

# RubiFi

## Smart Contract Security Assessment

VERSION 1.1



AUDIT DATES:

September 10th to September 15th, 2025

AUDITED BY:

adriro  
ether\_sky

**Contents**

<b>1</b>	<b>Introduction</b>	<b>2</b>
1.1	About Zenith	3
1.2	Disclaimer	3
1.3	Risk Classification	3
<hr/>		
<b>2</b>	<b>Executive Summary</b>	<b>3</b>
2.1	About RubiFi	4
2.2	Scope	4
2.3	Audit Timeline	5
2.4	Issues Found	5
2.5	Security Note	5
<hr/>		
<b>3</b>	<b>Findings Summary</b>	<b>5</b>
<hr/>		
<b>4</b>	<b>Findings</b>	<b>7</b>
4.1	Critical Risk	8
4.2	High Risk	15
4.3	Medium Risk	26
4.4	Low Risk	39
4.5	Informational	45

# 1

## Introduction

Zenith assembles auditors with proven track records: finding critical vulnerabilities in public audit competitions.

Our audits are carried out by a curated team of the industry's top-performing security researchers, selected for your specific codebase, security needs, and budget.

Learn more about us at <https://zenith.security>.

---

## 1.2 Disclaimer

This report reflects an analysis conducted within a defined scope and time frame, based on provided materials and documentation. It does not encompass all possible vulnerabilities and should not be considered exhaustive.

The review and accompanying report are presented on an "as-is" and "as-available" basis, without any express or implied warranties.

Furthermore, this report neither endorses any specific project or team nor assures the complete security of the project.

---

## 1.3 Risk Classification

SEVERITY LEVEL	IMPACT: HIGH	IMPACT: MEDIUM	IMPACT: LOW
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

# 2

## Executive Summary

### 2.1 About RubiFi

The world's first democratized market making platform. Powered by \$RUB, the most scarce and expensive asset in crypto.

---

### 2.2 Scope

The engagement involved a review of the following targets:

**Target** Rubifi\_v1.0

---

**Repository** [https://github.com/SORaRi/Rubifi\\_v1.0](https://github.com/SORaRi/Rubifi_v1.0)

---

**Commit Hash** e844b24c453e95fae4412481a99bf2aa3ed2a9bf

---

**Files** src/\*\*.sol

## 2.3 Audit Timeline

<b>September 10, 2025</b>	Audit start
<b>September 15, 2025</b>	Audit end
<b>November 12, 2025</b>	Report published

---

## 2.4 Issues Found

SEVERITY	COUNT
Critical Risk	4
High Risk	4
Medium Risk	5
Low Risk	6
Informational	1
<b>Total Issues</b>	<b>20</b>

## 2.5 Security Note

Considering the number of issues identified, it is statistically likely that there are more complex bugs still present that could not be identified given the time-boxed nature of this engagement. It is recommended that a follow-up audit and development of a more complex stateful test suite be undertaken prior to continuing to deploy significant monetary capital to production.

# 3

## Findings Summary

ID	Description	Status
C-1	Pending deposits Core<>EVM sync issue affecting pool balance accuracy	Resolved
C-2	processRawAction() should be permissioned	Resolved
C-3	Additional deposits don't inherit profit & loss debt correctly	Resolved
C-4	Gas fees cannot be accidentally sent to the pool	Resolved
H-1	updateUserFeeRateHistory() fails to update the user rate when the time delta is zero	Resolved
H-2	The onUserFeeRateChanged hook should be called after the user's locked balance is updated	Resolved
H-3	The completeWithdrawal function updates totalUsdcCore and totalRubCore incorrectly	Resolved
H-4	Withdrawals don't apply net loss-side adjustments	Resolved
M-1	Fees should not be adjusted by the reduction factor	Resolved
M-2	RF tracking breaks when transferring the token	Resolved
M-3	emergencyWithdraw() in RubLock doesn't trigger fee re-calculation	Resolved
M-4	The NFT should be minted during emergency confirmation actions	Resolved
M-5	Withdrawals may be reverted in some cases	Resolved
L-1	Fee tracking is never reset	Resolved
L-2	Unreachable code	Resolved
L-3	Emergency confirmation actions do not validate if already confirmed	Resolved

ID	Description	Status
L-4	Incorrect fee tier values	Resolved
L-5	Deposits of zero RUB should not require a confirmation	Resolved
L-6	Calling <code>initiateDeposit()</code> with an ongoing deposit may lead to asset loss	Resolved
I-1	The <code>onUserFeeRateChanged</code> hook can not be called in the <code>completeUnlock</code> function	Resolved

# 4

## Findings

### 4.1 Critical Risk

A total of 4 critical risk findings were identified.

#### [C-1] Pending deposits Core<>EVM sync issue affecting pool balance accuracy

SEVERITY: Critical

IMPACT: High

STATUS: Resolved

LIKELIHOOD: High

#### Target

- [ProfitLib.sol](#)

#### Description:

To process profit and losses, the implementation of ProfitLib takes the current spot balance of the pool and adjusts it by the total amount of pending deposits.

```
307:     function _getActualPoolBalances(SharedVaultState.VaultState
  storage state) internal view returns (uint256 actualUsdcBalance,
  uint256 actualRubBalance) {
308:         // Get raw balances from precompiles directly
309:         uint64 rawUsdcBalance
= PrecompileLib.spotBalance(state.poolCoreAddress, 0).total; // USDC
index 0
310:         uint64 rawRubBalance
= PrecompileLib.spotBalance(state.poolCoreAddress, 277).total; // RUB
index 277
311:
312:         // Exclude pending deposits to prevent race conditions
313:         uint256 adjustedUsdcBalance = uint256(rawUsdcBalance)
>= state.totalPendingUsdcDeposits
314:             ? uint256(rawUsdcBalance) - state.totalPendingUsdcDeposits
315:             : 0;
316:         uint256 adjustedRubBalance = uint256(rawRubBalance)
>= state.totalPendingRubDeposits
317:             ? uint256(rawRubBalance) - state.totalPendingRubDeposits
318:             : 0;
```

```
319:  
320:         return (adjustedUsdcBalance, adjustedRubBalance);  
321:     }
```

As deposits can be initiated without any actual transfers, a malicious actor can simply initiate a deposit with an arbitrary amount of assets to manipulate the assets held by the pool.

