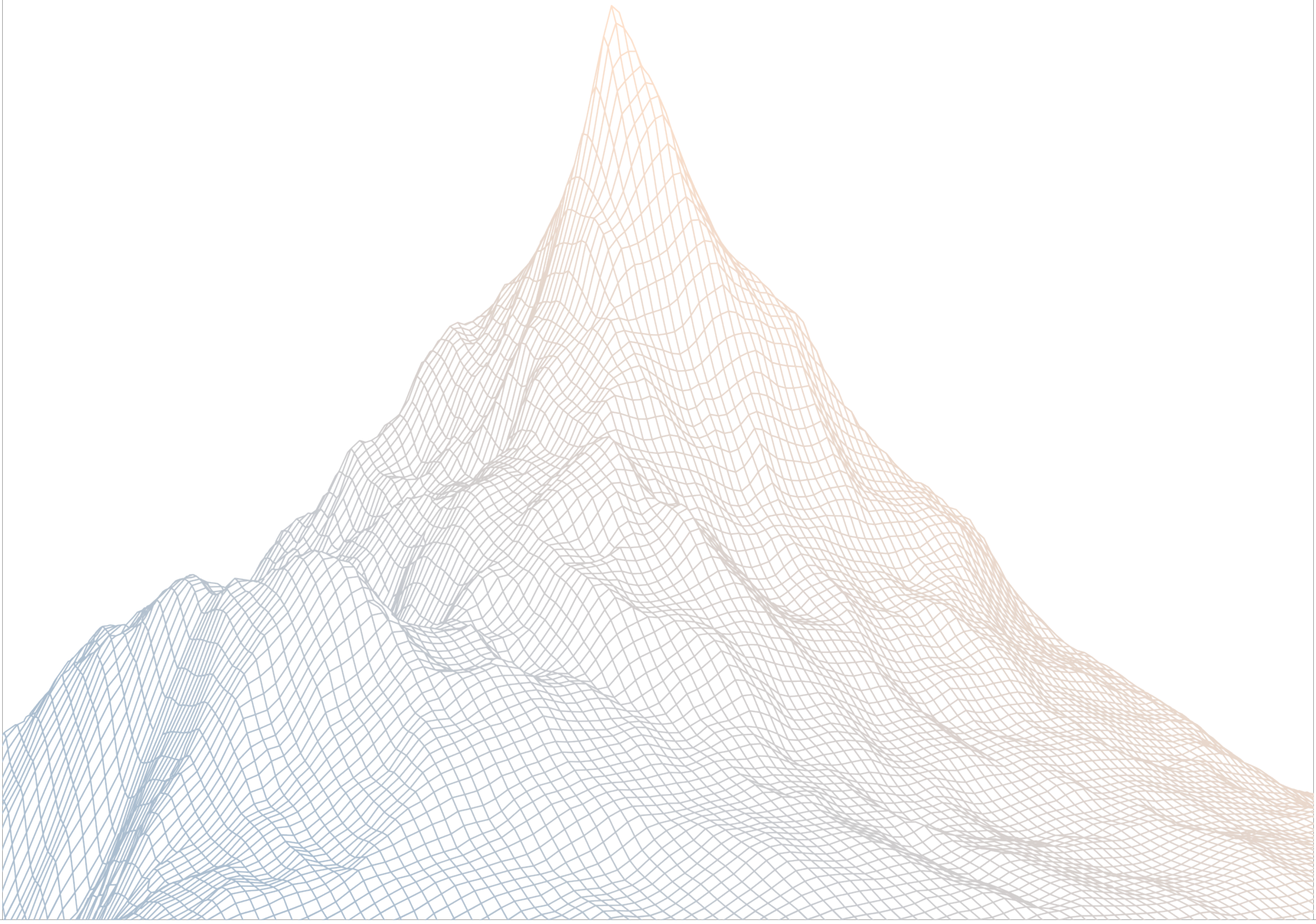
Zenith

RubiFi

Smart Contract Patch Review

VERSION 2.0



AUDIT DATES: October 23rd to October 27th, 2025

AUDITED BY: adriro
ether_sky

Contents

1 Introduction

1.1 About Zenith

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/S0RaRi/Rubifi_v1.0
Commit Hash	`70b24f574d16e0335704af31815aff8ccda2c704`
Files	`src/**.sol`

2.3 Audit Timeline

October 23, 2025	Audit start
October 27, 2025	Audit end
November 12, 2025	Report published

2.4 Issues Found

SEVERITY	COUNT
Critical Risk	1
High Risk	7
Medium Risk	6
Low Risk	2
Informational	3
Total Issues	**19**

2.5 Security Note

This report represents a follow-up audit to the engagement conducted between October 23rd and October 27th, 2025, where our team completed 5 days of in-depth security review of the agreed-upon scope.

