BitChill Security Review

Ivan Fitro

April 29th, 2025

Contents

1 Introduction

Ivan Fitro conducted a time-limited security assessment of the **BitChill** protocol, concentrating on the security properties and implementation of its smart contracts.

2 Disclaimer

A smart contract security review cannot ensure the total absence of vulnerabilities. The assessment is inherently limited by time, resources, and available expertise, aiming to identify as many issues as possible within those constraints. While I strive to uncover potential risks, I cannot guarantee 100% security or that any issues will necessarily be found. It is strongly advised to follow up with additional security reviews, implement bug bounty programs, and maintain continuous on-chain monitoring.

3 About Ivan Fitro

Ivan Fitro, also known as **Fitro**, is an independent security researcher specializing in smart contracts. With a track record of uncovering vulnerabilities across various blockchain protocols, he is dedicated to enhancing the ecosystem through diligent security research and thorough audits. You can explore his past work here or connect with him on Twitter @FitroIvan.

4 About BitChill

BitChill is a protocol built on the Rootstock blockchain, focused on facilitating dollar-cost averaging (DCA) purchases of rBTC. It leverages various stablecoins (ERC-20 tokens), which are deposited into lending protocols to earn interest while waiting to execute scheduled rBTC purchases.

5 Severity classification

Severity	Impact: High	Impact: Medium	Impact: Low
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

Issues that do not compromise the protocol but aim to improve it are marked as Informational.

Impact - the potential impact of a successful attack includes technical failures, financial losses, and reputational harm

Likelihood - the probability that a given vulnerability is detected and subsequently exploited.

Severity - the overall criticality of the risk

6 Security Assessment Summary

***review commit hash* - fd73092c99234713bea57cc839d189552f21edee**

Scope

The following smart contracts were in scope of the audit:

- `AdminOperations.sol`
- `DcaManager.sol`
- `DcaManagerAccessControl.sol`
- `FeeHandler.sol`
- `PurchaseMoc.sol`
- `SovrynDocHandler.sol`
- `SovrynDocHandlerMoc.sol`
- `TokenHandler.sol`
- `TokenLending.sol`
- `TropykusDocHandler.sol`
- `TropykusDocHandlerMoc.sol`

7 Executive Summary

Over the course of the security review, Fitro engaged with BitChill toreview BitChill. In this period of time a total of **9** issues were uncovered.

Protocol Summary	
Protocol Name	BitChill
Date	April 29th, 2025

Findings Count	
Severity	Amount
Medium	3
Low	4
Informational	2
Total Findings	**9**

Audit Findings Summary			
ID	**Title**	**Severity**	**Status**
[M-01]	DcaManager.sol :: withdrawRbtcFromTokenHandler(): if the receiver is a smart contract that cannot accept rBTC, the funds will become permanently stuck in the contract	Medium	Fixed
[M-02]	DcaManager.sol :: withdrawAllAccumulatedRbtc() will revert if two different tokens use the same lendingProtocolIndex and rBTC has already been withdrawn for one of them or if the schedule for the token is removed	Medium	Fixed
[M-03]	DcaManager.sol :: buyRbtc() can be frontrun to pay less fee	Medium	Fixed
[L-01]	DcaManager.sol :: deleteDcaSchedule(): if the schedule has a zero balance, it can't be deleted	Low	Fixed
[L-02]	DcaManager.sol :: createDcaSchedule(): same scheduleId can be generated for different schedules	Low	Fixed
[L-03]	DcaManager.sol :: deleteDcaSchedule(): if a user creates too many schedules, it can lead to a DoS, preventing them from being removed	Low	Fixed
[L-04]	PurchaseMoc.sol :: withdrawAccumulatedRbtc() allows anyone to withdraw the accumulated rBTC on behalf of any user	Low	Fixed
[I-01]	TokenHandler.sol :: depositToken() does not need to explicitly check the user's allowance, as transferFrom will handle it and revert if insufficient	Info	Fixed
[I-02]	Replace <= with == for zero-value checks	Info	Fixed

8 Findings

8.1 Medium Findings

[M-01] DcaManager.sol :: withdrawRbtcFromTokenHandler() if the receiver is a smart contract that cannot accept rBTC, the funds will become permanently stuck in the contract

Description

Anyone, including smart contracts, can call `createDcaSchedule()`. Since the deposit token is an ERC20, this poses no issue, any address can send and receive ERC20 tokens.

