Zellic

Prepared for
Prince
AVON TECH LTD

Prepared by
Qingying Jie
Varun Verma
Zellic

September 4, 2025

Avon
Smart Contract Security Assessment

Contents

About Zellic

Zellic is a vulnerability research firm with deep expertise in blockchain security. We specialize in EVM, Move (Aptos and Sui), and Solana as well as Cairo, NEAR, and Cosmos. We review L1s and L2s, cross-chain protocols, wallets and applied cryptography, zero-knowledge circuits, web applications, and more.

Prior to Zellic, we founded the #1 CTF (competitive hacking) team ↗ worldwide in 2020, 2021, and 2023. Our engineers bring a rich set of skills and backgrounds, including cryptography, web security, mobile security, low-level exploitation, and finance. Our background in traditional information security and competitive hacking has enabled us to consistently discover hidden vulnerabilities and develop novel security research, earning us the reputation as the go-to security firm for teams whose rate of innovation outpaces the existing security landscape.

For more on Zellic's ongoing security research initiatives, check out our website zellic.io ↗ and follow @zellic_io ↗ on Twitter. If you are interested in partnering with Zellic, contact us at hello@zellic.io ↗.

1. Overview

1.1. Executive Summary

Zellic conducted a security assessment for AVON TECH LTD from August 11th to August 29th, 2025. During this engagement, Zellic reviewed Avon's code for security vulnerabilities, design issues, and general weaknesses in security posture.

1.2. Goals of the Assessment

In a security assessment, goals are framed in terms of questions that we wish to answer. These questions are agreed upon through close communication between Zellic and the client. In this assessment, we sought to answer the following questions:

- Is the protocol working properly?
- Could an attacker steal liquidity deposited by other users?
- Are access control and authorization checks implemented correctly?
- Is the pool accurately accounting for assets and liabilities?

1.3. Non-goals and Limitations

We did not assess the following areas that were outside the scope of this engagement:

- Front-end components
- Infrastructure relating to the project
- Key custody

Due to the time-boxed nature of security assessments in general, there are limitations in the coverage an assessment can provide.

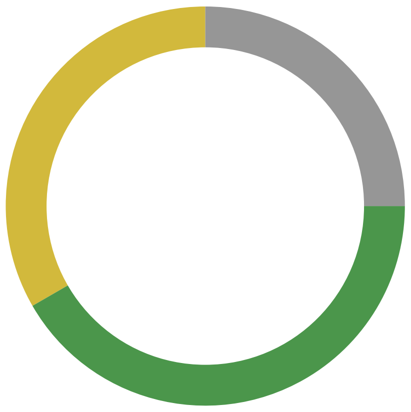
1.4. Results

During our assessment on the scoped Avon contracts, we discovered 12 findings. No critical issues were found. Four findings were of medium impact, five were of low impact, and the remaining findings were informational in nature.

Additionally, Zellic recorded its notes and observations from the assessment for the benefit of AVON TECH LTD in the Discussion section (4. ↗).

Breakdown of Finding Impacts

Impact Level	Count
🟥 Critical	0
🟧 High	0
🟨 Medium	4
🟩 Low	5
⬜ Informational	3



2. Introduction

2.1. About Avon

AVON TECH LTD contributed the following description of Avon:

> Avon is a decentralized lending and borrowing protocol that combines capital-efficient pools with sophisticated liquidity management. The scope for review includes Solidity contracts in the avon-core and avon-periphery repositories.

2.2. Methodology

During a security assessment, Zellic works through standard phases of security auditing, including both automated testing and manual review. These processes can vary significantly per engagement, but the majority of the time is spent on a thorough manual review of the entire scope.

Alongside a variety of tools and analyzers used on an as-needed basis, Zellic focuses primarily on the following classes of security and reliability issues:

Basic coding mistakes. Many critical vulnerabilities in the past have been caused by simple, surface-level mistakes that could have easily been caught ahead of time by code review. Depending on the engagement, we may also employ sophisticated analyzers such as model checkers, theorem provers, fuzzers, and so on as necessary. We also perform a cursory review of the code to familiarize ourselves with the contracts.

Business logic errors. Business logic is the heart of any smart contract application. We examine the specifications and designs for inconsistencies, flaws, and weaknesses that create opportunities for abuse. For example, these include problems like unrealistic tokenomics or dangerous arbitrage opportunities. To the best of our abilities, time permitting, we also review the contract logic to ensure that the code implements the expected functionality as specified in the platform's design documents.

Integration risks. Several well-known exploits have not been the result of any bug within the contract itself; rather, they are an unintended consequence of the contract's interaction with the broader DeFi ecosystem. Time permitting, we review external interactions and summarize the associated risks: for example, flash loan attacks, oracle price manipulation, MEV/sandwich attacks, and so on.

Code maturity. We look for potential improvements in the codebase in general. We look for violations of industry best practices and guidelines and code quality standards. We also provide suggestions for possible optimizations, such as gas optimization, upgradability weaknesses, centralization risks, and so on.

For each finding, Zellic assigns it an impact rating based on its severity and likelihood. There is no hard-and-fast formula for calculating a finding’s impact. Instead, we assign it on a case-by-case basis based on our judgment and experience. Both the severity and likelihood of an issue affect its impact. For instance, a highly severe issue's impact may be attenuated by a low likelihood. We assign the following impact ratings (ordered by importance): Critical, High, Medium, Low, and Informational.

Zellic organizes its reports such that the most important findings come first in the document, rather than being strictly ordered on impact alone. Thus, we may sometimes emphasize an "Informational" finding higher than a "Low" finding. The key distinction is that although certain findings may have the same impact rating, their *importance* may differ. This varies based on various soft factors, like our clients' threat models, their business needs, and so on. We aim to provide useful and actionable advice to our partners considering their long-term goals, rather than a simple list of security issues at present.

Finally, Zellic provides a list of miscellaneous observations that do not have security impact or are not directly related to the scoped contracts itself. These observations — found in the Discussion (4. ↗) section of the document — may include suggestions for improving the codebase, or general recommendations, but do not necessarily convey that we suggest a code change.