3 Findings Summary

ID	Description	Status
C-1	In-flight EVM <-> CORE movements would cause fluctuations in share price	Resolved
H-1	The deposit can fail because of an underflow	Resolved
H-2	Vault core balances need to be accounted in _getActualPoolBalances()	Resolved
H-3	Share issuance does not reflect current vault valuation	Resolved
H-4	RUB Locks from airdrop are not accounted for in the fee calculation	Resolved
H-5	The depositSingleToken function performs an incorrect check for the minimum USDC deposit amount	Resolved
H-6	Profits and losses should be updated at least one block after a deposit	Resolved
H-7	Profits and losses should be updated before processing a deposit	Resolved
M-1	Protocol fees should also be accounted for in totals during withdrawals	Resolved
M-2	emergencyCompleteWithdrawal() should not decrement the user's gas contribution	Resolved
M-3	Permanent revert in depositSingleToken() due to zero-amount validation	Resolved
M-4	The pool cap check in depositSingleToken should use the actual deposited amount	Resolved
M-5	The completeWithdrawal function performs an incorrect check for the available amount	Resolved
M-6	Deposits should be disabled while withdrawals are pending	Resolved
L-1	Implement emergency actions to handle the vault's spot balances	Resolved
L-2	Validate user has not claimed aidrop before assigning an amount	Resolved

ID	Description	Status
I-1	The admin should have the ability to transfer funds from EVM to the core	Resolved
I-2	updateLifiSelector() should not restrict zero selector	Resolved
I-3	Duplicate setter	Resolved

4 Findings

4.1 Critical Risk

A total of 1 critical risk findings were identified.

[C-1] In-flight EVM <-> CORE movements would cause fluctuations in share price

SEVERITY: Critical	IMPACT: High
STATUS: Resolved	LIKELIHOOD: High

Target

- DepositLib.sol
- WithdrawalLib.sol

Description:

Deposits and withdrawals require coordination between the EVM and the CORE side. Users deposit assets on the EVM, which bridges to the CORE side and transfers them to the pool. Similarly, withdrawals first transfer assets out of the pool and are later completed by an operator on the EVM side.

Communication between EVM and CORE is asynchronous in HyperLiquid, meaning that actions cannot be executed atomically. This causes an issue in the assumptions implemented to handle vault actions.

The deposit flow bridges assets to CORE and immediately records the totals on the EVM side.

```
        // 4. Transfer tokens from user to contract
        IERC20(state.usdcToken).safeTransferFrom(user, address(this), usdcAmount);
        IERC20(state.rubToken).safeTransferFrom(user, address(this), rubAmount);

        // 5. Bridge both tokens to Core (they go to contract's Core balance)
        CoreWriterLib.bridgeToCore(state.usdcToken, usdcAmount);
        CoreWriterLib.bridgeToCore(state.rubToken, rubAmount);

        // 6.  Send from contract's Core balance to pool's Core address
        uint64 usdcIndex = PrecompileLib.getTokenIndex(state.usdcToken);
        uint64 rubIndex = PrecompileLib.getTokenIndex(state.rubToken);

        CoreWriterLib.spotSend(state.poolCoreAddress, usdcIndex, usdcCore);
        CoreWriterLib.spotSend(state.poolCoreAddress, rubIndex, rubCore);

        // 7.  Update Core balances
        state.userUsdcCore[user] += usdcCore;
        state.userRubCore[user] += rubCore;
        state.totalUsdcCore += usdcCore;
        state.totalRubCore += rubCore;
```

In the withdrawal case, an operator transfers assets from the pool to the vault (on the CORE side) and later calls `completeWithdrawal()`, which queues the transfers for the user and also adjusts the totals on the EVM side.

```
        // 2.  Send user's withdrawal from pool to user
        if (usdcAmount > 0) {
            CoreWriterLib.spotSend(user, 0, uint64(usdcAmount));
        }
        if (rubAmount > 0) {
            CoreWriterLib.spotSend(user, 277, uint64(rubAmount));
        }

        // 3.  Update pool balances BEFORE clearing user state
        state.totalUsdcCore -= usdcAmount;
        state.totalRubCore -= rubAmount;
```

In both of these paths, there is a gap where the totals are not in sync with the actual pool balances, which are used to determine the vault's profits and losses through the `_updateSeparateAssetProfitAccumulators()` function.