However, the issue arises because the protocol uses the deposited ERC20 tokens to purchase rBTC, Rootstock's native token. The rBTC is then sent to the deposit address using `withdrawAccumulatedRbtc()`, which transfers native tokens. If the deposit address is a smart contract that doesn't support receiving native tokens, this can cause the transaction to fail.

```
function withdrawAccumulatedRbtc(address user) external virtual override {
        uint256 rbtcBalance = s_usersAccumulatedRbtc[user];
        if (rbtcBalance == 0) revert PurchaseRbtc__NoAccumulatedRbtcToWithdraw();

        s_usersAccumulatedRbtc[user] = 0;
        // Transfer RBTC from this contract back to the user
        (bool sent,) = user.call{value: rbtcBalance}("");
        if (!sent) revert PurchaseRbtc__rBtcWithdrawalFailed();
        emit PurchaseRbtc__rBtcWithdrawn(user, rbtcBalance);
    }
```

If the `user` is a smart contract that does not implement a function to receive rBTC, the transaction will revert. Since the user of a schedule cannot be changed and there is no function to withdraw rBTC from the contract, any rBTC sent to such a contract would be permanently stuck, effectively lost.

Recommendations

Implement a function that allows the contract owner to withdraw any rBTC that becomes stuck.

```
function withdrawStuckrBTC(address receiver, uint256 amount) external onlyOwner {
        (bool sent,) = receiver.call{value: amount}("");
        if (!sent) revert PurchaseRbtc__rBtcWithdrawalFailed()
    }
```

[M-02] DcaManager.sol :: withdrawAllAccumulatedRbtc() will revert if two different tokens use the same lendingProtocolIndex and rBTC has already been withdrawn for one of them using withdrawRbtcFromTokenHandler() or if the schedule for the token is removed

Description

`withdrawRbtcFromTokenHandler()` is used to withdraw the accumulated rBTC for a specific token and lending protocol.

`withdrawAllAccumulatedRbtc()` allows the user to withdraw rBTC from all tokens they've deposited into a specific lending protocol.

However, there's a problem: if a user first withdraws rBTC using `withdrawRbtcFromTokenHandler()` for one of the tokens, and then calls `withdrawAllAccumulatedRbtc()` for the same lending protocol (which includes that token), the transaction will revert. This happens because `s_usersAccumulatedRbtc` is already zero for the token that was previously withdrawn.

```
function withdrawAccumulatedRbtc(address user) external virtual override {
        uint256 rbtcBalance = s_usersAccumulatedRbtc[user];
@>      if (rbtcBalance == 0) revert PurchaseRbtc__NoAccumulatedRbtcToWithdraw();

        s_usersAccumulatedRbtc[user] = 0;
        // Transfer RBTC from this contract back to the user
        (bool sent,) = user.call{value: rbtcBalance}("");
        if (!sent) revert PurchaseRbtc__rBtcWithdrawalFailed();
        emit PurchaseRbtc__rBtcWithdrawn(user, rbtcBalance);
    }
```

This will render the function ineffective and cause it to behave incorrectly.

Let's walk through an example to illustrate the issue more clearly:

1. Bob deposits into 10 different tokens using `lendingProtocolIndex = 1`.
2. Later, Bob calls `withdrawRbtcFromTokenHandler()` with `token = token0` and `lendingProtocolIndex = 1`, successfully withdrawing the accumulated rBTC for that specific token.
3. Then, Bob calls `withdrawAllAccumulatedRbtc()` with `lendingProtocolIndexes = [1]` to withdraw rBTC from all tokens under that protocol.
4. The transaction reverts because `token0` no longer has any accumulated rBTC, resulting in a failed execution.

This can also occur if Bob deletes the schedule for a token he no longer wants to buy with, but `s_usersDepositedTokens` does not remove the token when the user has no remaining balance in any schedule and all the rBTC is claimed. As a result, the transaction will always revert, rendering the function unusable.