2.3. Scope

The engagement involved a review of the following targets:

Avon Contracts

Type	Solidity
Platform	EVM-compatible

Target	avon-core
Repository	https://github.com/avon-xyz/avon-core ↗
Version	`028f3a969efa63438a0d2523e91326df12c1e52b`

Programs	`*.sol`
	`libraries/ErrorsLib.sol`
	`libraries/EventsLib.sol`
	`libraries/MathLib.sol`
	`libraries/OrderbookLib.sol`

Target	Only changes between RedBlackTreeLib.sol and solady's RedBlackTreeLib.sol from commit 29d61c5
Repository	https://github.com/avon-xyz/avon-core ↗
Version	`028f3a969efa63438a0d2523e91326df12c1e52b`
Programs	`RedBlackTreeLib.sol`

Target	avon-periphery
Repository	https://github.com/avon-xyz/avon-periphery ↗
Version	`97b7a0e71e41c7094b2f9a4d8fdad87a506bfb51`
Programs	`src/*.sol`

2.4. Project Overview

Zellic was contracted to perform a security assessment for a total of 4.5 person-weeks. The assessment was conducted by two consultants over the course of three calendar weeks.

Contact Information

The following project managers were associated with the engagement:

Jacob Goreski
Engagement Manager
jacob@zellic.io ↗

Chad McDonald
Engagement Manager
chad@zellic.io ↗

Pedro Moura
Engagement Manager
pedro@zellic.io ↗

The following consultants were engaged to conduct the assessment:

Qingying Jie
Engineer
qingying@zellic.io ↗

Varun Verma
Engineer
varun@zellic.io ↗

2.5. Project Timeline

The key dates of the engagement are detailed below.

August 11, 2025	Kick-off call
August 11, 2025	Start of primary review period
August 29, 2025	End of primary review period

3. Detailed Findings

3.1. Incorrect heapify index in OrderbookLib._matchOrder causing heap corruption

Target	OrderbookLib		
Category	Coding Mistakes	**Severity**	Medium
Likelihood	High	**Impact**	Medium

Description

The `_matchOrder` function in OrderbookLib contains a heap corruption bug when handling partial order fills. After reducing an entry's amount in a partial fill, the code incorrectly increments the index `i` before calling `_heapifyDown`, causing the heap maintenance operation to be performed on the wrong index.

The heap is ordered by available liquidity (descending), meaning larger liquidity amounts should be at the root. When an entry's amount is reduced, it needs to be moved down the heap to maintain this property. However, the code performs heapify on index `i+1` instead of index `i` where the modification occurred.

Impact

This bug progressively corrupts the min-heap structure used for order matching, leading to:

- Orders not being matched in the correct priority order (best liquidity/rates first)
- Lenders with better rates potentially being skipped
- Borrowers receiving worse rates than available in the orderbook
- Violation of the protocol's core promise of efficient order matching

While this doesn't directly cause loss of funds, it undermines the fundamental fairness and efficiency of the orderbook mechanism, potentially causing users to receive suboptimal matches.

```
} else {
    i++;                           // Bug: increment happens before heapify
    tree._heapifyDown(compositeKey, i); // This operates on i+1, not the
    modified index
}
```

Recommendations

Maintain the heap property at the correct index by performing heapify before incrementing:

```
} else {
    tree._heapifyDown(compositeKey, i); // Heapify at the index we just
    modified
    i++;                                // Then move to next index
}
```

Remediation

This issue has been acknowledged by AVON TECH LTD, and a fix was implemented in commit `1c7ea3d2` ↗.

3.2. Incorrect borrowed assets calculation in the function `_isPositionSafe`

Target	PositionGuard		
Category	Coding Mistakes	**Severity**	Medium
Likelihood	Medium	**Impact**	Medium

Description

The function `_isPositionSafe` is used to check whether the position of a specified borrower is healthy by comparing the amount of borrowed assets with the `maxBorrowLimit` calculated based on the collateral value and the loan-to-value ratio.

In the calculation of `maxBorrowLimit`, `s.config.lltv` is represented with the precision of `PoolConstants.WAD`. Therefore, when calculating the borrowed assets, it is necessary to multiply by `PoolConstants.WAD` to maintain consistency in precision with the `maxBorrowLimit`.

An intuitive way to calculate the borrowed assets is `PoolConstants.WAD * position.borrowShares.toAssetUp(s.totalBorrowAssets, s.totalBorrowShares)`. However, the implementation uses `position.borrowShares * s.totalBorrowAssets.toAssetUp(PoolConstants.WAD, s.totalBorrowShares)`.

```
function _isPositionSafe(
    PoolStorage.PoolState storage s,
    address borrower
) internal view returns (bool) {
    PoolStorage.Position memory position = s.positions[borrower];
    if (position.borrowShares == 0) return true;

    uint256 assetRatio = s.totalBorrowAssets.toAssetsUp(PoolConstants.WAD,
    s.totalBorrowShares);
    uint256 borrowedAssets = position.borrowShares * assetRatio;

    uint256 collateralPrice
    = IOracle(s.config.oracle).getCollateralToLoanPrice();
    uint256 maxBorrowLimit = position.collateral
        .mulDiv((collateralPrice * s.config.lltv),
    PoolConstants.ORACLE_PRICE_SCALE);

    return maxBorrowLimit >= borrowedAssets;
}
```

Impact

Because the function `toAssetsUp` adds `VIRTUAL_ASSETS` to the `totalAssets` and adds `VIRTUAL_SHARES` to the `totalShares`, the results of formula `PoolConstants.WAD.toAssetUp(s.totalBorrowAssets, s.totalBorrowShares)` and formula `s.totalBorrowAssets.toAssetUp(PoolConstants.WAD, s.totalBorrowShares)` may differ slightly.

```
function toAssetsUp(uint256 shares, uint256 totalAssets, uint256 totalShares)
    internal pure returns (uint256) {
    return shares.mulDiv(totalAssets + VIRTUAL_ASSETS, totalShares
    + VIRTUAL_SHARES, Math.Rounding.Ceil);
}
```

This will affect the calculation of the `borrowedAssets` and further impact the assessment of the position's health status.