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

        // STEP 2:  Store old balances for logging
        uint256 expectedUsdc = state.totalUsdcCore;
        uint256 expectedRub = state.totalRubCore;
```

This asymmetry leads to fluctuations in the share price that would affect deposits or withdrawals executed in between. Suppose we start from a point where EVM totals match the actual balances on CORE:

1. Attacker executes a normal deposit. This increases `totalUsdcCore/totalRubCore` but not the actual balances in CORE, as these will impact in the future.
2. Attacker checkpoints the profits using `updateSeparateAssetProfitAccumulators()`. Given that the actual balances are below `totalUsdcCore/totalRubCore`, the vault will suffer a temporary loss.
3. Attacker makes a new deposit.
4. Once balances are settled, the attacker checkpoints the profits again, the vault recovers the difference as a profit, and the attacker withdraws the assets with the artificial positive difference.

Recommendations:

The asynchronous nature between EVM and CORE would require a design change.

1. Instead of relying on effective balances, implement a permissioned oracle that could tell the pool's share price or total assets.
2. Or, implement a mechanism to consider and account for in-flight movements. For example, when depositing, besides adjusting `totalUsdcCore/totalRubCore`, increment a temporary counter that should only be considered for the current L1 block number.

RubiFi: Resolved with `@02b5ea968f...`. Also, the `backend` checks with `getCoreHypeBalance()` before calling `completeWithdrawal()`.We control when the `backend` calls and we check for gas before the call. We also have the `retryVaultCoreToUser()` function in case it happens.

Zenith: Verified. The coordination between CORE and EVM cannot fully guarantee an atomic operation, potentially causing unintended shifts in the share price. As an example, the state can go out of sync if the queued transfer in CORE from the vault account to the user fails or isn't executed immediately after the EVM block.1) The `backend` calls `completeWithdrawal()`, which updates the user/global state in the EVM and queues the transfer in CORE. 2) The transfer in CORE fails and isn't executed. 3) Back in the EVM, the next call to `updateSeparateAssetProfitAccumulators()` would count the difference in assets as a profit.

Given that the backend verifies gas availability before invoking completeWithdrawal(), this scenario is not considered probable in practice. The backend will check all preconditions to ensure the transacion in CORE doesn't fail

4.2 High Risk

A total of 7 high risk findings were identified.

[H-1] The deposit can fail because of an underflow

SEVERITY: High	IMPACT: High
STATUS: Resolved	LIKELIHOOD: Medium

Target

- DepositLib.sol

Description:

In the `depositSingleToken` function, the `minRubOut` parameter is used to check for `slippage` and the actual `swap amount` should be greater than this value.

- DepositLib.sol#L399

```
function depositSingleToken(
    SharedVaultState.VaultState storage state,
    address user,
    uint256 usdcAmount,
    bytes calldata lifiSwapData,
    uint256 minRubOut
) external {
     ...
    uint256 rubReceived = rubBalanceAfter - rubBalanceBefore;

    // 12. Validate slippage protection
    if (rubReceived < minRubOut) {
        revert SlippageExceeded(minRubOut, rubReceived);
    }

    uint256 slippageBps = 0;
    if (rubReceived < minRubOut * 105 / 100) { // Allow 5% buffer above
    minRubOut
        slippageBps = ((minRubOut - rubReceived) * 10000) / minRubOut;
    }
}
```

However, if the amount is less than `105%` of the `slippage`, an `underflow` occur, causing the `deposit` transaction to revert.

Recommendations:

```
function depositSingleToken(
    SharedVaultState.VaultState storage state,
    address user,
    uint256 usdcAmount,
    bytes calldata lifiSwapData,
    uint256 minRubOut
) external {
    ...
    uint256 rubReceived = rubBalanceAfter - rubBalanceBefore;

    // 12. Validate slippage protection
    if (rubReceived < minRubOut) {
        revert SlippageExceeded(minRubOut, rubReceived);
    }