Recommendations

To resolve the issue, add a check to ensure that `rbtcBalance != 0` before calling `withdrawAccumulatedRbtc()`.

```
function withdrawAllAccumulatedRbtc(uint256[] calldata lendingProtocolIndexes)
    external override nonReentrant {
        address[] memory depositedTokens = s_usersDepositedTokens[msg.sender];
        for (uint256 i; i < depositedTokens.length; ++i) {
            for (uint256 j; j < lendingProtocolIndexes.length; ++j) {

+               IPurchaseRbtc purchase = IPurchaseRbtc(address(_handler(
    depositedTokens[i], lendingProtocolIndexes[j])));

+               if (purchase.s_usersAccumulatedRbtc[msg.sender] == 0) continue;

-               IPurchaseRbtc(address(_handler(depositedTokens[i], -
    lendingProtocolIndexes[j]))).withdrawAccumulatedRbtc(
-                   msg.sender
-               );

+              purchase.withdrawAccumulatedRbtc(msg.sender);
            }
        }
    }
```

To achieve this, the `s_usersAccumulatedRbtc` mapping in `PurchaseMoc.sol` must be declared as `public`.

[M-03] DcaManager.sol :: buyRbtc() can be frontrun to pay less fee

Description

`buyRbtc()` allows a user to purchase rBTC, with a fee applied to each transaction, calculated using the `_calculateFee()`.

```
function _calculateFee(uint256 purchaseAmount, uint256 purchasePeriod) internal view
    returns (uint256) {

@>      uint256 annualSpending = (purchaseAmount * 365 days) / purchasePeriod;
        uint256 feeRate;

        if (annualSpending >= s_maxAnnualAmount) {
            feeRate = s_minFeeRate;
        } else if (annualSpending <= s_minAnnualAmount) {
            feeRate = s_maxFeeRate;
        } else {
            // Calculate the linear fee rate
            feeRate = s_maxFeeRate
                - ((annualSpending - s_minAnnualAmount) * (s_maxFeeRate -
                    s_minFeeRate))
                    / (s_maxAnnualAmount - s_minAnnualAmount);
        }
        return purchaseAmount * feeRate / FEE_PERCENTAGE_DIVISOR;
    }
```

The `annualSpending` is calculated, and if it exceeds `s_maxAnnualAmount`, the `s_minFeeRate` is applied to determine the fee.

However, a user could **frontrun** the `buyRbtc()` transaction by setting a smaller `purchasePeriod` through `setPurchasePeriod()`. This would artificially increase the `annualSpending` allowing the user to trigger a lower fee rate and pay a lower fee.

To better illustrate the issue, let's consider the following example. Assume:

- `s_maxAnnualAmount = 30_000`
- `s_minPurchasePeriod = 1 days`
- `s_minAnnualAmount = 5_000`

1. Bob creates a schedule with:
    - `purchaseAmount = 100`
    - `purchasePeriod = 10 days`
2. 10 days pass, and `buyRbtc()` is called.

3. Bob notices the transaction and calls `setPurchasePeriod()` to reduce the period to 1 days (this will not cause revert in `_rBtcPurchaseChecksEffects()` because 1 days is less than 10 days, and the time elapsed will be enought to complete the purchase).
4. The fee is calculated based on the new `purchasePeriod`:
    - `annualSpending = purchaseAmount * 365_days / purchasePeriod = 100 * 365 / 1 = 36_500`
    - Since `annualSpending > s_maxAnnualAmount`, the `s_minFeeRate` is applied to calculate the fees.

    However, the correct calculation of `annualSpending` should be:
    - `annualSpending = purchaseAmount * 365_days / purchasePeriod = 100 * 365 / 10 = 3_650`
    - Since `annualSpending < s_minAnnualAmount`, the `s_maxFeeRate` should be applied.
5. Bob backruns the transaction by calling `setPurchasePeriod()` again, resetting it back to 10 days because he intends to buy rBTC only every 10 days.
6. As a result, Bob pays a reduced fee, exploiting the system and causing a loss to the protocol.

Proof of Concept

To run the POC, copy the test below in `RbtcPurchaseTest.t.sol`, `DcaDappTest.t.sol` and modify the `setUp()`, specifically, set the `purchasePeriod` of the created schedule for USER to 20 days.