Recommendations

Consider updating based on the following code:

```
uint256 assetRatio = s.totalBorrowAssets.toAssetsUp(PoolConstants.WAD, s.
    totalBorrowShares);

    uint256 assetRatio = uint256(PoolConstants.WAD).toAssetsUp(s.totalBorrowAssets
    , s.totalBorrowShares);
```

Remediation

This issue has been acknowledged by AVON TECH LTD, and a fix was implemented in commit 9b91dbf1 ↗.

3.3. No interest accrual before flash loan

Target	AvonPool		
Category	Business Logic	**Severity**	Medium
Likelihood	Low	**Impact**	Medium

Description

The function `flashLoan` does not accrue interest before executing a flash loan. If `s.flashLoanFee` is not zero, `s.totalSupplyAssets` will be increased after the flash loan.

```
function flashLoan(
    // [...]
) external whenNotPaused {
    PoolStorage.PoolState storage s = PoolStorage._state();
    s._flashLoan(token, assets, data);
    s._updateOrders();
}

function _flashLoan(
    PoolStorage.PoolState storage s,
    address token,
    uint256 assets,
    bytes calldata data
) internal {
    // [...]

    // Calculate flash loan fee
    uint256 feeAmount = assets.mulDiv(s.flashLoanFee, PoolConstants.WAD,
    Math.Rounding.Ceil);

    emit PoolEvents.FlashLoan(msg.sender, token, assets);

    SafeERC20.safeTransfer(ERC20(token), msg.sender, assets);

    IAvonFlashLoanCallback(msg.sender).onAvonFlashLoan(assets, data);

    SafeERC20.safeTransferFrom(ERC20(token), msg.sender, address(this), assets
    + feeAmount);

    s.totalSupplyAssets += feeAmount;
}
```

Since the function `_accrueInterest` calculates the borrow rate based on `totalSupplyAssets` and `totalBorrowAssets`, and then further computes the accrued interest, not accruing interest before executing a flash loan may cause some issues.

Impact

For example, if a user borrows and no one invokes the function `accrueInterest` for a long time, but right before the borrower prepares to repay, someone triggers a flash loan that increases the `s.totalSupplyAssets`. This reduces the utilization of the loan token during this period, causing the borrower to spend fewer assets when repaying compared to the case without a flash loan.

Recommendations

Consider accruing interest before executing a flash loan.

Remediation

This issue has been acknowledged by AVON TECH LTD, and a fix was implemented in commit 5855a721 ↗.

3.4. Malicious users can reduce the manager fee charges by making frequent calls to the function `accrueInterest`

Target	Vault		
Category	Business Logic	**Severity**	Medium
Likelihood	Medium	**Impact**	Medium

Description

The function `accrueInterest` of the contract Vault allows anyone to call.

```
function accrueInterest() external {
    _accrueInterest();
    _updatePrevTotal();
}
```

This function first calculates `managerFeesAmount` based on the difference between the current return value of the function `totalAssets` and the previously recorded value, together with the manager fee rate. It then derives the corresponding shares, mints them to the `feeRecipient`, and finally records the current return value of the function `totalAssets`.

```
function _accrueInterest() internal {
    uint256 currentAssets = totalAssets();

    // Only calculate fees if this isn't the first accrual and there's a gain
    if (prevTotal > 0 && currentAssets > prevTotal) {
        // Calculate the gain (interest earned)
        uint256 gain = currentAssets - prevTotal;

        // Calculate manager's share of the gains
        uint256 managerFeesAmount = gain.mulDiv(managerFees, 1e18);

        if (managerFeesAmount > 0) {
            uint256 shares = managerFeesAmount.mulDiv(
                totalSupply(),
                currentAssets - managerFeesAmount,
                Math.Rounding.Floor
            );
            if (shares > 0) {
                _mint(feeRecipient, shares);
                emit ManagerFeesAccrued(managerFeesAmount, shares);
            }
        }
    }
}

function _updatePrevTotal() internal {
    prevTotal = totalAssets();
    emit TotalAssetsUpdated(prevTotal);
}
```

Impact

If the function `accrueInterest` is called frequently, the change in total assets will be very small. Due to integer division, the manager fees collected will be reduced accordingly.

Recommendations

Consider checking the call interval.

Remediation

This issue has been acknowledged by AVON TECH LTD.

3.5. Configuration parameter checks can be bypassed

Target	AvonPool, Vault		
Category	Business Logic	**Severity**	Medium
Likelihood	Low	**Impact**	Low

Description

In the contracts AvonPool and Vault, some functions wrap the function `schedule` of the contract TimelockController to conveniently schedule updates to the contract settings and perform some sanity checks, such as the functions `updateFlashLoanFee` and `updateManagerFees`. However, for some functions that are scheduled to be called in these wrapper functions, the corresponding checks are not executed, for example in the following code related to updating the `flashLoanFee`.

```
function updateFlashLoanFee(
    uint64 newFee
) external onlyRole(PROPOSER_ROLE) {
    if (newFee > PoolConstants.MAX_FLASH_LOAN_FEE) revert PoolErrors.
        InvalidInput();

    // [...]

    emit PoolEvents.FlashLoanFeeUpdateScheduled(address(this), newFee);
}

// Internal function that will be called by the timelock
function _executeUpdateFlashLoanFee(
    uint64 newFee
) external {
    if (msg.sender != address(this)) revert PoolErrors.Unauthorized();

    PoolStorage.PoolState storage s = PoolStorage._state();
    uint64 oldFee = s.flashLoanFee;
    s.flashLoanFee = newFee;

    emit PoolEvents.FlashLoanFeeUpdated(address(this), oldFee, newFee);
}
```

Impact

Since functions `schedule` and `scheduleBatch` of the contract TimelockController are still callable, proposers can directly call the function `schedule` or function `scheduleBatch` to schedule an update, thereby bypassing checks in the update wrapper functions and setting invalid values.