    uint256 slippageBps = 0;
    if (rubReceived < minRubOut * 105 / 100) { // Allow 5% buffer above
    minRubOut
-       slippageBps = ((minRubOut - rubReceived) * 10000) / minRubOut;
+       slippageBps = ((rubReceived - minRubOut) * 10000) / minRubOut;
    }
}
```

RubiFi: Resolved with @1b9f47cce7 ... .

Zenith: Verified.

[H-2] Vault core balances need to be accounted in _getActualPoolBalances()

SEVERITY: High	IMPACT: Medium
STATUS: Resolved	LIKELIHOOD: High

Target

- WithdrawalLib.sol

Description:

The withdrawal flow requires users to first initiate the process, and then for an operator to complete it.

1. User calls `initiateWithdrawal()`.
2. Operator moves funds from the pool to the vault on the CORE side.
3. Operator calls `completeWithdrawal()` which queues asset transfers from the vault to the user on the CORE side.

This has two major problems:

1. The operator must know beforehand how many assets the `completeWithdrawal()` function will require. It can be simulated, but ultimately would be determined by the on-chain execution. Note that transfers in `completeWithdrawal()` are asynchronous, which means that a successful execution of the EVM transaction doesn't mean the corresponding asset transfers on CORE will be successful too.
2. Profit and losses must be calculated before completing the withdrawal to apply the PnL to the user's deposit. The `updateSeparateAssetProfitAccumulators()` function uses the current pool balances to compare against the registered totals, but since assets have been moved from the pool to the vault account (step 2 in the process), they won't be accounted for in `_getActualPoolBalances()` and potentially assessed as a loss.

Recommendations:

For the asset transfers, ensure to provide enough margin when moving funds from the pool to the vault that could act as a buffer to account for potential shifts in the final amounts.

This would also require a protocol operator accessible function to eventually return an excess of funds from the vault back to the pool.

The implementation of `_getActualPoolBalances()` should also consider CORE balances in the vault, not just the pool, to account for these transient funds being moved between pool and vault.

RubiFi: Resolved with `@320d0333ff ...`.

Zenith: Verified.

[H-3] Share issuance does not reflect current vault valuation

`SEVERITY`: High	`IMPACT`: High
`STATUS`: Resolved	`LIKELIHOOD`: High

Target

- DepositLib.sol

Description:

Currently, the `share` price remains the same regardless of the `deposit` time. `Profits` and `losses` are calculated based on these `shares`, which can lead to incorrect results. For example, if `User 1` `deposits` `1,000 assets`, he receive `1,000 shares`. If there is a `600 loss`, only `400 assets` remain. `User 2` `deposits` `1,000 assets` and receives `1,000 shares`. At this point, `User 1` and `User 2` hold the same `shares`. If there is a `1,000 loss`, it is applied equally to both users because their `shares` are the same. This results in:

- `User 1`'s available `withdrawal` amount: `1,000 - 600 - 500 = -100`
- `User 2`'s available `withdrawal` amount: `1,000 - 500 = 500`

Clearly, this calculation is incorrect.

Recommendations:

The `share` price should be calculated as the ratio of `total assets` to `total shares`.

RubiFi: Resolved with `@77fb291ac6 ...`.

Zenith: Verified.

[H-4] RUB Locks from airdrop are not accounted for in the fee calculation

SEVERITY: High	IMPACT: Medium
STATUS: Resolved	LIKELIHOOD: High

Target

- RUB_USDC_Vault.sol
- FeeLib.sol

Description:

The RubLock contract has been modified to include user locks coming from the new RubAirdrop mechanism. These tokens are locked for a period of 180 days and have a 1.5x multiplier, which is implemented in the `getEffectiveBalance()`

```
    function getEffectiveBalance(address user) external view returns (uint256) {
        // 7-day locked + (180-day locked * 1.5)
        return lockedBalance[user] + (totalLocked180[user] * LOCK180_MULTIPLIER / 100);
    }
```

However, the vault implementation queries the locks using the `lockedBalance()` function, which only accounts for the 7-day type of lock. For example, this is the implementation of `updateUserFeeRateHistory()`, used to re-calculate the fee after a lock balance change:

```
    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];

        // 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);
    }
```

Recommendations:

Calculate fees using `getEffectiveBalance()` instead of `lockedBalance()`.

The following functions are affected:

- `Vault.updateFeeTiers()`
- `FeeLib.getPlatformFeePercentage()`
- `FeeLib.getCurrentStakingStatus()`
- `FeeLib.updateUserFeeRateHistory()`
- `FeeLib.getTimeWeightedFeeRate()`
- `FeeLib.initializeUserFeeTracking()`

RubiFi: Resolved with `@3a2ba848f9 ...`.

Zenith: Verified.

[H-5] The depositSingleToken function performs an incorrect check for the minimum USDC deposit amount

SEVERITY: High	IMPACT: High
STATUS: Resolved	LIKELIHOOD: Medium

Target

- DepositLib.sol

Description:

In the `depositSingleToken` function, the `usdcAmount` parameter uses `6 decimals`, as it represents an `EVM amount`. However, the `getMinDepositValueUsdc` function returns the minimum `USDC` amount with `8 decimals`.

- DepositLib.sol#L343-L344

