SMART CONTRACT AUDIT REPORT

for

Atlendis Protocol

Prepared By: Xiaomi Huang

PeckShield

May 21, 2022

Document Properties

Client	Atlendis
Title	Smart Contract Audit Report
Target	Atlendis
Version	1.0
Author	Xuxian Jiang
Auditors	Luck Hu, Xuxian Jiang
Reviewed by	Xiaomi Huang
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	May 21, 2022	Xuxian Jiang	Final Release
1.0-rc	May 18, 2022	Xuxian Jiang	Release Candidate

Contact

For more information about this document and its contents, please contact PeckShield Inc.

Name	Xiaomi Huang
Phone	+86 183 5897 7782
Email	contact@peckshield.com

Contents

1 Introduction

Given the opportunity to review the design document and related smart contract source code of the `Atlendis` protocol, we outline in this report our systematic approach to evaluate potential security issues in the smart contract implementation, expose possible semantic inconsistencies between smart contract code and design document, and provide additional suggestions or recommendations for improvement. Our results show that the given version of smart contract can be further improved due to the presence of several issues. This document outlines our audit results.

1.1 About Atlendis

`Atlendis` is a capital-efficient `DeFi` lending protocol where organizations evolving in the crypto space such as protocols, `DEXes` or `DAOs` can get access to uncollateralized lines of credit. `Atlendis` is targeting entities with regular and short term liquidity needs. Similarly to a revolving line of credit, the protocol allows entities to borrow or issue bonds as many times as they need, up to a preset borrowing limit, without any collateral. `Atlendis`' pools are borrower specific and the borrowing rates are discovered via a limit order book specific to each pool. The basic information of the audited protocol is as follows:

Table 1.1: Basic Information of Atlendis

Item	Description
Name	Atlendis
Website	https://atlendis.io/
Type	EVM Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	May 21, 2022

In the following, we show the Git repository of reviewed files and the commit hash value used in this audit.

- https://github.com/Atlendis/priv-contracts.git (dd0c237)

And here is the commit ID after all fixes for the issues found in the audit have been checked in:

- https://github.com/Atlendis/priv-contracts.git (1843085)

1.2 About PeckShield

PeckShield Inc. [9] is a leading blockchain security company with the goal of elevating the security, privacy, and usability of current blockchain ecosystems by offering top-notch, industry-leading services and products (including the service of smart contract auditing). We are reachable at Telegram (https://t.me/peckshield), Twitter (http://twitter.com/peckshield), or Email (contact@peckshield.com).

Table 1.2: Vulnerability Severity Classification

Impact \ Likelihood	*High*	*Medium*	*Low*
High	Critical	High	Medium
Medium	High	Medium	Low
Low	Medium	Low	Low

1.3 Methodology

To standardize the evaluation, we define the following terminology based on OWASP Risk Rating Methodology [8]:

- Likelihood represents how likely a particular vulnerability is to be uncovered and exploited in the wild;
- Impact measures the technical loss and business damage of a successful attack;
- Severity demonstrates the overall criticality of the risk.

Likelihood and impact are categorized into three ratings: *H*, *M* and *L*, i.e., *high*, *medium* and *low* respectively. Severity is determined by likelihood and impact and can be classified into four categories accordingly, i.e., *Critical*, *High*, *Medium*, *Low* shown in Table 1.2.

Table 1.3: The Full List of Check Items

Category	Check Item
Basic Coding Bugs	Constructor Mismatch
	Ownership Takeover
	Redundant Fallback Function
	Overflows & Underflows
	Reentrancy
	Money-Giving Bug
	Blackhole
	Unauthorized Self-Destruct
	Revert DoS
	Unchecked External Call
	Gasless Send
	Send Instead Of Transfer
	Costly Loop
	(Unsafe) Use Of Untrusted Libraries
	(Unsafe) Use Of Predictable Variables
	Transaction Ordering Dependence
	Deprecated Uses
Semantic Consistency Checks	Semantic Consistency Checks
Advanced DeFi Scrutiny	Business Logics Review
	Functionality Checks
	Authentication Management
	Access Control & Authorization
	Oracle Security
	Digital Asset Escrow
	Kill-Switch Mechanism
	Operation Trails & Event Generation
	ERC20 Idiosyncrasies Handling
	Frontend-Contract Integration
	Deployment Consistency
	Holistic Risk Management
Additional Recommendations	Avoiding Use of Variadic Byte Array
	Using Fixed Compiler Version
	Making Visibility Level Explicit
	Making Type Inference Explicit
	Adhering To Function Declaration Strictly
	Following Other Best Practices