Recommendations

Consider performing sanity checks within the functions that execute updates.

Remediation

This issue has been acknowledged by AVON TECH LTD, and fixes were implemented in the following commits:

- 4d904970 ↗
- 2f0b8876 ↗
- b796ded4 ↗

3.6. The function `previewBorrow` fails when matched orders exceed `MAX_MATCH_DETAILS`

Target	Orderbook		
Category	Business Logic	**Severity**	Low
Likelihood	Low	**Impact**	Low

Description

The functions `_previewMatchBorrow` and `_previewMatchBorrowWithExactCollateral` return a struct `PreviewMatchedOrder`, which records the detailed information of matched orders, the total amount of matched loan tokens (`totalMatched`), and the total match count. However, when the total match count exceeds `MAX_MATCH_DETAILS`, the order details will no longer be recorded. As a result, the `totalCount` may be greater than the length of the order details arrays.

```
struct PreviewMatchedOrder {
    address[] counterParty;
    uint256[] amounts;
    uint256[] irs;
    uint256[] ltvs;
    uint256 totalMatched;
    uint256 totalCount;
}
```

Impact

When `previewMatchedOrders.totalMatched` is greater than 0, the function `previewBorrow` iterates through the order details, but the maximum value of the index is `totalCount - 1`. Therefore, if the actual number of matched orders exceeds `MAX_MATCH_DETAILS`, the following code will fail due to an out-of-bounds array access.

```
function previewBorrow(PreviewBorrowParams memory previewBorrowParams)
    external
    view
    returns (
        // [...]
    )
{
    if (previewBorrowParams.isMarketOrder) {
        (previewBorrowParams.rate,) = lenderTree._getBestLenderRate();
        previewBorrowParams.ltv = OrderbookLib.MIN_LTV;
    }
    previewMatchedOrders = previewBorrowParams.isCollateral
        ? lenderTree._previewMatchBorrowWithExactCollateral(
            // [...]
        )
        : lenderTree._previewMatchBorrow(previewBorrowParams.rate,
    previewBorrowParams.ltv, previewBorrowParams.amount);
    // [...]
    if (previewMatchedOrders.totalMatched > 0) {
        uint256 matchedOrderCount = previewMatchedOrders.totalCount;
        for (uint256 i; i < matchedOrderCount; i++) {
            previewBorrowParams.isCollateral
                ? loanTokenAmount += previewMatchedOrders.amounts[i]
                : collateralRequired +=
    IPoolImplementation(previewMatchedOrders.counterParty[i]).previewBorrow(
                    previewBorrowParams.borrower,
    previewMatchedOrders.amounts[i], previewBorrowParams.collateralBuffer
                );
        }
        // [...]
    }
}
```

Recommendations

Consider using another variable to record the actual length of the order details arrays.

Remediation

This issue has been acknowledged by AVON TECH LTD, and fixes were implemented in the following commits:

- 444f1cdf ↗
- 1bdcf4a2 ↗

3.7. Matched orders removed from the tree without being recorded when their count exceeds `MAX_MATCH_DETAILS`

Target	OrderbookLib		
Category	Business Logic	**Severity**	Medium
Likelihood	Low	**Impact**	Low

Description

The function `_matchOrder` searches for matching orders in the specified tree based on the given parameters, and returns a struct `MatchedOrder`, which records the detailed information of the matched orders (`counterParty` and `amounts`), the total loan token amount of the matched orders (`totalMatched`), and the total number of matched orders (`totalCount`).

```
struct MatchedOrder {
    address[] counterParty;
    uint256[] amounts;
    uint256 totalMatched;
    uint256 totalCount;
}
```

When an order is matched, the function `_matchOrder` removes it from the tree. However, if the number of matched orders exceeds `MAX_MATCH_DETAILS`, the arrays recording the detailed information of matched orders will no longer be updated, though the matched orders will still be removed from the tree.

```
function _matchOrder(RedBlackTreeLib.Tree storage tree, bool isLender,
    uint64 rate, uint64 ltv, uint256 amount)
    internal
    returns (MatchedOrder memory matchedOrders)
{
    if (amount == 0) revert ErrorsLib.InvalidInput();

    matchedOrders.counterParty = new address[](MAX_MATCH_DETAILS);
    matchedOrders.amounts = new uint256[](MAX_MATCH_DETAILS);
    uint256 matchedCount = 0;
    uint256 remaining = amount;

    bytes32 currentPtr = tree.first();
    while (remaining > 0 && currentPtr != bytes32(0)) {
        // [...]
```

```
            if (_isMatchingOrder(currentPtr, rate, ltv, isLender)) {
                // [...]
                while (i < entriesCount && remaining > 0) {
                    RedBlackTreeLib.Entry storage entry
    = tree.getEntryAt(compositeKey, i);
                    uint256 fill = entry.amount > remaining ? remaining :
    entry.amount;

                    if (recordDetails) {
                        matchedOrders.counterParty[matchedCount] = entry.account;
                        matchedOrders.amounts[matchedCount] = fill;
                        matchedCount++;
                        matchedOrders.totalCount++;
                        recordDetails = matchedCount < MAX_MATCH_DETAILS;
                    }

                    matchedOrders.totalMatched += fill;
                    remaining -= fill;

                    // [...]

                    entry.amount = entry.amount - fill;

                    if (entry.amount == 0) {
                        tree.removeEntry(compositeKey, i);
                        // [...]
                    } else {
                        i++;
                        tree._heapifyDown(compositeKey, i);
                    }
                }
            }
            // [...]
        }
    }
```

Impact

In actual usage, based on the result returned by the function `_matchOrder`, the unique pool addresses involved in the array `counterParty` are aggregated, and the loan token amounts of orders belonging to the same pool are summed. Then, according to the required collateral token amounts returned by each pool's `previewBorrow` function, the functions `depositCollateral` and `borrow` of each pool are called in sequence.