```
function depositSingleToken(
    SharedVaultState.VaultState storage state,
    address user,
    uint256 usdcAmount,
    bytes calldata lifiSwapData,
    uint256 minRubOut
) external {
    // 3. Validate input amount (needs double minimum since half will be
    swapped)
    uint256 minUsdcForSingleDeposit
    = VaultConstants.getMinDepositValueUsdc(state.usdcToken) * 2;
    require(usdcAmount >= minUsdcForSingleDeposit, "USDC below minimum for
    swap");
}
```

As a result, the `minimum deposit` is interpreted as `2,000 USDC` instead of `20 USDC`.

Recommendations:

```
function depositSingleToken(
    SharedVaultState.VaultState storage state,
    address user,
    uint256 usdcAmount,
    bytes calldata lifiSwapData,
    uint256 minRubOut
) external {
+    uint64 usdcCore;
+    (usdcCore, ) = _convertToCore(state, usdcAmount, 0);

    // 3. Validate input amount (needs double minimum since half will be
    swapped)
    uint256 minUsdcForSingleDeposit
    = VaultConstants.getMinDepositValueUsdc(state.usdcToken) * 2;
-   require(usdcAmount >= minUsdcForSingleDeposit, "USDC below minimum for
        swap");
+   require(usdcCore >= minUsdcForSingleDeposit, "USDC below minimum for
        swap");

-    uint64 usdcCore;
}
```

RubiFi: Resolved with @b2da39cbf6 ... .

Zenith: Verified.

[H-6] Profits and losses should be updated at least one block after a deposit

SEVERITY: High	IMPACT: High
STATUS: Resolved	LIKELIHOOD: Medium

Target

- DepositLib.sol

Description:

When `profits` and `losses` are updated, the `core balances` of `poolCoreAddress` are used.

- ProfitLib.sol#L199-L200

```
function _getActualPoolBalances(SharedVaultState.VaultState storage state)
    internal view returns (uint256 actualUsdcBalance,
    uint256 actualRubBalance) {
    // Get raw balances from precompiles directly
    uint64 rawUsdcBalance = PrecompileLib.spotBalance(state.poolCoreAddress,
    0).total; // USDC index 0
    uint64 rawRubBalance = PrecompileLib.spotBalance(state.poolCoreAddress,
    277).total; // RUB index 277
}
```

Therefore, these `balances` must be correctly updated after a `deposit`. According to the documentation, `CoreWriter` actions are processed after the `EVM` block is built, meaning the `core balance` increases one block after a `deposit` in the `EVM`. If `deposits` and `updateSeparateAssetProfitAccumulators` occur in the same `block`, the newly `deposited` amounts are not yet included in the `core balance` of `poolCoreAddress`. However, `totalUsdcCore` and `totalRubCore` already include these new `deposits`. This mismatch is treated as a `loss` and breaks the accounting of `profits` and `losses`.

- ProfitLib.sol#L155

```
function _updateSeparateAssetProfitAccumulators(SharedVaultState.VaultState
    storage state) internal {
    (uint256 actualUsdcBalance, uint256 actualRubBalance)
    = _getActualPoolBalances(state);
    // STEP 4B: Handle LOSSES (when actualBalance < expectedBalance)
    if (actualUsdcBalance < state.totalUsdcCore) {
        uint256 usdcLosses = state.totalUsdcCore - actualUsdcBalance;
        state.totalUsdcCore = actualUsdcBalance;
        ...
    }
```

Recommendations:

```
struct VaultState {
    lastDepositBlockNumber;
}

function allocateShares(
    SharedVaultState.VaultState storage state,
    address user,
    uint256 usdcDeposit,
    uint256 rubDeposit
) internal {
    ...

    state.lastDepositBlockNumber = block.number;
}

function updateSeparateAssetProfitAccumulators(SharedVaultState.VaultState
    storage state) external returns (bool updated) {
    require(block.number > state.lastDepositBlockNumber, "");
}
```

RubiFi: Resolved with @4b5aa8e557 ... .

Zenith: Verified.

[H-7] Profits and losses should be updated before processing a deposit

SEVERITY: High	IMPACT: Medium
STATUS: Resolved	LIKELIHOOD: High

Target

- DepositLib.sol

Description:

The accumulated `profits` and `losses` up to now are updated using the `updateSeparateAssetProfitAccumulators` function.

- ProfitLib.sol#L45