### Recommendations:

There is no straightforward way to determine the pool balance with the current design, as a natural race condition exists due to the asynchronous mechanics of deposits. Once a user has initiated the deposit flow, the funds may or may not have been sent, and there is no feasible way to determine which state is actually true.

**RubiFi:** Resolved with [@553337014f ...](#) and [@e3b140b800 ...](#).

**Zenith:** Verified. The updated implementation of `_getActualPoolBalances()` doesn't depend on pending deposits anymore and it works correctly.

## [C-2] processRawAction() should be permissioned

SEVERITY: Critical

IMPACT: High

STATUS: Resolved

LIKELIHOOD: High

### Target

- [RUB\\_USDC\\_Vault.sol](#)

### Description:

The processRawAction() action is used to confirm deposits once the user has transferred the assets on the HyperCore layer. A malicious actor can simply skip transferring the assets and call this function with spoofed data, in order to credit deposits out of thin air.

### Recommendations:

The processRawAction() function should be permissioned, presumably to the backend role.

**RubiFi:** Resolved with [@553337014f...](#) and [@e3b140b800...](#). Issue eliminated with deposits on EVM.Core deposit deletions.

**Zenith:** Verified. The processRawAction() function has been removed, and the new deposits implementation correctly handles all related cases without it.

## [C-3] Additional deposits don't inherit profit & loss debt correctly

SEVERITY: Critical

IMPACT: High

STATUS: Resolved

LIKELIHOOD: High

### Target

- [DepositLib.sol](#)

### Description:

Suppose the current `cumulativeUsdcProfitPerShare` is 100. User A makes an initial deposit of 1,000, and their `userUsdcProfitDebt` is updated to 100.

#### [DepositLib.sol#L362](#)

```
function allocateShares(
    SharedVaultState.VaultState storage state,
    address user,
    uint256 usdcDeposit,
    uint256 rubDeposit
) internal {
    // TIME-WEIGHTED PROFIT DEBT: Lock out historical profits
    if (state.userExactUsdcDeposit[user] == 0 &&
        state.userExactRubDeposit[user] == 0) {
        // First deposit - set profit debt to current accumulators
        state.userDepositTimestamp[user] = block.timestamp;
        state.userUsdcProfitDebt[user] = state.cumulativeUsdcProfitPerShare;
        state.userRubProfitDebt[user] = state.cumulativeRubProfitPerShare;
        state.userProfitDebtPerShare[user] = state.cumulativeProfitPerShare;
        // Initialize LOSS debt to current accumulators (prevents claiming
        // historical losses)
        state.userUsdcLossDebt[user] = state.cumulativeUsdcLossPerShare;
        state.userRubLossDebt[user] = state.cumulativeRubLossPerShare;

        // INITIALIZE INDIVIDUAL FEE TRACKING FOR NEW USERS
        Feelib.initializeUserFeeTracking(state, user);
    }
}
```

At this point, they cannot receive any profit, which is correct.

Later, profits accrue in the pool, and `cumulativeUsdcProfitPerShare` increases to 200. This means that User A could receive a total profit of 10,000 based on their current shares.

If User A then makes an additional deposit of 4,000, their `userUsdcProfitDebt` remains unchanged. If they immediately decide to withdraw, their available profit is calculated as 50,000.

#### [WithdrawalLib.sol#L418-L419](#)

```
function _calculateTimeWeightedProfits(
    SharedVaultState.VaultState storage state,
    address user
) internal view returns (uint256 userUsdcNetProfits,
    uint256 userRubNetProfits) {
    // Calculate USDC profits (existing logic)
    uint256 userUsdcProfits = 0;
    if (state.userUsdcShares[user] > 0 && state.cumulativeUsdcProfitPerShare
        > state.userUsdcProfitDebt[user]) {
        uint256 usdcProfitPerShareSinceJoin
        = state.cumulativeUsdcProfitPerShare - state.userUsdcProfitDebt[user];
        userUsdcProfits = (usdcProfitPerShareSinceJoin
        * state.userUsdcShares[user]) / 1e18;
    }
}
```

This occurs because the new shares incorrectly accrue profits based on the old `userUsdcProfitDebt`, which is a bug.

### Recommendations:

The logic for calculating profits and losses should be rewritten correctly.

One approach is to track each user's accumulated profits and losses, and update `userUsdcProfitDebt` to the current `cumulativeUsdcProfitPerShare` whenever the user performs actions such as deposits or withdrawals.

**RubiFi:** Resolved with [@51ad5fa993 ...](#).

**Zenith:** Verified.

## [C-4] Gas fees cannot be accidentally sent to the pool

SEVERITY: Critical

IMPACT: High

STATUS: Resolved

LIKELIHOOD: High

### Target

- [WithdrawalLib.sol](#)

### Description:

When users initiate a withdrawal, they must provide the gas fees. These fees are tracked using the `totalGasFeesCollected` variable.

[WithdrawalLib.sol#L63](#)

```
function initiateWithdrawal(SharedVaultState.VaultState storage state,
    address user, uint256 nftId, uint256 gasFeePaid) external {
    state.userGasFees[user] = gasFeePaid;
    state.totalGasFeesCollected += gasFeePaid;
}
```

The owner can transfer the accumulated fees to the pool using the `transferGasFeesToPool` function.

[WithdrawalLib.sol#L495](#)

```
function transferGasFeesToPool(SharedVaultState.VaultState storage state,
    uint256 amount) external {
    (bool success, ) = state.poolCoreAddress.call{value: amount}("");
    require(success, "Transfer failed");

    state.totalGasFeesCollected -= amount;
}
```

However, the `totalGasFeesCollected` value decreases in the `completeWithdrawal` function even without transferring the fees, which is clearly a bug.

[WithdrawalLib.sol#L113](#)

```
function completeWithdrawal(
    SharedVaultState.VaultState storage state,
    SharedVaultState.VaultState storage feeState,
    address user
) external {
    // Deduct user's gas fee from total collected (before clearing user
    // state)
    state.totalGasFeesCollected -= state.userGasFees[user];

    // Clear user state including exact deposit tracking
    _clearUserState(state, user);
}
```

### Recommendations:

```
function completeWithdrawal(
    SharedVaultState.VaultState storage state,
    SharedVaultState.VaultState storage feeState,
    address user
) external {
    // Deduct user's gas fee from total collected (before clearing user state)
    state.totalGasFeesCollected -= state.userGasFees[user];

    // Clear user state including exact deposit tracking
    _clearUserState(state, user);
}
```

**RubiFi:** Resolved with [@c2cc61e4af ...](#).

**Zenith:** Verified.

## 4.2 High Risk

A total of 4 high risk findings were identified.