To evaluate the risk, we go through a list of check items and each would be labeled with a severity category. For one check item, if our tool or analysis does not identify any issue, the contract is considered safe regarding the check item. For any discovered issue, we might further deploy contracts on our private testnet and run tests to confirm the findings. If necessary, we would additionally build a PoC to demonstrate the possibility of exploitation. The concrete list of check items is shown in Table 1.3.

In particular, we perform the audit according to the following procedure:

- Basic Coding Bugs: We first statically analyze given smart contracts with our proprietary static code analyzer for known coding bugs, and then manually verify (reject or confirm) all the issues found by our tool.

- Semantic Consistency Checks: We then manually check the logic of implemented smart contracts and compare with the description in the white paper.

- Advanced DeFi Scrutiny: We further review business logics, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.

- Additional Recommendations: We also provide additional suggestions regarding the coding and development of smart contracts from the perspective of proven programming practices.

To better describe each issue we identified, we categorize the findings with Common Weakness Enumeration (CWE-699) [7], which is a community-developed list of software weakness types to better delineate and organize weaknesses around concepts frequently encountered in software development. Though some categories used in CWE-699 may not be relevant in smart contracts, we use the CWE categories in Table 1.4 to classify our findings.

1.4 Disclaimer

Note that this security audit is not designed to replace functional tests required before any software release, and does not give any warranties on finding all possible security issues of the given smart contract(s) or blockchain software, i.e., the evaluation result does not guarantee the nonexistence of any further findings of security issues. As one audit-based assessment cannot be considered comprehensive, we always recommend proceeding with several independent audits and a public bug bounty program to ensure the security of smart contract(s). Last but not least, this security audit should not be used as investment advice.

Table 1.4: Common Weakness Enumeration (CWE) Classifications Used in This Audit

Category	Summary
Configuration	Weaknesses in this category are typically introduced during the configuration of the software.
Data Processing Issues	Weaknesses in this category are typically found in functionality that processes data.
Numeric Errors	Weaknesses in this category are related to improper calculation or conversion of numbers.
Security Features	Weaknesses in this category are concerned with topics like authentication, access control, confidentiality, cryptography, and privilege management. (Software security is not security software.)
Time and State	Weaknesses in this category are related to the improper management of time and state in an environment that supports simultaneous or near-simultaneous computation by multiple systems, processes, or threads.
Error Conditions, Return Values, Status Codes	Weaknesses in this category include weaknesses that occur if a function does not generate the correct return/status code, or if the application does not handle all possible return/status codes that could be generated by a function.
Resource Management	Weaknesses in this category are related to improper management of system resources.
Behavioral Issues	Weaknesses in this category are related to unexpected behaviors from code that an application uses.
Business Logic	Weaknesses in this category identify some of the underlying problems that commonly allow attackers to manipulate the business logic of an application. Errors in business logic can be devastating to an entire application.
Initialization and Cleanup	Weaknesses in this category occur in behaviors that are used for initialization and breakdown.
Arguments and Parameters	Weaknesses in this category are related to improper use of arguments or parameters within function calls.
Expression Issues	Weaknesses in this category are related to incorrectly written expressions within code.
Coding Practices	Weaknesses in this category are related to coding practices that are deemed unsafe and increase the chances that an exploitable vulnerability will be present in the application. They may not directly introduce a vulnerability, but indicate the product has not been carefully developed or maintained.

2 | Findings

2.1 Summary

Here is a summary of our findings after analyzing the implementation of the `Atlendis` protocol. During the first phase of our audit, we study the smart contract source code and run our in-house static code analyzer through the codebase. The purpose here is to statically identify known coding bugs, and then manually verify (reject or confirm) issues reported by our tool. We further manually review business logics, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.