If the actual number of orders actually matched by the function `_matchOrder` is greater than `MAX_MATCH_DETAILS`, there may be a pool to which the matched order belongs to does not overlap with the pools in the array `counterParty`. The matched orders from this unrecorded pool will be removed from the `lenderTree`, while the corresponding pool's `borrow` function will not be invoked by the contract Orderbook during this transaction. As a result, the pool will not be aware that its orders have been matched and thus cannot update its orders in the contract Orderbook in time.

```
function matchMarketBorrowOrder(uint256 amount, uint256 minAmountExpected,
    uint256 collateralBuffer, uint64 ltv, uint64 rate)
    external
    nonReentrant
    whenNotPaused
{
    // [...]
    MatchedOrder memory matchedOrder = lenderTree._matchOrder(false, rate,
    ltv, amount);

    uint256 amountReceived;
    if (matchedOrder.totalMatched > 0) {
        // [...]

        // 1. Aggregate amounts by pool (O(n^2) but acceptable for small n)
        (poolData, uniquePoolCount) = _aggregatePoolData(matchedOrder);

        // 2. Update collateral fields in poolData using the helper function
        totalCollateral = _calculateAndSetPoolCollateral(poolData,
    uniquePoolCount, msg.sender, collateralBuffer);

        // [...]

        // 5. Process all pools
        amountReceived = _processPoolMatches(poolData, uniquePoolCount,
    msg.sender, minAmountExpected);
    }
}
```

Recommendations

Consider ending the matching process when the number of matches reaches `MAX_MATCH_DETAILS`.

Remediation

This issue has been acknowledged by AVON TECH LTD, and fixes were implemented in the following commits:

- 444f1cdf ↗

- `1bdcf4a2` ↗

3.8. Extra time elapsed in the function `_previewAccrueInterest` may cause inaccurate calculations in other functions

Target	PoolGetter		
Category	Business Logic	**Severity**	Low
Likelihood	N/A	**Impact**	Low

Description

The calculation of accrued interest depends on the time elapsed. The function `_previewAccrueInterest` simulates the execution of the function `_accrueInterest`, but for frontend convenience and to allow off-chain users to obtain a predictable state, it adds `PoolConstants.QUOTE_VALID_PERIOD` seconds on top of the original time elapsed.

```
function _previewAccrueInterest(PoolStorage.PoolState storage s)
    internal view returns (PoolData memory previewPool) {
    uint256 elapsed = block.timestamp + PoolConstants.QUOTE_VALID_PERIOD
    - s.lastUpdate;

    // [...]

    int128 scaledRate = ABDKMath64x64.divu(rate, PoolConstants.WAD);
    int128 scaledTime = ABDKMath64x64.fromUInt(elapsed);
    int128 expInput = ABDKMath64x64.mul(scaledRate, scaledTime);
    int128 expResult = ABDKMath64x64.exp(expInput); // e^(r*t)

    // Convert back to uint256 with WAD precision
    uint256 expFactor = ABDKMath64x64.mulu(expResult, PoolConstants.WAD);

    // Accrued interest = totalBorrowAssets * (e^(r*t) - 1)
    uint256 accruedInterest = s.totalBorrowAssets.mulDiv(expFactor
    - PoolConstants.WAD, PoolConstants.WAD);

    // [...]
}
```

Impact

Some functions perform their calculations based on the state returned by `_previewAccrueInterest`. Because the function `_previewAccrueInterest` increases the time elapsed by `PoolConstants.QUOTE_VALID_PERIOD`, these functions — which should be using AvonPool's current state — end up using a future state, leading to inaccurate results. For example, the function `withdraw` of the contract AvonPool uses the function `previewWithdraw` to compute the corresponding number of shares, but the function `previewWithdraw` overridden by the contract AvonPool performs its calculation against the future state returned by `_previewAccrueInterest`, whereas the function `withdraw` should be using AvonPool's current state.

```
// AvonPool
function withdraw(
    uint256 assets,
    address receiver,
    address owner
) public override whenNotPaused returns (uint256 shares) {
    // [...]
    shares = super.withdraw(assets, receiver, owner);
    // [...]
}

function previewWithdraw(uint256 assets)
    public view override returns (uint256) {
    PoolStorage.PoolState storage s = PoolStorage._state();
    (PoolGetter.PoolData memory previewPool) = s._previewAccrueInterest();
    return assets.mulDiv(previewPool.totalSupplyShares
    + 10 ** _decimalsOffset(), previewPool.totalSupplyAssets + 1,
    Math.Rounding.Ceil);
}

// ERC4626
function withdraw(uint256 assets, address receiver, address owner)
    public virtual returns (uint256) {
    uint256 maxAssets = maxWithdraw(owner);
    if (assets > maxAssets) {
        revert ERC4626ExceededMaxWithdraw(owner, assets, maxAssets);
    }

    uint256 shares = previewWithdraw(assets);
    _withdraw(_msgSender(), receiver, owner, assets, shares);

    return shares;
}
```

Recommendations

Consider providing wrapper functions, which could add `PoolConstants.QUOTE_VALID_PERIOD` to the time elapsed, for off-chain use.

Remediation

This issue has been acknowledged by AVON TECH LTD, and a fix was implemented in commit c042d2a1 ↗.

3.9. Unbounded withdrawal loop in Vault could theoretically block withdrawals

Target	Vault		
Category	Code Maturity	**Severity**	Low
Likelihood	Low	**Impact**	Low

Description

The `withdraw` and `redeem` functions in the contract Vault call the function `_performWithdraw`, which contains an unbounded loop through the withdrawal queue:

```
uint256 toWithdraw = assets;
uint256 len = _queue.withdrawQueue.length;
while (toWithdraw > 0 && len > 0)
```

If the withdrawal queue contains many pools and the early pools in the queue are illiquid or encounter errors during withdrawal, the function could potentially run out of gas or fail to access available liquidity that exists later in the queue.