[H-1] `updateUserFeeRateHistory()` fails to update the user rate when the time delta is zero

SEVERITY: High

IMPACT: Medium

STATUS: Resolved

LIKELIHOOD: High

### Target

- [FeeLib.sol](#)

### Description:

The implementation of `updateUserFeeRateHistory()` shortcuts the logic when the elapsed time since the last call is zero.

```
200:         uint256 currentTime = block.timestamp;
201:         uint256 timeDelta = currentTime
202:             - state.userLastFeeRateUpdateTime[user];
203:         if (timeDelta > 0) {
204:             // Accumulate: THIS user's previous rate × time at that rate
205:             state.userFeeRateAccumulator[user]
206:                 += (state.userCurrentFeeRate[user] * timeDelta);
207:             // Update to THIS user's new fee rate based on their new RUB
208:             // balance
209:             uint256 currentlyLocked = state.rubLock.lockedBalance(user);
210:             state.userCurrentFeeRate[user]
211:                 = getPlatformFeeByAmount(state, currentlyLocked);
212:             state.userLastFeeRateUpdateTime[user] = currentTime;
213:             emit UserFeeRateUpdated(user,
```

`state.userCurrentFeeRate[user], state.userFeeRateAccumulator[user],  
currentTime);`

While it is acceptable to skip the update of `userFeeRateAccumulator`, as it would be incremented by zero, the implementation also skips the update of `userCurrentFeeRate`

An attacker can manipulate the rate by locking an amount of RUB and unlocking it in the same transaction or block. The FeeLib will store the amount on the first call, and silently skip the second update as the time delta is zero.

### **Recommendations:**

Update the value for `userCurrentFeeRate` even if `timeDelta` is zero.

**RubiFi:** Resolved with [`@0da1f24726 ...`](#).

**Zenith:** Verified.

[H-2] The `onUserFeeRateChanged` hook should be called after the user's locked balance is updated

SEVERITY: High

IMPACT: Medium

STATUS: Resolved

LIKELIHOOD: High

## Target

- [RubLock.sol](#)

## Description:

The `onUserFeeRateChanged` hook is currently called before the `lockedBalance` is updated.

[RubLock.sol#L113](#)

```
function lock(uint256 amount) external nonReentrant whenNotPaused {
    if (amount == 0) revert ZeroAmount();

    // UPDATE THIS USER'S INDIVIDUAL FEE RATE HISTORY (BEFORE BALANCE CHANGE)
    if (address(vaultContract) != address(0)) {
        IVaultFeeTracking(vaultContract).onUserFeeRateChanged(msg.sender);
    }

    // Transfer tokens from user to this contract
    rubToken.safeTransferFrom(msg.sender, address(this), amount);

    // Update balances
    lockedBalance[msg.sender] += amount;
    totalLocked += amount;

    emit TokensLocked(msg.sender, amount, lockedBalance[msg.sender]);
}
```

As a result, the new balance is only considered from the next update, while the old balance is still used for the current calculation.

[FeeLib.sol#L208](#)

\*/

```
function updateUserFeeRateHistory(SharedVaultState.VaultState storage state,
    address user) external {
    // Skip if user hasn't started fee tracking yet
    if (state.userFeeTrackingStartTime[user] == 0) return;

    uint256 currentTime = block.timestamp;
    uint256 timeDelta = currentTime - state.userLastFeeRateUpdateTime[user];

    if (timeDelta > 0) {
        // Accumulate: THIS user's previous rate × time at that rate
        state.userFeeRateAccumulator[user]
        += (state.userCurrentFeeRate[user] * timeDelta);

        // Update to THIS user's new fee rate based on their new RUB balance
        uint256 currentlyLocked = state.rubLock.lockedBalance(user);
        state.userCurrentFeeRate[user] = getPlatformFeeByAmount(state,
        currentlyLocked);
        state.userLastFeeRateUpdateTime[user] = currentTime;

        emit UserFeeRateUpdated(user, state.userCurrentFeeRate[user],
        state.userFeeRateAccumulator[user], currentTime);
    }
}
```

This creates an unfair situation: users who are unlocking can benefit, while those who are locking do not see their updated balance reflected immediately.

### Recommendations:

The `onUserFeeRateChanged` hook should be called at the end of the `lock`, `initiateUnlock`, and `cancelUnlock` functions.

**RubiFi:** Resolved with [@e5f39772a4 ...](#).

**Zenith:** Verified.

## [H-3] The completeWithdrawal function updates totalUsdcCore and totalRubCore incorrectly

SEVERITY: High

IMPACT: High

STATUS: Resolved

LIKELIHOOD: High

### Target

- [WithdrawalLib.sol](#)

### Description:

In the `completeWithdrawal` function, withdrawal amounts are calculated by `_calculateUserWithdrawAmountsEnhanced`, and profits and losses are first updated in `updateSeparateAssetProfitAccumulators`.

#### [WithdrawalLib.sol#L294](#)

```
function _calculateUserWithdrawAmountsEnhanced(
    SharedVaultState.VaultState storage state,
    SharedVaultState.VaultState storage feeState,
    address user
) internal returns (
    uint256 usdcAmount,
    uint256 rubAmount,
    int256 pnlAmount,
    bool isProfitable,
    uint256 usdcFeeAmount,
    uint256 rubFeeAmount
) {
    require(state.userNftId[user] != 0, "User has no pool position");

    // STEP 1: Get user's original deposits (NO IMPERMANENT LOSS)
    uint256 userOriginalUsdc = state.userUsdcDeposits[user];
    uint256 userOriginalRub = state.userRubDeposits[user];

    // STEP 2: Update profit accumulators to capture latest profits
    // This ensures we have the most current profit data for calculation
    ProfitLib.updateSeparateAssetProfitAccumulators(state);
}
```

In this function, `totalUsdcCore` and `totalRubCore` are updated to reflect the current pool balance.