RbtcPurchaseTest.t.sol

```
function testFrontrunSinglePurchase() external {
        super.makeFrontrunSinglePurchase();
    }
```

DcaDappTest.t.sol

```
function makeFrontrunSinglePurchase() internal {

        vm.startPrank(USER);
        uint256 docBalanceBeforePurchase = dcaManager.getScheduleTokenBalance(
            address(docToken), SCHEDULE_INDEX);
        uint256 rbtcBalanceBeforePurchase = IPurchaseRbtc(address(docHandler)).
            getAccumulatedRbtcBalance();
        IDcaManager.DcaDetails[] memory dcaDetails = dcaManager.getMyDcaSchedules(
            address(docToken));
        vm.stopPrank();

        uint256 realFee = feeCalculator.calculateFee(DOC_TO_SPEND, 20 days);
        uint256 exploitedFee = feeCalculator.calculateFee(DOC_TO_SPEND, 1 days);

        console2.log("realFee:", realFee);
        console2.log("exploitedFee:", exploitedFee);
```

```
        console2.log("Loss:", realFee - exploitedFee);

        //frontrun to set to 1 day the purcahsed to reduce the fee paid
        vm.prank(USER);
        dcaManager.setPurchasePeriod(address(docToken), SCHEDULE_INDEX, 1 days);

        vm.prank(SWAPPER);
        dcaManager.buyRbtc(USER, address(docToken), SCHEDULE_INDEX, dcaDetails[
            SCHEDULE_INDEX].scheduleId);

        //backrun to set to 20 days the purachase
        vm.prank(USER);
        dcaManager.setPurchasePeriod(address(docToken), SCHEDULE_INDEX, 20 days);

        vm.startPrank(USER);
        uint256 docBalanceAfterPurchase = dcaManager.getScheduleTokenBalance(address
            (docToken), SCHEDULE_INDEX);
        uint256 rbtcBalanceAfterPurchase = IPurchaseRbtc(address(docHandler)).
            getAccumulatedRbtcBalance();
        vm.stopPrank();

        uint256 balanceFeeCollector = docToken.balanceOf(address(FEE_COLLECTOR));
        assertEq(balanceFeeCollector, exploitedFee);
        assertNotEq(balanceFeeCollector, realFee);
    }
```

```
--------LOGS--------
realFee: 3960000000000000000
exploitedFee: 25600000000000000000
Loss: 14000000000000000000
```

Recomendations

One solution is to allow users to only increase the purchase period but not decrease it.

8.2 Low Findings

[L-01] DcaManager.sol :: deleteDcaSchedule() if the schedule has a zero balance can't be deleted

Description

`deleteDcaSchedule()` is used to remove existing schedules.

```
function deleteDcaSchedule(address token, bytes32 scheduleId) external nonReentrant
    {

        DcaDetails[] storage schedules = s_dcaSchedules[msg.sender][token];

        uint256 scheduleIndex;
        bool found = false;

        // Find the schedule by scheduleId
        for (uint256 i = 0; i < schedules.length; i++) {
            if (schedules[i].scheduleId == scheduleId) {
                scheduleIndex = i;
                found = true;
                break;
            }
        }

        if (!found) revert DcaManager__InexistentScheduleId();

        // Store the balance and scheduleId before modifying the array
        uint256 tokenBalance = schedules[scheduleIndex].tokenBalance;
        uint256 lendingProtocolIndex = schedules[scheduleIndex].lendingProtocolIndex
            ;

        // Remove the schedule
        uint256 lastIndex = schedules.length - 1;
        if (scheduleIndex != lastIndex) {
            // Overwrite the schedule getting deleted with the one in the last index
            schedules[scheduleIndex] = schedules[lastIndex];
        }
        // Remove the last schedule
        schedules.pop();

        // Withdraw all balance
@>      _handler(token, lendingProtocolIndex).withdrawToken(msg.sender, tokenBalance
    );

        // Emit event
        emit DcaManager__DcaScheduleDeleted(msg.sender, token, scheduleId,
            tokenBalance);
    }
```