Severity	# of Findings	
Critical	0	
High	0	
Medium	3	■■■
Low	3	■■■
Informational	0	
Total	6	

We have so far identified a list of potential issues: some of them involve subtle corner cases that might not be previously thought of, while others refer to unusual interactions among multiple contracts. For each uncovered issue, we have therefore developed test cases for reasoning, reproduction, and/or verification. After further analysis and internal discussion, we determined a few issues of varying severities need to be brought up and paid more attention to, which are categorized in the above table. More information can be found in the next subsection, and the detailed discussions of each of them are in Section 3.

2.2 Key Findings

Overall, these smart contracts are well-designed and engineered, though the implementation can be improved by resolving the identified issues (shown in Table 2.1), including 3 medium-severity vulnerabilities and 3 low-severity vulnerabilities.

Table 2.1: Key Atlendis Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Low	Proper Initialization Logic in BorrowerPools	Security Features	Fixed
PVE-002	Low	Proper Role Management in BorrowerPools	Coding Practices	Fixed
PVE-003	Medium	Incorrect Amount Of Rewards Used in closePool()	Business Logic	Fixed
PVE-004	Low	Improved Validation Of Function Arguments	Business Logic	Fixed
PVE-005	Medium	Trust Issue Of Admin Keys	Security Features	Confirmed
PVE-006	Medium	Improved yieldProviderLiquidityRatio Update in BorrowerPools	Business Logic	Fixed

Beside the identified issues, we emphasize that for any user-facing applications and services, it is always important to develop necessary risk-control mechanisms and make contingency plans, which may need to be exercised before the mainnet deployment. The risk-control mechanisms should kick in at the very moment when the contracts are being deployed on mainnet. Please refer to Section 3 for details.

3 | Detailed Results

3.1 Proper Initialization Logic in BorrowerPools

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: BorrowerPools
- Category: Security Features [4]
- CWE subcategory: CWE-287 [2]

Description

The Atlendis contract allows for lazy contract initialization, i.e., the initialization does not need to be performed inside the constructor at deployment. This feature is enabled by introducing the initializer() and onlyInitializing() modifiers. The initializer() protects an initializer function from being invoked twice, and the onlyInitializing() modifier protects an initialization function so that it can only be invoked by functions with the initializer() modifier, directly or indirectly. While examining the usage of these two modifiers, we notice the existence of abuse of the initializer() modifier, which needs to be corrected.

To elaborate, we show below the code snippet of the BorrowerPools::initialize() routine. As the name indicates, it is an initialization function for the BorrowerPools contract. This initialize() function is protected by the initializer() modifier and it further invokes the subcalls to _initialize (), etc. (line 25). It comes to our attention that the _initialize() sub-call is marked as public. To correct, we suggest to protect the subcalls with the internal onlyInitializing() modifiers, as recommended by Openzeppelin: #3006.

```
24  function initialize(ILendingPool _aaveLendingPool, address governance) public
        initializer {
25    _initialize();
26    yieldProvider = _aaveLendingPool;
27    if (governance == address(0)) {
28      // Prevent setting governance to null account
29      governance = _msgSender();
30    }
```

```
    _grantRole(DEFAULT_ADMIN_ROLE, governance);
    _grantRole(GOVERNANCE_ROLE, governance);
    _setRoleAdmin(BORROWER_ROLE, GOVERNANCE_ROLE);
  }
```

Listing 3.1: BorrowerPools::initialize()

```
  function _initialize() public {
    // both initializers below are called to comply with OpenZeppelin's
    // recommendations even if in practice they don't do anything
    __AccessControl_init();
    __Pausable_init_unchained();
  }
```

Listing 3.2: PoolsController::_initialize()

Recommendation Enforce the initialization-related subcalls with the internal onlyInitializing modifier.

Status This issue has been fixed by this commit: db7f5e1.

3.2 Proper Role Management in BorrowerPools

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: BorrowerPools
- Category: Coding Practices [5]
- CWE subcategory: CWE-1126 [1]

Description

The Atlendis protocol provides uncollateralized loans to yield higher interest rates than existing collateralized lending protocols. The protocol defines a number of different roles: BORROWER_ROLE, POSITION_ROLE, GOVERNANCE_ROLE, and DEFAULT_ADMIN_ROLE. As their names indicate, the first role allows for the holder to borrow from the pool; the second role is capable of providing position management for liquidity providers; the third one performs the governance role; and the last one manages the above three roles.