#### [ProfitLib.sol#L236](#)

```
function _updateSeparateAssetProfitAccumulators(SharedVaultState.VaultState
    storage state) internal {
    // STEP 1: Get actual balances (ATOMIC, cannot be manipulated)
    (uint256 actualUsdcBalance, uint256 actualRubBalance)
        = _getActualPoolBalances(state);

    if (actualUsdcBalance > state.totalUsdcCore) {
        usdcProfits = actualUsdcBalance - state.totalUsdcCore;
        // ATOMIC UPDATE: Sync contract state with reality
        state.totalUsdcCore = actualUsdcBalance;
    }

    if (actualRubBalance > state.totalRubCore) {
        rubProfits = actualRubBalance - state.totalRubCore;
        // ATOMIC UPDATE: Sync contract state with reality
        state.totalRubCore = actualRubBalance;
    }
}
```

At this point, the `withdrawn` tokens no longer exist in the pool, and the `fees` have already been transferred to the vault. Therefore, fees are sent directly from the vault to the treasury, meaning there are no token transfers from the pool in `completeWithdrawal`, and `totalUsdcCore` and `totalRubCore` are correctly updated.

However, these totals are incorrectly updated again when transferring `fees` from the vault and then deducted by the `withdrawal` amounts a second time.

#### [WithdrawalLib.sol#L116-L117](#)

```
function completeWithdrawal(
    SharedVaultState.VaultState storage state,
    SharedVaultState.VaultState storage feeState,
    address user
) external {
    // Calculate withdrawal amounts using ENHANCED method (exact deposit + P&L)
    (uint256 usdcAmount, uint256 rubAmount, int256 pnlAmount,
    bool isProfitable, uint256 usdcFeeAmount, uint256 rubFeeAmount) =
        _calculateUserWithdrawAmountsEnhanced(state, feeState, user);

    // Transfer platform fees
    if (usdcFeeAmount > 0 || rubFeeAmount > 0) {
```

```

        FeeLib.transferPlatformFees(feeState, usdcFeeAmount, rubFeeAmount);
    }

    // Update pool balances
    state.totalUsdcCore -= usdcAmount;
    state.totalRubCore -= rubAmount;
}

function transferPlatformFees(SharedVaultState.VaultState storage state,
    uint256 usdcFeeAmount, uint256 rubFeeAmount) external {
    // Transfer USDC fees to treasury using direct CoreWriter
    if (usdcFeeAmount > 0) {
        CoreWriterLib.spotSend(VaultConstants.PLATFORM_TREASURY, 0,
        uint64(usdcFeeAmount)); // USDC index 0

        // Update pool balance to reflect transfer
        state.totalUsdcCore -= usdcFeeAmount;
    }

    // Transfer RUB fees to treasury using direct CoreWriter
    if (rubFeeAmount > 0) {
        CoreWriterLib.spotSend(VaultConstants.PLATFORM_TREASURY, 277,
        uint64(rubFeeAmount)); // RUB index 277

        // Update pool balance to reflect transfer
        state.totalRubCore -= rubFeeAmount;
    }
}

```

This causes `totalUsdcCore` and `totalRubCore` to be lower than the actual balance.

In the next profit calculation, this discrepancy appears as a large difference between the current pool balance and the last `totalUsdcCore` value, which is then incorrectly treated as immediate profits.

### Recommendations:

```

function completeWithdrawal(
    SharedVaultState.VaultState storage state,
    SharedVaultState.VaultState storage feeState,
    address user
) external {
    // Calculate withdrawal amounts using ENHANCED method (exact deposit +
    P&L)
    (uint256 usdcAmount, uint256 rubAmount, int256 pnlAmount,

```

```
bool isProfitable, uint256 usdcFeeAmount, uint256 rubFeeAmount) =  
    _calculateUserWithdrawAmountsEnhanced(state, feeState, user);  
  
    // Transfer platform fees  
    if (usdcFeeAmount > 0 || rubFeeAmount > 0) {  
        FeeLib.transferPlatformFees(feeState, usdcFeeAmount, rubFeeAmount);  
    }  
  
    // Update pool balances  
    state.totalUsdcCore -= usdcAmount;  
    state.totalRubCore -= rubAmount;  
}  
  
function transferPlatformFees(SharedVaultState.VaultState storage state,  
    uint256 usdcFeeAmount, uint256 rubFeeAmount) external {  
    // Transfer USDC fees to treasury using direct CoreWriter  
    if (usdcFeeAmount > 0) {  
        CoreWriterLib.spotSend(VaultConstants.PLATFORM_TREASURY, 0,  
        uint64(usdcFeeAmount)); // USDC index 0  
  
        // Update pool balance to reflect transfer  
        state.totalUsdcCore -= usdcFeeAmount;  
    }  
  
    // Transfer RUB fees to treasury using direct CoreWriter  
    if (rubFeeAmount > 0) {  
        CoreWriterLib.spotSend(VaultConstants.PLATFORM_TREASURY, 277,  
        uint64(rubFeeAmount)); // RUB index 277  
  
        // Update pool balance to reflect transfer  
        state.totalRubCore -= rubFeeAmount;  
    }  
}
```

Or the `transferPlatformFees` function can be removed here.

**RubiFi:** Resolved with [@b80e058847 ...](#).

**Zenith:** Verified.

## [H-4] Withdrawals don't apply net loss-side adjustments

SEVERITY: High

IMPACT: High

STATUS: Resolved

LIKELIHOOD: Medium

### Target

- [WithdrawalLib.sol](#)

### Description:

There can be profits or losses in the pool depending on the trading strategy, and these metrics are tracked correctly.

[ProfitLib.sol#L249](#)

```
function _updateSeparateAssetProfitAccumulators(SharedVaultState.VaultState
storage state) internal {
    // STEP 1: Get actual balances (ATOMIC, cannot be manipulated)
    (uint256 actualUsdcBalance, uint256 actualRubBalance)
    = _getActualPoolBalances(state);

    // STEP 4: Update profit accumulators (ONLY if profits detected)
    if (usdcProfits > 0 && state.totalUsdcShares > 0) {
        uint256 newUsdcProfitPerShare = (usdcProfits * 1e18)
        / state.totalUsdcShares;
        // ACCUMULATE profits (don't replace) - this preserves historical
        profits
        state.cumulativeUsdcProfitPerShare += newUsdcProfitPerShare;
        state.lastUsdcProfitUpdate = block.timestamp;

        emit TradingProfitDetected("USDC", usdcProfits,
        newUsdcProfitPerShare, block.timestamp);
    }

    // STEP 4B: Handle LOSSES (when actualBalance < expectedBalance)
    if (actualUsdcBalance < state.totalUsdcCore) {
        uint256 usdcLosses = state.totalUsdcCore - actualUsdcBalance;
        // ATOMIC UPDATE: Sync contract state with reality
        state.totalUsdcCore = actualUsdcBalance;
```

```

        // ACCUMULATE losses (same logic as profits)
        if (usdcLosses > 0 && state.totalUsdcShares > 0) {
            uint256 newUsdcLossPerShare = (usdcLosses * 1e18)
            / state.totalUsdcShares;
            // ACCUMULATE losses (don't replace) - preserves historical
            losses
            state.cumulativeUsdcLossPerShare += newUsdcLossPerShare;
            state.lastUsdcLossUpdate = block.timestamp;

            emit TradingLossDetected("USDC", usdcLosses,
            newUsdcLossPerShare, block.timestamp);
        }
    }
}