```
function updateSeparateAssetProfitAccumulators(SharedVaultState.VaultState
    storage state) external returns (bool updated) {
    uint256 oldUsdcProfit = state.cumulativeUsdcProfitPerShare;
    uint256 oldRubProfit = state.cumulativeRubProfitPerShare;

    _updateSeparateAssetProfitAccumulators(state);

    return (state.cumulativeUsdcProfitPerShare ≠ oldUsdcProfit ||
            state.cumulativeRubProfitPerShare ≠ oldRubProfit);
}
```

When users `deposit` new `assets`, their `shares` are updated as well. This means that the current accumulated `profits` and `losses` should be allocated to the existing `shares` before applying the update for the new `deposit`.

Recommendations:

```
import "./ProfitLib.sol";

function depositDualTokens(
    SharedVaultState.VaultState storage state,
    address user,
    uint256 usdcAmount,
    uint256 rubAmount
) external {
    ProfitLib.updateSeparateAssetProfitAccumulators(state);
}

function depositSingleToken(
    SharedVaultState.VaultState storage state,
    address user,
    uint256 usdcAmount,
    bytes calldata lifiSwapData,
    uint256 minRubOut
) external {
    ProfitLib.updateSeparateAssetProfitAccumulators(state);
}
```

RubiFi: Resolved with `@7e2bfba36a ...`.

Zenith: Verified.

4.3 Medium Risk

A total of 6 medium risk findings were identified.

[M-1] Protocol fees should also be accounted for in totals during withdrawals

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

Target

- WithdrawalLib.sol

Description:

The logic in `completeWithdrawal()` calculates the final amounts for the withdrawal and splits the protocol fees from it. Fees and user amounts are then transferred, but only the user portion is deducted from the totals.

Recommendations:

Deduct `usdcFeeAmount` and `rubFeeAmount` from `totalUsdcCore` and `totalRubCore`, respectively.

```diff
  // 3. Update pool balances BEFORE clearing user state
- state.totalUsdcCore -= usdcAmount;
- state.totalRubCore -= rubAmount;
+ state.totalUsdcCore -= usdcAmount + usdcFeeAmount;
+ state.totalRubCore -= rubAmount + rubFeeAmount;
```

RubiFi: Resolved with `@a54a2b57e9 ...`.

Zenith: Verified.

[M-2] `emergencyCompleteWithdrawal()` should not decrement the user's gas contribution

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

Target

- WithdrawalLib.sol

Description:

In `emergencyCompleteWithdrawal()`, the implementation deducts the user's supplied gas from the total gas collected in the accumulator.

```
215:        // 6. Deduct user's gas fee from total collected
216:        state.totalGasFeesCollected -= state.userGasFees[user];
```

This is a fee paid by the user to execute the withdrawal, and should not be deducted from the collected gas owed to the protocol.

Recommendations:

Remove line 216.

RubiFi: Resolved with `@1639deeadbf6 ...`.

Zenith: Verified.

[M-3] Permanent revert in `depositSingleToken()` due to zero-amount validation

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

Target

- DepositLib.sol

Description:

The implementation of `depositSingleToken()` converts the `remainingUsdc` and `minRubOut` values separately using `_convertToCore()`.

```
        // 5.  Early validation BEFORE any transfers (cheapest possible revert)
        // Convert amounts to Core decimals for validation
        uint64 usdcCore;
        uint64 rubCore;
        (usdcCore, ) = _convertToCore(state, remainingUsdc, 0);
        (, rubCore) = _convertToCore(state, 0, minRubOut);
```

This hardcoded zero amount causes a denial of service in the `_convertToCore()` function, as the converted amount is checked to be non-zero.

```
        // Validate conversions didn't result in 0
        if (usdcCore == 0 || rubCore == 0) {
            revert CoreWriterLib.CoreWriterLib__EvmAmountTooSmall(0);
        }
```

Recommendations:

Submit the two values at once to `_convertToCore()`. Alternatively, this validation can be moved after the effective amounts (`usdcCore` and `rubCore`) are calculated.

RubiFi: Resolved with `@026c655e50 ...`.

Zenith: Verified.

[M-4] The pool cap check in depositSingleToken should use the actual deposited amount

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

Target

- DepositLib.sol

Description:

In the `depositSingleToken` function, the `minRubOut` parameter is used for `slippage` checks and does not represent the `actual deposited amount`. However, the `pool cap` and `minimum deposit` checks currently use `minRubOut` instead of the `actual deposited amount`.

- DepositLib.sol#L355