The final step involves transferring the remaining balance from the schedule back to the user. However, if the schedule has a zero balance and is using the Sovryn handler, the transaction reverts, preventing the schedule from being deleted.

```
function _redeemDoc(address user, uint256 docToRedeem, uint256 exchangeRate, address
    docRecipient)
        internal
        virtual
        returns (uint256)
    {
        uint256 usersIsusdBalance = s_iSusdBalances[user];
        uint256 iSusdToRepay = _docToLendingToken(docToRedeem, exchangeRate);
        if (iSusdToRepay > usersIsusdBalance) {
            emit TokenLending__AmountToRepayAdjusted(user, iSusdToRepay,
                usersIsusdBalance);
            iSusdToRepay = usersIsusdBalance;
        }
        s_iSusdBalances[user] -= iSusdToRepay;
        //redeem DOC to buy rBTC
        uint256 docRedeemed = i_iSusdToken.burn(docRecipient, iSusdToRepay);
        if (docRedeemed == 0) revert SovrynDocLending__RedeemUnderlyingFailed();
        emit TokenLending__SuccessfulDocRedemption(user, docRedeemed, iSusdToRepay);
        return docRedeemed;
    }
```

The transaction reverts with the `SovrynDocLending__RedeemUnderlyingFailed` custom error because the amount of DOC to redeem is zero. As a result, schedules with a zero balance cannot be deleted.

Currently, users must deposit funds into the schedule before being able to delete it, which is not an ideal solution.

Proof of Concept

To reproduce the issue, copy the following POC into `DcaScheduleTest.t.sol`.

```
function testDeleteDcaScheduleRevert() external {
        vm.startPrank(USER);
        docToken.approve(address(docHandler), DOC_TO_DEPOSIT * 5);
        // Create two schedules in different blocks
        bytes32 scheduleId =
            keccak256(abi.encodePacked(USER, block.timestamp, dcaManager.
                getMyDcaSchedules(address(docToken)).length));
        dcaManager.createDcaSchedule(
            address(docToken), DOC_TO_DEPOSIT * 2, DOC_TO_SPEND, MIN_PURCHASE_PERIOD
                , s_lendingProtocolIndex
        );

        //withdraw all the balance
        dcaManager.withdrawToken(address(docToken), 1, DOC_TO_DEPOSIT * 2);

        // Delete one
        //revert with SovrynDocLending__RedeemUnderlyingFailed() custom error
```

```
        vm.expectRevert();
        dcaManager.deleteDcaSchedule(address(docToken), scheduleId);

        vm.stopPrank();
    }
```

Recommendations

To resolve the issue, add a check to ensure the remaining balance is greater than zero.

```
function deleteDcaSchedule(address token, bytes32 scheduleId) external nonReentrant
    {

        DcaDetails[] storage schedules = s_dcaSchedules[msg.sender][token];

        uint256 scheduleIndex;
        bool found = false;

        // Find the schedule by scheduleId
        for (uint256 i = 0; i < schedules.length; i++) {
            if (schedules[i].scheduleId == scheduleId) {
                scheduleIndex = i;
                found = true;
                break;
            }
        }

        if (!found) revert DcaManager__InexistentScheduleId();

        // Store the balance and scheduleId before modifying the array
        uint256 tokenBalance = schedules[scheduleIndex].tokenBalance;
        uint256 lendingProtocolIndex = schedules[scheduleIndex].lendingProtocolIndex
            ;

        // Remove the schedule
        uint256 lastIndex = schedules.length - 1;
        if (scheduleIndex != lastIndex) {
            // Overwrite the schedule getting deleted with the one in the last index
            schedules[scheduleIndex] = schedules[lastIndex];
        }
        // Remove the last schedule
        schedules.pop();

        // Withdraw all balance
-       _handler(token, lendingProtocolIndex).withdrawToken(msg.sender, tokenBalance)
    ;
+       if(tokenBalance > 0) {
+         _handler(token, lendingProtocolIndex).withdrawToken(msg.sender,
    tokenBalance);
+       }

        // Emit event
        emit DcaManager__DcaScheduleDeleted(msg.sender, token, scheduleId,
            tokenBalance);
```

```
    }
```

[L-02] DcaManager.sol :: createDcaSchedule() same scheduleId can be generated for differents schedules