```

When calculating withdrawal amounts, the `_calculateTimeWeightedProfits` function returns 0 if losses exceed profits.

#### [WithdrawalLib.sol#L444-L445](#)

```

function _calculateTimeWeightedProfits(
    SharedVaultState.VaultState storage state,
    address user
) internal view returns (uint256 userUsdcNetProfits,
    uint256 userRubNetProfits) {
    // Calculate USDC profits (existing logic)
    uint256 userUsdcProfits = 0;
    if (state.userUsdcShares[user] > 0 && state.cumulativeUsdcProfitPerShare
    > state.userUsdcProfitDebt[user]) {
        uint256 usdcProfitPerShareSinceJoin
        = state.cumulativeUsdcProfitPerShare - state.userUsdcProfitDebt[user];
        userUsdcProfits = (usdcProfitPerShareSinceJoin
        * state.userUsdcShares[user]) / 1e18;
    }

    // Calculate USDC losses (NEW - mirrors profit logic exactly)
    uint256 userUsdcLosses = 0;
    if (state.userUsdcShares[user] > 0 && state.cumulativeUsdcLossPerShare
    > state.userUsdcLossDebt[user]) {
        uint256 usdcLossPerShareSinceJoin = state.cumulativeUsdcLossPerShare
        - state.userUsdcLossDebt[user];
        userUsdcLosses = (usdcLossPerShareSinceJoin
        * state.userUsdcShares[user]) / 1e18;
    }

    // NET PROFITS = Profits - Losses (ensure no underflow)
}

```

```
    userUsdcNetProfits = userUsdcProfits > userUsdcLosses ? userUsdcProfits  
    - userUsdcLosses : 0;  
}
```

This means that users can always withdraw at least their original deposits.

However, this can lead to a situation where the actual pool balance is less than the requested withdrawal amounts for later withdrawers, preventing them from receiving their full amounts.

### Recommendations:

Losses should also be taken into account when calculating withdrawal amounts. That is, if losses exceed profits, users should receive less than their original deposit.

**RubiFi:** Resolved with [@931b8b8d06 ...](#).

**Zenith:** Verified. The function `_calculateTimeWeightedProfits()` now considers the case when losses are greater than profits. The logic in `_calculateUserWithdrawAmountsEnhanced()` has been updated to apply the net PnL to the original deposited amounts.

## 4.3 Medium Risk

A total of 5 medium risk findings were identified.

### [M-1] Fees should not be adjusted by the reduction factor

SEVERITY: Medium

IMPACT: Medium

STATUS: Resolved

LIKELIHOOD: Medium

#### Target

- [WithdrawalLib.sol](#)

#### Description:

The WithdrawalLib applies a reduction factor when the withdrawal amount is higher than the assets available.

```
299:         // ===== STEP 4: Apply TIME-WEIGHTED platform fees ONLY to
     profits =====
300:         uint256 platformFeePercentage
     = FeeLib.getPlatformFeePercentage(feeState, user);
301:         usdcFeeAmount = 0;
302:         rubFeeAmount = 0;
303:
304:         if (userUsdcProfits > 0) {
305:             usdcFeeAmount = (userUsdcProfits * platformFeePercentage)
     / 1e18;
306:             userUsdcProfits -= usdcFeeAmount;
307:         }
308:
309:         if (userRubProfits > 0) {
310:             rubFeeAmount = (userRubProfits * platformFeePercentage)
     / 1e18;
311:             userRubProfits -= rubFeeAmount;
312:         }
313:
314:         // STEP 5: Final withdrawal amounts (original deposits +
     time-weighted profits)
315:         usdcAmount = userOriginalUsdc + userUsdcProfits;
```

```

316:         rubAmount = userOriginalRub + userRubProfits;
317:
318:
319:         // STEP 6: Handle edge case - insufficient pool balance for
   withdrawal
320:         uint256 availableUsdc = state.totalUsdcCore;
321:         uint256 availableRub = state.totalRubCore;
322:
323:         if (usdcAmount > availableUsdc) {
324:             // Pool doesn't have enough USDC - proportionally reduce
325:             uint256 reductionFactor = (availableUsdc * 1e18)
   / usdcAmount;
326:             usdcAmount = availableUsdc;
327:             userUsdcProfits = (userUsdcProfits * reductionFactor) / 1e18;
328:             usdcFeeAmount = (usdcFeeAmount * reductionFactor) / 1e18;
329:         }
330:
331:         if (rubAmount > availableRub) {
332:             // Pool doesn't have enough RUB - proportionally reduce
333:             uint256 reductionFactor = (availableRub * 1e18) / rubAmount;
334:             rubAmount = availableRub;
335:             userRubProfits = (userRubProfits * reductionFactor) / 1e18;
336:             rubFeeAmount = (rubFeeAmount * reductionFactor) / 1e18;
337:         }

```

In particular, the implementation applies this `reductionFactor` to the `usdcFeeAmount` and `rubFeeAmount` (lines 328 and 336). However, fees are not part of `usdcAmount` and `rubAmount`. Note that these amounts are basically calculated as the sum of the original deposit amount and the profit (lines 315 and 316), and profit has its fee already subtracted (lines 306 and 311).

This means if `usdcAmount > availableUsdc` or `rubAmount > availableRub`, then fees must necessarily be zero, as the available amount doesn't even cover the user-entitled amount.