Impact

In edge cases where:

- The withdrawal queue is long
- Early pools are consistently illiquid or reverting
- Significant liquidity exists in pools later in the queue

Users might be temporarily unable to withdraw their funds even though sufficient liquidity exists in the system. However, this scenario requires a specific misconfiguration of the withdrawal queue ordering.

```
while (toWithdraw > 0 && len > 0) {
    // If early pools fail or have no liquidity, loop continues
    // Could theoretically exhaust gas before reaching liquid pools
}
```

Recommendations

Consider implementing a maximum iteration limit or gas checkpoint to ensure the function remains bounded. The vault manager can also mitigate this by properly ordering the withdrawal queue with most liquid pools first.

Remediation

This issue has been acknowledged by AVON TECH LTD.

AVON TECH LTD provided the following comment:

> Generally entries have high amounts to withdraw and up to 10 entries. It’s safe to run and for withdrawal of higher amount than this we don’t want to restrict so it is left as it is.

3.10. Function `_previewAccrueInterest` and function `_accrueInterest` calculate the fee shares in different ways

Target	PoolGetter, AccrueInterest		
Category	Coding Mistakes	**Severity**	Informational
Likelihood	N/A	**Impact**	Informational

Description

The function `_previewAccrueInterest` is used to preview the result of accruing interest before calling a function of the contract AvonPool, while the function `_accrueInterest` is the one invoked by the contract AvonPool during actual execution. However, there are discrepancies between the function `_previewAccrueInterest` and function `_accrueInterest` in the calculation of `managerFeeShares` and `protocolFeeShares`.

In the function `_accrueInterest`, it's like treating `managerFeeAmount` and `protocolFeeAmount` as a whole, and calculating `managerFeeShares` and `protocolFeeShares` based on `totalSupplyShares` and `assetsWithoutFees`. While in the function `_previewAccrueInterest`, the calculations of `managerFeeShares` and `protocolFeeShares` are based on the different total amounts of assets.

```
function _previewAccrueInterest(PoolStorage.PoolState storage s)
    internal view returns (PoolData memory previewPool) {
    // [...]

    uint256 managerFeeAmount;
    uint256 protocolFeeAmount;

    if (s.managerFee != 0) {
        managerFeeAmount = accruedInterest.mulDiv(s.managerFee,
    PoolConstants.WAD);
        uint256 managerFeeShares
    = managerFeeAmount.mulDiv(s.totalSupplyShares,
    previewPool.totalSupplyAssets - managerFeeAmount);
        previewPool.totalSupplyShares += managerFeeShares;
    }

    protocolFeeAmount =
    accruedInterest.mulDiv(IPoolImplementation(address(this)).getProtocolFee(),
    PoolConstants.WAD);
    uint256 protocolFeeShares = protocolFeeAmount.mulDiv(s.totalSupplyShares,
    previewPool.totalSupplyAssets - protocolFeeAmount);
    previewPool.totalSupplyShares += protocolFeeShares;
    // [...]
}

function _accrueInterest(PoolStorage.PoolState storage s)
    internal returns (uint256 managerFeeShares, uint256 protocolFeeShares) {
    // [...]

    uint256 totalNewShares;
    uint256 managerFeeAmount = accruedInterest.mulDiv(managerFee,
    PoolConstants.WAD);
    uint256 protocolFeeAmount =
    accruedInterest.mulDiv(IPoolImplementation(address(this)).getProtocolFee(),
    PoolConstants.WAD);

    uint256 assetsWithoutFees = totalSupplyAssets - managerFeeAmount
    - protocolFeeAmount;

    // Calculate manager fee shares if applicable
    if (managerFee != 0 && accruedInterest > 0) {
        managerFeeShares = managerFeeAmount.mulDiv(totalSupplyShares,
    assetsWithoutFees);
        totalNewShares += managerFeeShares;
    }

    // Calculate protocol fee shares if there's interest
    if (accruedInterest > 0) {
        protocolFeeShares = protocolFeeAmount.mulDiv(totalSupplyShares,
    assetsWithoutFees);
        totalNewShares += protocolFeeShares;
    }

    // Only update if there are new shares
    if (totalNewShares > 0) {
        totalSupplyShares += totalNewShares;
    }

    // [...]
}
```

Impact

If the time elapsed is the same, this will cause the fee shares calculated by the function `_previewAccrueInterest` to be slightly smaller than that from the function `_accrueInterest`.

Recommendations

Consider keeping the calculation of `managerFeeShares` and `protocolFeeShares` consistent between function `_previewAccrueInterest` and function `_accrueInterest`.

Remediation

This issue has been acknowledged by AVON TECH LTD, and a fix was implemented in commit fc5e1754 ↗.

3.11. Handling of the `flatMatchingFee` in the function `previewBorrow`

Target	Orderbook		
Category	Business Logic	**Severity**	Informational
Likelihood	N/A	**Impact**	Informational

Description

The function `previewBorrow` can simulate the execution of a market borrow order or a limit borrow order and return the simulated result. When `previewBorrowParams.isCollateral` is true, it simulates and returns the amount of loan tokens (`loanTokenAmount`) obtainable by borrowing with an exact amount of collateral tokens. Otherwise, it returns the amount of collateral tokens required (`collateralRequired`) to borrow a specified amount of loan tokens.

Although the function `previewBorrow` adds the `flatMatchingFee` to the `collateralRequired` when `isCollateral` is true, it's intuitive to consider that this function returns the amount of loan tokens that can be borrowed with the exact amount of collateral tokens. This is because it returns the amount of collateral tokens required (including the `flatMatchingFee`) with the given amount of loan tokens when `isCollateral` is false.

```
function previewBorrow(PreviewBorrowParams memory previewBorrowParams)
    external
    view
    returns (
        PreviewMatchedOrder memory previewMatchedOrders,
        uint256 loanTokenAmount,
        uint256 collateralRequired,
        uint256 amountLeft
    )
{
    // [...]
    previewMatchedOrders = previewBorrowParams.isCollateral
        ? lenderTree._previewMatchBorrowWithExactCollateral(
            previewBorrowParams.borrower,
            previewBorrowParams.rate,
            previewBorrowParams.ltv,
            previewBorrowParams.amount,
            previewBorrowParams.collateralBuffer
        )
        : lenderTree._previewMatchBorrow(previewBorrowParams.rate,
    previewBorrowParams.ltv, previewBorrowParams.amount);
    if ((previewBorrowParams.amount - previewMatchedOrders.totalMatched) > 0) {
        amountLeft = previewBorrowParams.amount
    - previewMatchedOrders.totalMatched;
    }
    if (previewMatchedOrders.totalMatched > 0) {
        uint256 matchedOrderCount = previewMatchedOrders.totalCount;
        for (uint256 i; i < matchedOrderCount; i++) {
            previewBorrowParams.isCollateral
                ? loanTokenAmount += previewMatchedOrders.amounts[i]
                : collateralRequired +=
    IPoolImplementation(previewMatchedOrders.counterParty[i]).previewBorrow(
                    previewBorrowParams.borrower,
    previewMatchedOrders.amounts[i], previewBorrowParams.collateralBuffer
                );
        }
        if (flatMatchingFee > 0) {
            collateralRequired += flatMatchingFee;
        }
    }
}
```