To elaborate, we show below the related initialize() function from the BorrowerPools contract. It has properly configured the given governance to have the GOVERNANCE_ROLE and DEFAULT_ADMIN_ROLE. It comes to our attention that the explicit role admin for the POSITION_ROLE is not given. To avoid unnecessary confusion and improve readability and maintenance, there is a need to explicitly grant the role admin of POSITION_ROLE to GOVERNANCE_ROLE as well!

```
function initialize(ILendingPool _aaveLendingPool, address governance) public
    initializer {
  _initialize();
  yieldProvider = _aaveLendingPool;
  if (governance == address(0)) {
    // Prevent setting governance to null account
    governance = _msgSender();
  }
  _grantRole(DEFAULT_ADMIN_ROLE, governance);
  _grantRole(GOVERNANCE_ROLE, governance);
  _setRoleAdmin(BORROWER_ROLE, GOVERNANCE_ROLE);
}
```

Listing 3.3: `BorrowerPools::initialize()`

Recommendation Revise the above-mentioned `initialize()` routine to properly set up the roles.

Status This issue has been fixed by this commit: `db7f5e1`.

3.3 Incorrect Amount Of Rewards Used in closePool()

- ID: PVE-003
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `PoolsController`
- Category: Business Logic [6]
- CWE subcategory: CWE-841 [3]

Description

Each pool in the `Atlendis` protocol follows the defined lifecycle with a number of states: `active`, `defaulted`, and `closed`. While reviewing the current logic to close an active pool, we notice the current implementation needs to be improved.

To elaborate, we show below the `closePool()` routine, which is used to close a current loan. While it properly validates the given loan indicated by its `poolHash`, it does not compute the right token amount to withdraw from the underlying yield provider. In particular, to facilitate the computation, the protocol normalizes the amount denominated at the 18 decimals, the final withdrawal amount from the underlying yield provider needs to convert back to have the token's decimals. In other words, the current `remainingNormalizedLiquidityRewardsReserve` (line 368) needs to be in the following form: `remainingNormalizedLiquidityRewardsReserve.scaleFromWad(pool.parameters.TOKEN_DECIMALS)`!

```
340 function closePool(bytes32 poolHash, address to) external override onlyRole(
        GOVERNANCE_ROLE) {
341   if (poolHash == bytes32(0)) {
```

```
      revert Errors.PC_ZERO_POOL();
    }
    if (to == address(0)) {
      revert Errors.PC_ZERO_ADDRESS();
    }
    Types.Pool storage pool = pools[poolHash];
    if (pool.parameters.POOL_HASH != poolHash) {
      revert Errors.PC_POOL_NOT_ACTIVE();
    }
    if (pool.state.closed) {
      revert Errors.PC_POOL_ALREADY_CLOSED();
    }
    pool.state.closed = true;

    uint128 remainingNormalizedLiquidityRewardsReserve = 0;
    if (pool.state.remainingAdjustedLiquidityRewardsReserve > 0) {
      uint128 yieldProviderLiquidityRatio = uint128(
        pool.parameters.YIELD_PROVIDER.getReserveNormalizedIncome(address(pool.
            parameters.UNDERLYING_TOKEN))
      );
      remainingNormalizedLiquidityRewardsReserve = pool.state.
          remainingAdjustedLiquidityRewardsReserve.wadRayMul(
        yieldProviderLiquidityRatio
      );

      pool.state.remainingAdjustedLiquidityRewardsReserve = 0;
      yieldProvider.withdraw(
        pools[poolHash].parameters.UNDERLYING_TOKEN,
        remainingNormalizedLiquidityRewardsReserve,
        to
      );
    }
    emit PoolClosed(poolHash, remainingNormalizedLiquidityRewardsReserve);
  }
```

Listing 3.4: `PoolsController::closePool()`

Recommendation Revise the above `closePool()` logic to compute the right amount to withdraw from the underlying yield provider.

Status This issue has been fixed by this commit: `db7f5e1`.