### Recommendations:

The implementation should prioritize the user amount (original deposit and profits), and then check if the rest of the available amount is enough to cover fees, and potentially apply a second cap. For example:

```

usdcAmount = Math.min(usdcAmount, availableUsdc);
availableUsdc -= usdcAmount;
usdcFeeAmount = Math.min(availableUsdc, usdcFeeAmount);

```

**RubiFi:** Resolved with [@86e1df1fb... .](#)

**Zenith:** Verified. Reduction factor has been removed. Available USDC/RUB now prioritizes first user amounts and then fees.

## [M-2] RF tracking breaks when transferring the token

SEVERITY: Medium

IMPACT: Medium

STATUS: Resolved

LIKELIHOOD: Medium

### Target

- [RF.sol](#)

### Description:

When minted, the NFT ID is associated with the user through the `userNftId` mapping.

```
290:     function mintNft(SharedVaultState.VaultState storage state,
  address user) internal {
291:         if (state.userNftId[user] == 0) {
292:             // First deposit - mint new NFT
293:             uint256 tokenId = state.rfNFT.mint(user);
294:             state.userNftId[user] = tokenId;
295:             emit NFTMinted(user, tokenId);
```

However, the NFT can be freely transferred as any other ERC-721 token. This can be problematic as the withdrawal flow requires the owner of the NFT to match the account stored in `userNftId`.

```
52:     function initiateWithdrawal(SharedVaultState.VaultState storage
  state, address user, uint256 nftId, uint256 gasFeePaid) external {
53:         require(state.rfNFT.ownerOf(nftId) == user, "Not your NFT");
54:         require(state.userNftId[user] == nftId, "NFT not associated with
  your account");
```

### Recommendations:

Consider making the NFT soul-bound, so the user cannot transfer it to a third party. This will also require an adjustment in the `initiateWithdrawal()` function, as the implementation pulls the token into the contract.

**RubiFi:** Resolved with [@60150bf1cf...](#). Applied a conditional soul-bound, preventing transfers except to the vault for escrow.

**Zenith:** Verified. Fixed by restricting NFT transfers.

## [M-3] emergencyWithdraw() in RubLock doesn't trigger fee recalculation

SEVERITY: Medium

IMPACT: Low

STATUS: Resolved

LIKELIHOOD: Medium

### Target

- [RubLock.sol](#)

### Description:

The `emergencyWithdraw()` function of the `RubLock` contract modifies the value of `lockedBalance[user]` but never calls the vault to notify of the modification.

### Recommendations:

Consider calling `onUserFeeRateChanged()` to properly track user fee rates. Alternatively, add an emergency function in the vault to zero the user fee rate.

**RubiFi:** Resolved with [@0e5bf5975b ...](#).

**Zenith:** Verified.

## [M-4] The NFT should be minted during emergency confirmation actions

SEVERITY: Medium

IMPACT: High

STATUS: Resolved

LIKELIHOOD: Low

### Target

- [DepositLib.sol](#)

### Description:

When a deposit is confirmed, an NFT is minted to the depositor, which is very important.

[DepositLib.sol#L122](#)

```
function processRawAction(
    SharedVaultState.VaultState storage state,
    address user,
    bytes calldata data
) external {
    // Handle based on token type (using direct token index comparison)
    if (token == 0) { // USDC token index
        // USDC deposit validation
        require(!state.usdcDepositConfirmed[user], "USDC already
confirmed");
        require(uint256(amount) == state.pendingUsdcDeposits[user], "USDC
amount mismatch");
        confirmUsdcDeposit(state, user, uint256(amount));
    } else if (token == 277) { // RUB token index
        // RUB deposit validation
        require(!state.rubDepositConfirmed[user], "RUB already confirmed");
        require(uint256(amount) == state.pendingRubDeposits[user], "RUB
amount mismatch");
        confirmRubDeposit(state, user, uint256(amount));
    } else {
        revert UnknownToken();
    }

    // Check if both deposits are complete - if so, mint NFT and allocate
shares
```

```

if (state.usdcDepositConfirmed[user] && state.rubDepositConfirmed[user])
{
    mintNft(state, user);
    clearPendingDeposit(state, user);
}
}

```

However, in the `emergencyConfirmUsdcDeposit` and `emergencyConfirmRubDeposit` functions, the NFT is not minted.

#### [DepositLib.sol#L163-L168](#)

```

function emergencyConfirmUsdcDeposit(SharedVaultState.VaultState storage
state, address user) external {
if (state.pendingUsdcDeposits[user] == 0) revert DepositNotInitiated();

uint256 amount = state.pendingUsdcDeposits[user];
confirmUsdcDeposit(state, user, amount);
}

```

As a result, depositors can not be able to withdraw their deposits and any accrued profits in the future.

#### Recommendations:

```

function emergencyConfirmUsdcDeposit(SharedVaultState.VaultState storage
state, address user) external {
if (state.pendingUsdcDeposits[user] == 0) revert DepositNotInitiated();

uint256 amount = state.pendingUsdcDeposits[user];
confirmUsdcDeposit(state, user, amount);

if (state.usdcDepositConfirmed[user]
&& state.rubDepositConfirmed[user]) {
    mintNft(state, user);
    clearPendingDeposit(state, user);
}
}

function emergencyConfirmRubDeposit(SharedVaultState.VaultState storage
state, address user) external {
if (state.pendingRubDeposits[user] == 0) revert DepositNotInitiated();
}

```

```
uint256 amount = state.pendingRubDeposits[user];
confirmRubDeposit(state, user, amount);

if (state.usdcDepositConfirmed[user]
    && state.rubDepositConfirmed[user]) {
    mintNft(state, user);
    clearPendingDeposit(state, user);
}
```

**RubiFi:** Resolved with [@553337014f...](#) and [@e3b140b800...](#). There are no more emergency confirmations needed with the EVM deposits implemented. Everything related to core deposits was deleted.

**Zenith:** Verified. The emergencyConfirmUsdcDeposit() and emergencyConfirmRubDeposit() have been removed, and the new deposits implementation correctly handles all related cases without them.

## [M-5] Withdrawals may be reverted in some cases

SEVERITY: Medium

IMPACT: High

STATUS: Resolved

LIKELIHOOD: Low

### Target

- [WithdrawalLib.sol](#)

### Description:

In the `completeWithdrawal` function, the withdrawal amounts, including fees, are calculated by the `_calculateUserWithdrawAmountsEnhanced` function.

