs acknowledges the issue. No on-chain solution will be implemented. The governance account will be multi-signature account.

B10: Missing intializer-calls

[Severity: Low | Difficulty: - | Category: Best Practice]

Description

Some contracts, e.g., `PoolsSettingsManager` inherit from third-party library contracts, e.g., `AccessControlUpgradeable` and `PausableUpgradeable`. These library contracts come with their own initialization-methods. However, the initializers are currently not executed by the child-contracts.

Status

Atlendis Labs addressed the issue.

Appendix A - Coq Model of Wad-Ray-Math

```
Set Warnings "-notation-overridden, -ambiguous-paths".
From mathcomp Require Import all_ssreflect.

Notation "n `exp m" := (n * 10 ^ m) (at level 40, format "n `exp m").

Lemma mulnexpA (n m p : nat) : n * (m`expp) = (n * m)`expp.
Proof. by rewrite mulnA. Qed.

Lemma muln1exp_add (n m : nat) : 1`expn * (1`expm) = 1`exp(n + m).
Proof. by rewrite !mul1n -expnD. Qed.

Lemma muln1expL (n m p : nat) : n`expm * (1`expp) = n`exp(m + p).
Proof.
  by rewrite -{1}(muln1 n) -mulnexpA -mulnA muln1exp_add mulnA muln1.
Qed.

Lemma expS (n m : nat) : n`expm.+2 = n`expm.+1 * 10.
Proof. by rewrite expnSr mulnA. Qed.

Lemma half_exp (n : nat) : 1`expn.+1 %/ 2 = 5`expn.
Proof.
  by elim: n=> [//|n IH]; rewrite expS -muln_divA// mulnC mulnA.
Qed.

Definition WAD := 1`exp18.
Definition RAY := 1`exp27.
Definition WAD_RAY_RATIO := 1`exp9.
Definition halfWAD := WAD %/ 2.
Definition halfRAY := RAY %/ 2.

Lemma rayE : RAY = WAD * WAD_RAY_RATIO.
Proof. by rewrite muln1exp_add. Qed.

Lemma halfRayE : halfRAY = halfWAD * WAD_RAY_RATIO.
Proof. by rewrite /halfRAY /halfWAD !half_exp muln1expL. Qed.

Definition wadMul (a b : nat) := (a * b + halfWAD) %/ WAD.
Definition rayMul (a b : nat) := (a * b + halfRAY) %/ RAY.

Definition rayDiv (a b : nat) := (a * RAY + b %/ 2) %/ b.

(* Notice that no check for addition overflow is performed *)
Definition rayToWad (a : nat) :=
  let halfRatio := WAD_RAY_RATIO %/ 2 in
  let result := halfRatio + a in
  result %/ WAD_RAY_RATIO.

Definition wadToRay (a : nat) := a * WAD_RAY_RATIO.

Lemma gas_opt (a b : nat) :
  rayToWad (rayMul (wadToRay a) b) = rayToWad (wadMul a b).
Proof.
  rewrite /rayMul /wadMul /wadToRay halfRayE rayE.
  by rewrite {1}mulnAC -mulnDl divnMr; first by reflexivity.
Qed.
```