```
function depositSingleToken(
    SharedVaultState.VaultState storage state,
    address user,
    uint256 usdcAmount,
    bytes calldata lifiSwapData,
    uint256 minRubOut
) external {
    uint256 halfUsdc = usdcAmount / 2;
    uint256 remainingUsdc = usdcAmount - halfUsdc;

    uint64 usdcCore;
    uint64 rubCore;
    (usdcCore, ) = _convertToCore(state, remainingUsdc, 0);
    (, rubCore) = _convertToCore(state, 0, minRubOut);

    validateMinimumDeposit(state, usdcCore, rubCore);
    validatePoolCap(state, usdcCore, rubCore);
}
```

As a result, `deposits` can exceed the `pool cap`.

Recommendations:

```
function depositSingleToken(
    SharedVaultState.VaultState storage state,
    address user,
    uint256 usdcAmount,
    bytes calldata lifiSwapData,
    uint256 minRubOut
) external {
    uint256 halfUsdc = usdcAmount / 2;
    uint256 remainingUsdc = usdcAmount - halfUsdc;

    uint64 usdcCore;
    uint64 rubCore;
    (usdcCore, ) = _convertToCore(state, remainingUsdc, 0);
    (, rubCore) = _convertToCore(state, 0, minRubOut);

    validateMinimumDeposit(state, usdcCore, rubCore);
    validatePoolCap(state, usdcCore, rubCore);

    ...

    if (rubReceived < minRubOut) {
        revert SlippageExceeded(minRubOut, rubReceived);
    }

    (usdcCore, rubCore) = _convertToCore(state, remainingUsdc, rubReceived);

    validateMinimumDeposit(state, usdcCore, rubCore);
    validatePoolCap(state, usdcCore, rubCore);
}
```

RubiFi: Resolved with @0c2fa3bf8d ... .

Zenith: Verified.

[M-5] The completeWithdrawal function performs an incorrect check for the available amount

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

Target

- WithdrawalLib.sol

Description:

When completing a `withdrawal`, the tokens are already in the `vault`, not in `poolCoreAddress`, so they are transferred from the `vault` to the user.

- WithdrawalLib.sol#L109

```
function completeWithdrawal(
    SharedVaultState.VaultState storage state,
    SharedVaultState.VaultState storage feeState,
    address user
) external {
    // 2. Send user's withdrawal from pool to user
    if (usdcAmount > 0) {
        CoreWriterLib.spotSend(user, 0, uint64(usdcAmount));
    }
    if (rubAmount > 0) {
        CoreWriterLib.spotSend(user, 277, uint64(rubAmount));
    }
}
```

However, the available amount is incorrectly calculated based on the `balance` of `poolCoreAddress` instead of the `vault`.

- WithdrawalLib.sol#L301

```
function _calculateUserWithdrawAmountsEnhanced(
    SharedVaultState.VaultState storage state,
    SharedVaultState.VaultState storage feeState,
    address user
) internal returns (
    ...
) {
    uint256 availableUsdc = state.totalUsdcCore;  // balance of the
    poolCoreAddress
    uint256 availableRub = state.totalRubCore;

    uint256 totalUsdcNeeded = usdcAmount + usdcFeeAmount;
    if (totalUsdcNeeded > availableUsdc) {
    }
}
```

Recommendations:

```
function _calculateUserWithdrawAmountsEnhanced(
    SharedVaultState.VaultState storage state,
    SharedVaultState.VaultState storage feeState,
    address user
) internal returns (
    ...
) {
    uint256 availableUsdc = state.totalUsdcCore;
    uint256 availableRub = state.totalRubCore;
     uint64 availableUsdc =
       PrecompileLib.spotBalance(address(this), 0).total; // USDC index 0
    uint64 availableRub =

     PrecompileLib.spotBalance(address(this), 277).total; // RUB index 277

    uint256 totalUsdcNeeded = usdcAmount + usdcFeeAmount;
    if (totalUsdcNeeded > availableUsdc) {
    }
}
```

RubiFi: Resolved with @813f3a3a19 ... .

Zenith: Verified.

[M-6] Deposits should be disabled while withdrawals are pending

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

Target

- DepositLib.sol

Description:

Users can `withdraw` their funds only after `48 hours` from initiating a `withdrawal`.

- WithdrawalLib.sol#L85

```
function completeWithdrawal(
    SharedVaultState.VaultState storage state,
    SharedVaultState.VaultState storage feeState,
    address user
) external {
    require(block.timestamp >= state.pendingWithdrawals[user]
    + VaultConstants.WITHDRAWAL_DELAY, "48h withdrawal delay not completed");
}
```

Therefore, `deposits` should be disabled during this period. However, the current implementation still allows `deposits` even when a `withdrawal` is pending.

Recommendations:

```
function allocateShares(
    SharedVaultState.VaultState storage state,
    address user,
    uint256 usdcDeposit,
    uint256 rubDeposit
) internal {
     require(state.pendingWithdrawals[user] == 0,
       "Withdrawal already pending");
}
```

RubiFi: Resolved with `@062b0912e7 ...`.

Zenith: Verified.

4.4 Low Risk

A total of 2 low risk findings were identified.

[L-1] Implement emergency actions to handle the vault's spot balances

`SEVERITY`: Low	`IMPACT`: Low
`STATUS`: Resolved	`LIKELIHOOD`: Low

Target

- RUB_USDC_Vault.sol

Description:

CORE actions queued from the EVM side require HYPE native balance when these are eventually executed.

In deposits, while `depositDualTokens()` and `depositSingleToken()` ensure a certain amount of gas in the CORE account, many deposits could be piled up and potentially require more than this threshold. Note that this balance would be consumed when the actions are executed, and not when queued.

For the withdrawal case, there isn't a matching validation, so presumably this is an operator's responsibility to check for enough gas, since this is a permissioned action.

In any case, lack of gas would cause a failed transaction and lead to stuck assets in the vault account.

Recommendations:

Provide emergency actions to handle spot balances for the vault account so an operator can eventually rescue funds or retry the failed transaction.

RubiFi: Resolved with `@285131dc06 ...`.

Zenith: Verified.

[L-2] Validate user has not claimed aidrop before assigning an amount

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

Target

- RubAirdrop.sol

Description:

The `setClaimableAmounts()` function can be used to adjust the user's claimable amount. Since users can claim at most once, adjusting the amount after the user has claimed would lead to an unclaimable assignment.

Recommendations:

Validate the user has not already claimed the airdrop.

```
function setClaimableAmounts(
    address[] calldata users,
    uint256[] calldata amounts
) external onlyOwner {
    require(users.length == amounts.length, "Length mismatch");

    for (uint256 i = 0; i < users.length; i++) {
        if (users[i] == address(0)) revert InvalidAddress();

          if (hasClaimed[users[i]]) revert AlreadyClaimed();

        // Update total allocated
        totalAllocated = totalAllocated - claimableAmount[users[i]]
    + amounts[i];

        // Set new amount
        claimableAmount[users[i]] = amounts[i];

        emit ClaimableAmountSet(users[i], amounts[i]);
    }
}
```

RubiFi: Resolved with `@6adaa4d3d1 ...`.

Zenith: Verified.

4.5 Informational

A total of 3 informational findings were identified.

[I-1] The admin should have the ability to transfer funds from EVM to the core

`SEVERITY`: Informational	`IMPACT`: Informational
`STATUS`: Resolved	`LIKELIHOOD`: Low

Target

- RUB_USDC_Vault.sol

Description:

Transfers from the `EVM` to `Core` can fail. The `admin` should be able to transfer any remaining tokens from the `EVM` to the `Core` or recover them.

Recommendations:

Add a new function to transfer from the `EVM` to the `Core` or `recover` function.

RubiFi: Resolved with `@3be4899d8c ...`.

Zenith: Verified.

[I-2] `updateLifiSelector()` should not restrict zero selector

SEVERITY: Informational	IMPACT: Informational
STATUS: Resolved	LIKELIHOOD: Low

Target

- RUB_USDC_Vault.sol

Description:

The `updateLifiSelector()` function validates that the new selector is not zero. However, this is a valid value a selector could have, as it can be anything in the bytes4 domain.

```
function updateLifiSelector(bytes4 newSelector) external onlyOwner {
    require(newSelector ≠ bytes4(0), "Invalid selector");
    bytes4 oldSelector = _vaultState.lifiSwapSelector;
    _vaultState.lifiSwapSelector = newSelector;
    emit LifiSelectorUpdated(oldSelector, newSelector);
}
```

Recommendations:

Remove the restriction.

RubiFi: Resolved with `@3d9d173ffa ...`.

Zenith: Verified.

[I-3] Duplicate setter

SEVERITY: Informational	IMPACT: Informational
STATUS: Resolved	LIKELIHOOD: Low

Target

- RUB_USDC_Vault.sol

Description:

The `updateBackend()` and `setBackend()` functions implement the same funcionality.

Recommendations:

Remove one of these setters.

RubiFi: Resolved with `@a1dce6e4fca ...`.

Zenith: Verified.