[WithdrawalLib.sol#L89](#)

```
function completeWithdrawal(
    SharedVaultState.VaultState storage state,
    SharedVaultState.VaultState storage feeState,
    address user
) external {
    // Calculate withdrawal amounts using ENHANCED method (exact deposit + P&L)
    (uint256 usdcAmount, uint256 rubAmount, int256 pnlAmount,
    bool isProfitable, uint256 usdcFeeAmount, uint256 rubFeeAmount) =
        _calculateUserWithdrawAmountsEnhanced(state, feeState, user);
}
```

If the withdrawal amount exceeds the available balance, `totalUsdcCore` or `totalRubCore` is set to 0.

[WithdrawalLib.sol#L344](#)

```
function _calculateUserWithdrawAmountsEnhanced(
    SharedVaultState.VaultState storage state,
    SharedVaultState.VaultState storage feeState,
    address user
) internal returns (
    uint256 usdcAmount,
    uint256 rubAmount,
    int256 pnlAmount,
```

```

        bool isProfitable,
        uint256 usdcFeeAmount,
        uint256 rubFeeAmount
    ) {
        // STEP 6.1: Sync pool accounting after reductions (Bug #4 fix)
        // This handles edge cases where pool balance < expected balance
        if (usdcAmount == availableUsdc && (userOriginalUsdc + userUsdcProfits) > availableUsdc) {
            // We gave user all available USDC due to shortfall - sync accounting to reality
            uint256 originalTotal = state.totalUsdcCore;
            state.totalUsdcCore = 0; // Pool will be empty after this withdrawal
            emit PoolUsdcShortfall(user, userOriginalUsdc + userUsdcProfits, availableUsdc);
            emit PoolBalanceReconciled(user, originalTotal, 0, "USDC");
        }

        if (rubAmount == availableRub && (userOriginalRub + userRubProfits) > availableRub) {
            // We gave user all available RUB due to shortfall - sync accounting to reality
            uint256 originalTotal = state.totalRubCore;
            state.totalRubCore = 0; // Pool will be empty after this withdrawal
            emit PoolRubShortfall(user, userOriginalRub + userRubProfits, availableRub);
            emit PoolBalanceReconciled(user, originalTotal, 0, "RUB");
        }
    }
}

```

However, even in this case, the `withdrawal` amounts are still subtracted from `totalUsdcCore` and `totalRubCore`, causing the transaction to be reverted.

#### [WithdrawalLib.sol#L230-L231](#)

```

function completeWithdrawal(
    SharedVaultState.VaultState storage state,
    SharedVaultState.VaultState storage feeState,
    address user
) external {
    // Calculate withdrawal amounts using ENHANCED method (exact deposit + P&L)
    (uint256 usdcAmount, uint256 rubAmount, int256 pnlAmount,
    bool isProfitable, uint256 usdcFeeAmount, uint256 rubFeeAmount) =
        _calculateUserWithdrawAmountsEnhanced(state, feeState, user);

    // Transfer platform fees
}

```

```

    if (usdcFeeAmount > 0 || rubFeeAmount > 0) {
        FeeLib.transferPlatformFees(feeState, usdcFeeAmount, rubFeeAmount);
    }

    // Update pool balances
    state.totalUsdcCore -= usdcAmount;
    state.totalRubCore -= rubAmount;
}

```

## Recommendations:

```

function _calculateUserWithdrawAmountsEnhanced(
    SharedVaultState.VaultState storage state,
    SharedVaultState.VaultState storage feeState,
    address user
) internal returns (
    uint256 usdcAmount,
    uint256 rubAmount,
    int256 pnlAmount,
    bool isProfitable,
    uint256 usdcFeeAmount,
    uint256 rubFeeAmount
) {
    // STEP 6.1: Sync pool accounting after reductions (Bug #4 fix)
    // This handles edge cases where pool balance < expected balance

    if (usdcAmount == availableUsdc && (userOriginalUsdc + userUsdcProfits) >
        availableUsdc) {

        // We gave user all available USDC due to shortfall - sync accounting
        // to reality
        uint256 originalTotal = state.totalUsdcCore;

        state.totalUsdcCore = 0; // Pool will be empty after this withdrawal
        emit PoolUsdcShortfall(user, userOriginalUsdc + userUsdcProfits,
            availableUsdc);
        emit PoolBalanceReconciled(user, originalTotal, 0, "USDC");
    }

    if (rubAmount == availableRub && (userOriginalRub + userRubProfits) >
        availableRub) {

```

```
// We gave user all available RUB due to shortfall - sync accounting  
    to reality  
    uint256 originalTotal = state.totalRubCore;  
  
    state.totalRubCore = 0;    // Pool will be empty after this withdrawal  
    emit PoolRubShortfall(user, userOriginalRub + userRubProfits,  
        availableRub);  
    emit PoolBalanceReconciled(user, originalTotal, 0, "RUB");  
}  
}
```

**RubiFi:** Resolved with [@cbc714f070 ...](#).

**Zenith:** Verified.

## 4.4 Low Risk

A total of 6 low risk findings were identified.

### [L-1] Fee tracking is never reset

SEVERITY: Low	IMPACT: Low
STATUS: Resolved	LIKELIHOOD: Medium

#### Target

- [RF.sol](#)

#### Description:

Fee tracking for users is enabled by the `initializeUserFeeTracking()` function, which checks if the `userFeeTrackingStartTime[user]` is initialized to initialize the state associated with fees.

Given that the `userFeeTrackingStartTime[user]` variable is never reset, fee tracking will start on the first deposit to the vault and accumulate historical changes, even if the user doesn't have an active deposit.

This means that time-weighted fees will not cover only the deposit -> withdrawal cycle, but the complete history, accumulating previous cycles and even intervals of time where the user didn't have an active position on the vault.

#### Recommendations:

Reset the variable on withdrawal so that the next deposit re-initializes the state to properly account for fees during each cycle.

**Rubifi:** Resolved with [@5389dc4e19 ...](#).

**Zenith:** Verified. Fee-related variables are now reset to zero in `_clearUserState ()`.

## [L-2] Unreachable code

SEVERITY: Low

IMPACT: Low

STATUS: Resolved

LIKELIHOOD: Low

### Target

- [FeeLib.sol#L145](#)
- [FeeLib.sol#L155](#)
- [FeeLib.sol#L165](#)
- [ProfitLib.sol#L148](#)
- [ProfitLib.sol#L168](#)
- [WithdrawalLib.sol#L143](#)
- [WithdrawalLib.sol#L254](#)

### Description:

The listed functions belong to libraries that are not exposed through the vault contract.