3.4 Improved Validation on Function Arguments

- ID: PVE-004
- Severity: Low
- Likelihood: Low
- Impact: High
- Target: `PoolsController`
- Category: Coding Practices [5]
- CWE subcategory: CWE-1126 [1]

Description

The `Atlendis` protocol has the `PoolsController` contract that is developed to facilitate the pool management and configuration. Notice that the pools in `Atlendis` are borrower specific and the borrowing rates are discovered via a limit order book specific to each pool. While examining the logic for pool management, we notice the borrower-specific pool management is not enforced.

To elaborate, we show below the related `allow()` function. As the name indicates, this function is proposed to allow the borrower to borrow from a specific `pool`. However, it comes to our attention that this routine does not enforce the given pool is specific to a borrower! If a pool is accidentally assigned to multiple borrowers, it may bring unexpected consequence as the internal pool accounting does not differentiate different borrowers.

```
function allow(address borrowerAddress, bytes32 poolHash) external override onlyRole(
    GOVERNANCE_ROLE) {
  if (poolHash == bytes32(0)) {
    revert Errors.PC_ZERO_POOL();
  }
  if (borrowerAddress == address(0)) {
    revert Errors.PC_ZERO_ADDRESS();
  }
  if (pools[poolHash].parameters.POOL_HASH != poolHash) {
    revert Errors.PC_POOL_NOT_ACTIVE();
  }
  grantRole(BORROWER_ROLE, borrowerAddress);
  borrowerAuthorizedPools[borrowerAddress] = poolHash;
  emit BorrowerAllowed(borrowerAddress, poolHash);
}
```

Listing 3.5: `PoolsController::allow()`

```
  function verifyPoolCreationParameters(PoolCreationParams calldata params) internal
      view {
  if ((params.maxRate - params.minRate) % params.rateSpacing != 0) {
    revert Errors.PC_RATE_SPACING_COMPLIANCE();
  }
  if (params.poolHash == bytes32(0)) {
    revert Errors.PC_ZERO_POOL();
  }
```

```
    if (pools[params.poolHash].parameters.POOL_HASH != bytes32(0)) {
      revert Errors.PC_POOL_ALREADY_SET_FOR_BORROWER();
    }
    DataTypes.ReserveData memory reserveData = yieldProvider.getReserveData(params.
        underlyingToken);
    if (reserveData.aTokenAddress == address(0)) {
      revert Errors.PC_POOL_TOKEN_NOT_SUPPORTED();
    }
    if (params.establishmentFeeRate > 1e18) {
      revert Errors.PC_ESTABLISHMENT_FEES_TOO_HIGH();
    }
  }
```

Listing 3.6: `PoolsController::verifyPoolCreationParameters()`

Similarly, the `verifyPoolCreationParameters()` function in the same contract can be improved to ensure the related `liquidity ratio` from the underlying yield provider is expected.

Recommendation Properly enforce the design invariant for the pool management so that the pool is indeed borrower-specific.

Status The issue in the `allow()` routine has been confirmed by the team. And the team clarifies that this is intended feature. When we refer to pools as single borrower, we are talking about the legal entity aspect. The issue in the `verifyPoolCreationParameters()` routine has been fixed by this commit: `2ccf8af`.

3.5 Trust Issue of Admin Keys

- ID: PVE-005
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `PoolsController`
- Category: Security Features [4]
- CWE subcategory: CWE-287 [2]