Description

`createDcaSchedule()` is used to set up different DCA strategies.

```
function createDcaSchedule(
        address token,
        uint256 depositAmount,
        uint256 purchaseAmount,
        uint256 purchasePeriod,
        uint256 lendingProtocolIndex
    ) external override {
        _validatePurchasePeriod(purchasePeriod);
        _validateDeposit(token, depositAmount);
        _handler(token, lendingProtocolIndex).depositToken(msg.sender, depositAmount
            );

@>      bytes32 scheduleId =
            keccak256(abi.encodePacked(msg.sender, block.timestamp, s_dcaSchedules[
                msg.sender][token].length));

        DcaDetails memory dcaSchedule = DcaDetails(
            depositAmount,
            purchaseAmount,
            purchasePeriod,
            0, // lastPurchaseTimestamp
            scheduleId,
            lendingProtocolIndex
        );

        _validatePurchaseAmount(token, purchaseAmount, dcaSchedule.tokenBalance,
            dcaSchedule.lendingProtocolIndex);

        s_dcaSchedules[msg.sender][token].push(dcaSchedule);
        emit DcaManager__DcaScheduleCreated(
            msg.sender, token, scheduleId, depositAmount, purchaseAmount,
                purchasePeriod
        );
    }
```

`scheduleId` is generated using multiple parameters, but there's a potential issue when multiple schedules are created by a smart contract in a single transaction with different tokens. Here's why collisions can happen:

- `msg.sender` will be the same (the smart contract).
- `block.timestamp` will also be the same, since the calls occur within the same transaction.
- `s_dcaSchedules[msg.sender][token].length` will be `0` for all, assuming no prior schedules exist for those tokens.

This means multiple schedules could end up with the same `scheduleId`, which can lead to tracking issues off-chain and potential logic errors in the system.

Recommendations

To solve this issue, it's better to use a user-specific nonce instead of relying on `s_dcaSchedules[msg.sender][token].length`. This ensures uniqueness across schedule creations, even within the same transaction.

To implement this, you can introduce a nonce mapping:

```
mapping(address user => uint256 nonce) public nonces;
```

```
function createDcaSchedule(
        address token,
        uint256 depositAmount,
        uint256 purchaseAmount,
        uint256 purchasePeriod,
        uint256 lendingProtocolIndex
    ) external override {
        _validatePurchasePeriod(purchasePeriod);
        _validateDeposit(token, depositAmount);
        _handler(token, lendingProtocolIndex).depositToken(msg.sender, depositAmount
            );

-        bytes32 scheduleId =
-            keccak256(abi.encodePacked(msg.sender, block.timestamp, s_dcaSchedules[
    msg.sender][token].length));

+        bytes32 scheduleId =
+            keccak256(abi.encodePacked(msg.sender, block.timestamp, nonces[msg.
    sender]));

+        nonces[msg.sender]++;

        DcaDetails memory dcaSchedule = DcaDetails(
            depositAmount,
            purchaseAmount,
            purchasePeriod,
            0, // lastPurchaseTimestamp
            scheduleId,
            lendingProtocolIndex
        );

        _validatePurchaseAmount(token, purchaseAmount, dcaSchedule.tokenBalance,
            dcaSchedule.lendingProtocolIndex);

        s_dcaSchedules[msg.sender][token].push(dcaSchedule);
        emit DcaManager__DcaScheduleCreated(
            msg.sender, token, scheduleId, depositAmount, purchaseAmount,
                purchasePeriod
        );
    }
```

[L-03] DcaManager.sol :: deleteDcaSchedule() if a user creates too many schedules it can lead to a DOS preventing the schedules from being removed

Description

`deleteDcaSchedule()` is used to remove a user's DCA schedule.

```
function deleteDcaSchedule(address token, bytes32 scheduleId) external nonReentrant
    {

       DcaDetails[] storage schedules = s_dcaSchedules[msg.sender][token];

       uint256 scheduleIndex;
       bool found = false;

       // Find the schedule by scheduleId
       for (uint256 i = 0; i < schedules.length; i++) {
           if (schedules[i].scheduleId == scheduleId) {
               scheduleIndex = i;
               found = true;
               break;
           }
       }

     /// code...
```

As you can see, it iterates through all the schedules a user has for a given token. The issue arises when a user has too many schedules and attempts to remove the last one. The transaction may exceed the block gas limit, causing it to revert. This forces the user to first remove earlier schedules to shorten the array, enabling them to remove the last one.