Impact

If the user does not pay attention to the value of returned `collateralRequired`, the order match may fail during actual execution due to insufficient collateral tokens provided.

Recommendations

Consider deducting the `flatMatchingFee` from the `previewBorrowParams.amount` when calling the function `_previewMatchBorrowWithExactCollateral`.

Remediation

This issue has been acknowledged by AVON TECH LTD, and a fix was implemented in commit a0d3d128 ↗.

3.12. Inconsistency between the function `getPosition` and function `_isPositionSafe` in calculating borrowed assets

Target	PoolGetter		
Category	Business Logic	Severity	Informational
Likelihood	N/A	Impact	Informational

Description

When calculating the `borrowAssets`, the function `getPosition` uses the function `toAssetsDown`, which rounds down during division.

```
function getPosition(
    PoolStorage.PoolState storage s,
    address account
) internal view returns (BorrowPosition memory currentPosition) {
    (currentPosition.borrowShares, currentPosition.collateral)
    = (s.positions[account].borrowShares, s.positions[account].collateral);
    PoolData memory previewPool = _previewAccrueInterest(s);
    currentPosition.borrowAssets = currentPosition.borrowShares.toAssetsDown(
        previewPool.totalBorrowAssets,
        previewPool.totalBorrowShares
    );
}
```

However, the function `_isPositionSafe` uses the function `toAssetsUp` when calculating the user's `borrowedAssets`, i.e. rounding up during division.

```
function _isPositionSafe(
    PoolStorage.PoolState storage s,
    address borrower
) internal view returns (bool) {
    PoolStorage.Position memory position = s.positions[borrower];
    if (position.borrowShares == 0) return true;

    uint256 assetRatio = s.totalBorrowAssets.toAssetsUp(PoolConstants.WAD,
    s.totalBorrowShares);
    uint256 borrowedAssets = position.borrowShares * assetRatio;

    // [...]
}
```

Impact

There is an inconsistency between the function `getPosition` and function `_isPositionSafe` in calculating borrowed assets, while the function `toAssetsDown` is in the borrower's favor.

Recommendations

Consider using the function `toAssetsUp` to calculate.

Remediation

This issue has been acknowledged by AVON TECH LTD, and a fix was implemented in commit cb4720de ↗.

4. Discussion

The purpose of this section is to document miscellaneous observations that we made during the assessment. These discussion notes are not necessarily security related and do not convey that we are suggesting a code change.

4.1. Liquidity sufficiency check in the function `_flashLoan`

`s.totalSupplyAssets` records the total supply of loan tokens in the contract AvonPool, while `s.totalBorrowAssets` records the amount of loan tokens currently borrowed. Therefore, before executing a flash loan, the function `_flashLoan` can check whether the sum of `s.totalBorrowAssets` and the amount of tokens to be sent (`assets`) exceeds `s.totalSupplyAssets` to determine whether the contract has sufficient liquidity.

```
function _flashLoan(
    PoolStorage.PoolState storage s,
    address token,
    uint256 assets,
    bytes calldata data
) internal {
    if (assets == 0) revert PoolErrors.ZeroAddress();
    if (token != s.config.loanToken) revert PoolErrors.InvalidInput();
    if (s.totalBorrowAssets > s.totalSupplyAssets) revert PoolErrors.
        InsufficientLiquidity();

    // [...]
}
```

This issue has been acknowledged by AVON TECH LTD, and a fix was implemented in commit ab04db57 ↗.

4.2. Inconsistency between comments and implementation

The comment for the function `setFlatMatchingFee` states that `flatMatchingFee` is the flat fee amount in loan token.

```
/// @notice Sets the flat matching fee amount
/// @param _flatMatchingFee The flat fee amount in loan token
/// @dev Only the owner can set the flat matching fee
function setFlatMatchingFee(uint256 _flatMatchingFee) external onlyOwner {
    flatMatchingFee = _flatMatchingFee;
    emit EventsLib.FlatMatchingFeeSet(_flatMatchingFee);
}
```

However, in actual usage, the flat fee amount charged is in collateral token. This is inconsistent and should be aligned.

```
function matchMarketBorrowOrder(uint256 amount, uint256 minAmountExpected,
    uint256 collateralBuffer, uint64 ltv, uint64 rate)
    external
    nonReentrant
    whenNotPaused
{
    // [...]

    if (matchedOrder.totalMatched > 0) {
        // [...]

        // 4. Collect matching fee
        if (flatMatchingFee > 0 && feeRecipient != address(0)) {
            collateralToken.safeTransferFrom(msg.sender, feeRecipient,
                flatMatchingFee);
            emit EventsLib.MatchingFeeCollected(msg.sender, feeRecipient,
    flatMatchingFee);
        }

        // [...]
    }
}
```

This issue has been acknowledged by AVON TECH LTD, and a fix was implemented in commit 4726222c ↗.

4.3. Oracle token decimal variance

The handling of oracle decimals and price precision is considered out of scope for this review. The protocol team has indicated that they maintain proper precision configurations for each oracle implementation based on the specific requirements of different token pairs and price feeds. Each oracle may have different decimal conventions and scaling factors that are handled at the oracle integration layer. The protocol assumes that oracle price feeds return values with the appropriate precision as specified in their documentation, and any decimal conversions or normalizations required for consistent pricing across different token decimals would be handled within the oracle implementation itself rather than at the protocol level.