Description

In the `Atlendis` protocol, there are privileged accounts (with various roles, e.g., `GOVERNANCE_ROLE` and `DEFAULT_ADMIN_ROLE`) that play a critical role in governing and regulating the system-wide operations (e.g., whitelist borrowers and configure various parameters). In the following, we examine the privileged account and their related privileged accesses in current contracts.

```
  function allow(address borrowerAddress, bytes32 poolHash) external override onlyRole(
      GOVERNANCE_ROLE) {
    if (poolHash == bytes32(0)) {
      revert Errors.PC_ZERO_POOL();
```

```
    }
    if (borrowerAddress == address(0)) {
      revert Errors.PC_ZERO_ADDRESS();
    }
    if (pools[poolHash].parameters.POOL_HASH != poolHash) {
      revert Errors.PC_POOL_NOT_ACTIVE();
    }
    grantRole(BORROWER_ROLE, borrowerAddress);
    borrowerAuthorizedPools[borrowerAddress] = poolHash;
    emit BorrowerAllowed(borrowerAddress, poolHash);
  }

  /**
   * @notice Remove borrower pool interaction rights from an address
   * @param borrowerAddress The address to disallow
   * @param poolHash The identifier of the pool
   **/
  function disallow(address borrowerAddress, bytes32 poolHash) external override
      onlyRole(GOVERNANCE_ROLE) {
    if (poolHash == bytes32(0)) {
      revert Errors.PC_ZERO_POOL();
    }
    if (borrowerAddress == address(0)) {
      revert Errors.PC_ZERO_ADDRESS();
    }
    if (pools[poolHash].parameters.POOL_HASH != poolHash) {
      revert Errors.PC_POOL_NOT_ACTIVE();
    }
    if (borrowerAuthorizedPools[borrowerAddress] != poolHash) {
      revert Errors.PC_DISALLOW_UNMATCHED_BORROWER();
    }
    revokeRole(BORROWER_ROLE, borrowerAddress);
    delete borrowerAuthorizedPools[borrowerAddress];
    emit BorrowerDisallowed(borrowerAddress, poolHash);
  }
```

Listing 3.7: Example Privileged Operations in `ComptrollerImplementation`

Notice that the privilege assignment is necessary and consistent with the protocol design. In the meantime, the extra power to the owner may also be a counter-party risk to the protocol users. Therefore, we list this concern as an issue here from the audit perspective and highly recommend making these privileges explicit or raising necessary awareness among protocol users.

Recommendation Making the above privileges explicit among protocol users.

Status This issue has been confirmed by the team.

3.6 Improved yieldProviderLiquidityRatio Update in BorrowerPools

- ID: PVE-006
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `BorrowerPools`
- Category: Business Logic [6]
- CWE subcategory: CWE-841 [3]

Description

The `Atlendis` protocol provides certain ways to incentivize liquidity providers to provide liquidity on `Atlendis`. First of all, the idle capital is placed on `Aave`, where the earned interest is integrated into the liquidity provider's position on the `Atlendis` protocol, enabling them to increase their exposure and earn additional interest. Moreover, Liquidity providers can accumulate liquidity rewards paid by the borrower once their funds have been exposed to being borrowed. Both the `Aave` interest and the liquidity rewards can be accumulated manually or automatically via the `collectFees()` routine. While examining the logic to accumulate fees, we notice the existence of improper update of the `pool.state.yieldProviderLiquidityRatio`, which impacts the fees collection for the following ticks.

To elaborate, we show below code snippets from the `PoolLogic` library. As the name indicates, the `collectFees()` routine is designed for a user to collect fees for the given tick in the pool. It will call the `collectFeesForTick()` routine which will further invoke the `peekFeesForTick()` routine to peek the updated liquidity ratio and accrued fees for the target tick. The `peekFeesForTick()` routine calculates the yield liquidity ratio increase via `yieldProviderLiquidityRatio - pool.state.yieldProviderLiquidityRatio` (line 563), where the `yieldProviderLiquidityRatio` is the latest yield liquidity ratio read from `Aave` and the `pool.state.yieldProviderLiquidityRatio` is the yield liquidity ratio recorded when the last time the `collectFees()` routine is invoked. It comes to our attention that the `pool.state.yieldProviderLiquidityRatio` is updated every time when the `collectFees()` is invoked, even the pool has available funds on several ticks. As a result, collecting fees for one tick will impact the fees collection for the following ticks as the `pool.state.yieldProviderLiquidityRatio` has been updated to the latest.

```
515     function collectFees(Types.Pool storage pool, uint128 rate) internal {
516         uint128 yieldProviderLiquidityRatio = uint128(
517           pool.parameters.YIELD_PROVIDER.getReserveNormalizedIncome(address(pool.
                parameters.UNDERLYING_TOKEN))
518         );
519         pool.collectFeesForTick(rate, yieldProviderLiquidityRatio);
520         pool.state.yieldProviderLiquidityRatio = yieldProviderLiquidityRatio;
```

```
    }
```