### Recommendations:

Consider exposing these functions or deleting them if not needed.

**RubiFi:** Resolved with [@3cc4377012 ...](#).

**Zenith:** Verified. The highlighted functions have been removed.

## [L-3] Emergency confirmation actions do not validate if already confirmed

SEVERITY: Low

IMPACT: Low

STATUS: Resolved

LIKELIHOOD: Low

### Target

- [DepositLib.sol](#)

### Description:

The `emergencyConfirmUsdcDeposit()` and `emergencyConfirmRubDeposit()` functions do not validate if the deposit is already confirmed.

### Recommendations:

Consider adding a validation to prevent an accidental double-confirmation.

**RubiFi:** Resolved with [@553337014f...](#) and [@e3b140b800...](#). Emergency confirmations no longer needed for EVM deposits.Core deposit deletions.

**Zenith:** Verified. The `emergencyConfirmUsdcDeposit()` and `emergencyConfirmRubDeposit()` have been removed and new deposits implementation correctly handles all related cases without them.

## [L-4] Incorrect fee tier values

SEVERITY: Low

IMPACT: Low

STATUS: Resolved

LIKELIHOOD: Medium

### Target

- [VaultConstants.sol](#)

### Description:

The values for FEE\_TIER\_2\_MIN\_RUB and FEE\_TIER\_3\_MIN\_RUB are incorrect with respect to their associated comments:

```
133:     /// @dev Tier 2: 0.00002 RUB locked = 50% platform fee = 50% user
      profit share
134:     uint256 internal constant FEE_TIER_2_MIN_RUB = 2e4;    // 0.00002 *
      1e10
135:     uint256 internal constant FEE_TIER_2_RATE = 0.50e18;
136:
137:     /// @dev Tier 3: 0.00020 RUB locked = 40% platform fee = 60% user
      profit share
138:     uint256 internal constant FEE_TIER_3_MIN_RUB = 2e5;    // 0.00020 *
      1e10
139:     uint256 internal constant FEE_TIER_3_RATE = 0.40e18;
```

### Recommendations:

The value for FEE\_TIER\_2\_MIN\_RUB should be 2e5, and the value for FEE\_TIER\_3\_MIN\_RUB should be 2e6.

**RubiFi:** Resolved with [@f99dbeadb ...](#).

**Zenith:** Verified.

## [L-5] Deposits of zero RUB should not require a confirmation

SEVERITY: Low

IMPACT: Low

STATUS: Resolved

LIKELIHOOD: Low

### Target

- [DepositLib.sol](#)

### Description:

A zero RUB deposit is technically possible according to the `validateMinimumDeposit()` function. A user can deposit some USDC to pass the minimum amounts and specify a zero value for `rubAmount`.

When this is the case, the `processRawAction()` function still requires an explicit confirmation for the RUB side. As there isn't a transfer of RUB, the backend may never confirm it, leading to a stalled deposit.

### Recommendations:

The implementation of `processRawAction()` should confirm the deposit when processing the USDC side, and the RUB deposit amount is zero.

**RubiFi:** Resolved with [@553337014f ...](#) and [@e3b140b800 ...](#). Issue eliminated with deposits on EVM.Core deposit deletions.

**Zenith:** Verified. The `processRawAction()` has been removed, and the new deposits implementation correctly handles all related cases without it.

## [L-6] Calling `initiateDeposit()` with an ongoing deposit may lead to asset loss

SEVERITY: Low

IMPACT: Low

STATUS: Resolved

LIKELIHOOD: Low

### Target

- [DepositLib.sol](#)

### Description:

The implementation of `initiateDeposit()` clears the user state if there is a pending deposit.

While calling it a second time before the deposit is confirmed could be interpreted as a correction of the amount, the user may mistakenly call it again after transferring the assets and be unaware that the backend hasn't actually completed the deposit (or it is partially confirmed).

### Recommendations:

Review the semantics of `initiateDeposit()` when an existing pending deposit is present. This can be strengthened by adding a validation to check that there isn't a confirmed deposit, and also by adding an explicit function to cancel a pending deposit.

**RubiFi:** Resolved with [@553337014f...](#) and [@e3b140b800...](#). Issue eliminated with deposits on EVM.Core deposit deletions.

**Zenith:** Verified. The `initiateDeposit()` function has been removed, and the new deposits implementation correctly handles all related cases without it

## 4.5 Informational

A total of 1 informational findings were identified.

### [I-1] The `onUserFeeRateChanged` hook can not be called in the `completeUnlock` function

SEVERITY: Informational

IMPACT: Informational

STATUS: Resolved

LIKELIHOOD: Low

#### Target

- [RubLock.sol](#)

#### Description:

There is no need to call the `onUserFeeRateChanged` hook in the `completeUnlock` function, since the user's locked balance does not change.

[RubLock.sol#L164-L166](#)

```
function completeUnlock() external nonReentrant whenNotPaused {
    uint256 amount = unlockingBalance[msg.sender];

    // UPDATE THIS USER'S INDIVIDUAL FEE RATE HISTORY
    if (address(vaultContract) != address(0)) {
        IVaultFeeTracking(vaultContract).onUserFeeRateChanged(msg.sender);
    }

    // Clear unlock state
    unlockingBalance[msg.sender] = 0;
    unlockTimestamp[msg.sender] = 0;
    totalUnlocking -= amount;

    // Transfer tokens back to user
    rubToken.safeTransfer(msg.sender, amount);

    emit UnlockCompleted(msg.sender, amount);
}
```

## Recommendations:

```
function completeUnlock() external nonReentrant whenNotPaused {
    uint256 amount = unlockingBalance[msg.sender];

    // UPDATE THIS USER'S INDIVIDUAL FEE RATE HISTORY
    if (address(vaultContract) != address(0)) {
        IVaultFeeTracking(vaultContract).onUserFeeRateChanged(msg.sender);
    }

    // Clear unlock state
    unlockingBalance[msg.sender] = 0;
    unlockTimestamp[msg.sender] = 0;
    totalUnlocking -= amount;

    // Transfer tokens back to user
    rubToken.safeTransfer(msg.sender, amount);

    emit UnlockCompleted(msg.sender, amount);
}
```

**RubiFi:** Resolved with [@a4948b4a79 ...](#).

**Zenith:** Verified.