5. System Design

This provides a description of the high-level components of the system and how they interact, including details like a function's externally controllable inputs and how an attacker could leverage each input to cause harm or which invariants or constraints of the system are critical and must always be upheld.

Not all components in the audit scope may have been modeled. The absence of a component in this section does not necessarily suggest that it is safe.

5.1. Component: Borrowing (market + limit matching via Orderbook → Pool borrow)

Description

- Turns borrower intent (market or limit) into loans across whitelisted pools.
- Enforces collateral, liquidity, and pricing checks.
- Updates borrow shares, pool totals, fees, and orderbook state.

What it does:

- Market borrow: `matchMarketBorrowOrder` finds best lender orders, aggregates pools, computes required collateral, pulls once, and may charge a flat fee.
- Limit borrow: `insertLimitBorrowOrder` escrows collateral; later a keeper calls `matchLimitBorrowOrder` at borrower's chosen terms, refunding any excess.
- Pool side: `BorrowRepay._borrow` mints borrow shares, updates totals, checks safety and liquidity, then transfers loan tokens.

How it works:

- Orders stored in a red-black tree keyed by rate/LTV.
- Collateral per pool calculated via `previewBorrow`, then summed and transferred once.
- Fees: optional flat matching fee goes to `feeRecipient`.
- Permissions: pools must be whitelisted; limit matching keeper-only; borrow allowed if sender is borrower, orderbook, or permitted.
- Safety: entry points are `nonReentrant` and `whenNotPaused`; pool accrues interest before updates.

Invariants

- Liquidity: `totalBorrowAssets ≤ totalSupplyAssets` always holds.
- Position safety: borrower's debt must stay ≤ collateral value * LLTV.
- Accounting: borrow shares and assets stay in sync; accrual only increases totals; fee

shares mint against supply.
- Access control: only whitelisted pools can list; only keepers match limit orders; only owner adjusts fees/recipients.

Attack surface

- Borrower params:
- Market: `amount > 0`, `collateralBuffer ≥ 1%`, enforce `minAmountExpected`.
    - Limit: upfront collateral, within `MAX_LIMIT_ORDERS`, keeper-only matching, refund logic, LTV/buffer bounds.
- Matching window: pool state may change, but final `minAmountExpected` guard and live `previewBorrow` mitigate.
- Collateral/oracle: depends on oracle pricing; weak or stale oracles risk under-collateralization, needs system-level hardening.
- Pool selection: can't inject arbitrary pools; whitelist + factory validation required.
- Fees: `flatMatchingFee` may be set high, but limited by `InsufficientAmountReceived` check.
- Reentrancy: guarded with `nonReentrant` and safe ERC20; state updated before pool calls.
- Allowances: orderbook manages approvals; user doesn't expose pools directly.
- DoS: paused pools skipped; match loops capped in length and gas.
- Escrow correctness: limit orders require `totalCollateral + fee ≤ escrowed`; order canceled before pool calls, excess refunded.

5.2. Component: Lending (pools providing liquidity via Orderbook → Borrowers)

Description

- Pools supply liquidity and advertise it in the orderbook.
- Orders are sorted by interest rate and LTV.
- Liquidity is tracked so borrowers can match against it.
- Pool totals, shares, and fees update when lenders deposit or withdraw.

What it does:

- `batchInsertOrder` posts pool supply at chosen rates, replacing older orders.

- When borrowers match, liquidity is used via `_processPoolMatches` calling `depositCollateral` + `borrow`.
- Lenders deposit through ERC4626 (`deposit/mint`), minting shares and updating totals.
- Withdraws burn shares, lower supply, and update orderbook entries.

How it works:

- Pools must be validated and whitelisted first.
- Orders stored in a red-black tree; each pool tied to its entries.
- `accrueInterest` grows total supply over time.
- Fees (protocol/manager) minted as new shares.
- Deposits and withdrawals wrap ERC4626 with pool-specific logic.

Invariants

- Deposits always raise both assets and shares proportionally.
- Withdrawals can't drop supply below outstanding borrows.
- Only whitelisted pools can list orders.
- Orders reset each `batchInsertOrder` call.
- Fees capped so total can't exceed `MAX_TOTAL_FEE`.

Attack surface

- Pool managers: control order posting; must be whitelisted and follow input rules.
- Deposits/withdrawals: checked for >0 amounts, valid receivers, and no overdrawing.
- Order manipulation: spam or bad inputs prevented by ordering checks and pool's LTV cap.
- Fee changes: guarded by timelock/admin, capped by `MAX_TOTAL_FEE`.
- Reentrancy: blocked by `nonReentrant`, safe ERC20, and state-first updates.
- DoS: paused pools skipped, orders cleared each insert.
- Oracle risk: less direct, but still matters for rate models and collateral safety.

6. Assessment Results

During our assessment on the scoped Avon contracts, we discovered 12 findings. No critical issues were found. Four findings were of medium impact, five were of low impact, and the remaining findings were informational in nature.

6.1. Disclaimer

This assessment does not provide any warranties about finding all possible issues within its scope; in other words, the evaluation results do not guarantee the absence of any subsequent issues. Zellic, of course, also cannot make guarantees about any code added to the project after the version reviewed during our assessment. Furthermore, because a single assessment can never be considered comprehensive, we always recommend multiple independent assessments paired with a bug bounty program.

For each finding, Zellic provides a recommended solution. All code samples in these recommendations are intended to convey how an issue may be resolved (i.e., the idea), but they may not be tested or functional code. These recommendations are not exhaustive, and we encourage our partners to consider them as a starting point for further discussion. We are happy to provide additional guidance and advice as needed.

Finally, the contents of this assessment report are for informational purposes only; do not construe any information in this report as legal, tax, investment, or financial advice. Nothing contained in this report constitutes a solicitation or endorsement of a project by Zellic.