Listing 3.8: PoolLogic::collectFees()

```
  function peekFeesForTick(
    Types.Pool storage pool,
    uint128 rate,
    uint128 yieldProviderLiquidityRatio
  )
    internal
    view
    returns (
      uint128 updatedAtlendisLiquidityRatio,
      uint128 updatedAccruedFees,
      uint128 liquidityRewardsIncrease
    )
  {
    Types.Tick storage tick = pool.ticks[rate];

    if (tick.atlendisLiquidityRatio == 0) {
      return (yieldProviderLiquidityRatio, 0, 0);
    }

    updatedAtlendisLiquidityRatio = tick.atlendisLiquidityRatio;
    updatedAccruedFees = tick.accruedFees;

    uint128 yieldProviderLiquidityRatioIncrease = yieldProviderLiquidityRatio - pool
        .state.yieldProviderLiquidityRatio;

    // get additional fees from liquidity rewards
    liquidityRewardsIncrease = pool.getLiquidityRewardsIncrease(rate);
    uint128 currentNormalizedRemainingLiquidityRewards = pool.state.
        remainingAdjustedLiquidityRewardsReserve.wadRayMul(
      yieldProviderLiquidityRatio
    );
    if (liquidityRewardsIncrease > currentNormalizedRemainingLiquidityRewards) {
      liquidityRewardsIncrease = currentNormalizedRemainingLiquidityRewards;
    }
    // if no ongoing loan, all deposited amount gets the yield provider
    // and liquidity rewards so the global liquidity ratio is updated
    if (pool.state.currentMaturity == 0) {
      updatedAtlendisLiquidityRatio += yieldProviderLiquidityRatioIncrease;
      if (tick.adjustedRemainingAmount > 0) {
        updatedAtlendisLiquidityRatio += liquidityRewardsIncrease.wadToRay().wadDiv(
            tick.adjustedRemainingAmount);
      }
    }
    // if ongoing loan, accruing fees components are added, liquidity ratio will be
        updated at repay time
    else {
      updatedAccruedFees +=
        tick.adjustedRemainingAmount.wadRayMul(yieldProviderLiquidityRatioIncrease)
            +
```

```
            liquidityRewardsIncrease;
        }
    }
```

Listing 3.9: `PoolLogic::peekFeesForTick()`

Recommendation Revise the above `collectFees()` logic to record the `yieldProviderLiquidityRatio` per tick properly.

Status The issue has been fixed by this commit: `1843085`.

4 Conclusion

In this audit, we have analyzed the design and implementation of the `Atlendis` protocol, which is a capital-efficient `DeFi` lending protocol where organizations evolving in the crypto space such as protocols, `DEXes` or `DAOs` can get access to uncollateralized lines of credit. `Atlendis` is targeting entities with regular and short term liquidity needs. Similarly to a revolving line of credit, the protocol allows entities to borrow or issue bonds as many times as they need, up to a preset borrowing limit, without any collateral. `Atlendis`' pools are borrower specific and the borrowing rates are discovered via a limit order book specific to each pool. The current code base is well organized and those identified issues are promptly confirmed and addressed.

Meanwhile, we need to emphasize that `Solidity`-based smart contracts as a whole are still in an early, but exciting stage of development. To improve this report, we greatly appreciate any constructive feedbacks or suggestions, on our methodology, audit findings, or potential gaps in scope/coverage.

References

[1] MITRE. CWE-1126: Declaration of Variable with Unnecessarily Wide Scope. https://cwe.mitre.org/data/definitions/1126.html.

[2] MITRE. CWE-287: Improper Authentication. https://cwe.mitre.org/data/definitions/287.html.

[3] MITRE. CWE-841: Improper Enforcement of Behavioral Workflow. https://cwe.mitre.org/data/definitions/841.html.

[4] MITRE. CWE CATEGORY: 7PK - Security Features. https://cwe.mitre.org/data/definitions/254.html.

[5] MITRE. CWE CATEGORY: Bad Coding Practices. https://cwe.mitre.org/data/definitions/1006.html.

[6] MITRE. CWE CATEGORY: Business Logic Errors. https://cwe.mitre.org/data/definitions/840.html.

[7] MITRE. CWE VIEW: Development Concepts. https://cwe.mitre.org/data/definitions/699.html.

[8] OWASP. Risk Rating Methodology. https://www.owasp.org/index.php/OWASP_Risk_Rating_Methodology.

[9] PeckShield. PeckShield Inc. https://www.peckshield.com.