Moreover, this can affect `withdrawAllAccumulatedInterest()`, as it iterates over all schedules created for a specific token, potentially making it impossible to claim the generated interest.

Recommendations

To solve this issue, enforce a maximum limit on the number of schedules a user can create for each token.

[L-04] PurchaseMoc.sol :: withdrawAccumulatedRbtc() allows anyone to withdraw the accumulated rBTC on behalf of any user

Description

`withdrawRbtcFromTokenHandler()` and `withdrawAllAccumulatedRbtc()` transfer the accumulated rBTC purchased by the user to their address.

```
function withdrawRbtcFromTokenHandler(address token, uint256 lendingProtocolIndex)
    external override nonReentrant {
        IPurchaseRbtc(address(_handler(token, lendingProtocolIndex))).
            withdrawAccumulatedRbtc(msg.sender);
    }
```

As you can see, `msg.sender` is used because only the user can claim their rewards. However, if we inspect the implementation of `withdrawAccumulatedRbtc()`.

```
function withdrawAccumulatedRbtc(address user) external virtual override {
        uint256 rbtcBalance = s_usersAccumulatedRbtc[user];
        if (rbtcBalance == 0) revert PurchaseRbtc__NoAccumulatedRbtcToWithdraw();

        s_usersAccumulatedRbtc[user] = 0;
        // Transfer RBTC from this contract back to the user
        (bool sent,) = user.call{value: rbtcBalance}("");
        if (!sent) revert PurchaseRbtc__rBtcWithdrawalFailed();
        emit PurchaseRbtc__rBtcWithdrawn(user, rbtcBalance);
    }
```

As you can see, anyone can claim rBTC on behalf of any user, which is not the intended behavior.

Moreover, this could cause confusion, as users may see rBTC sent to their wallets without having performed any action themselves.

Recommendations

To resolve the issue, add the `onlyDcaManager` modifier to restrict access so that only the DcaManager can call the function.

```
-function withdrawAccumulatedRbtc(address user) external virtual override {
+function withdrawAccumulatedRbtc(address user) external virtual override
    onlyDcaManager  {
        uint256 rbtcBalance = s_usersAccumulatedRbtc[user];
        if (rbtcBalance == 0) revert PurchaseRbtc__NoAccumulatedRbtcToWithdraw();

        s_usersAccumulatedRbtc[user] = 0;
        // Transfer RBTC from this contract back to the user
        (bool sent,) = user.call{value: rbtcBalance}("");
        if (!sent) revert PurchaseRbtc__rBtcWithdrawalFailed();
        emit PurchaseRbtc__rBtcWithdrawn(user, rbtcBalance);
    }
```

8.3 Informational Findings

[I-01] TokenHandler.sol :: depositToken() does not need to explicitly check the user's allowance, as transferFrom will handle it and revert if insufficient

Description

`depositToken()` verifies that the contract has sufficient allowance to spend tokens on behalf of the user.

```
function depositToken(address user, uint256 depositAmount) public virtual override
    onlyDcaManager {
        if (i_stableToken.allowance(user, address(this)) < depositAmount) {
            revert TokenHandler__InsufficientTokenAllowance(address(i_stableToken));
        }
        i_stableToken.safeTransferFrom(user, address(this), depositAmount);
        emit TokenHandler__TokenDeposited(address(i_stableToken), user,
            depositAmount);
    }
```

This check is unnecessary because `transferFrom` will automatically revert if the contract doesn't have sufficient allowance.

Recommendations

Remove the allowance check to reduce gas costs on each transaction.

```
function depositToken(address user, uint256 depositAmount) public virtual override
    onlyDcaManager {
-       if (i_stableToken.allowance(user, address(this)) < depositAmount) {
-           revert TokenHandler__InsufficientTokenAllowance(address(i_stableToken));
-       }
        i_stableToken.safeTransferFrom(user, address(this), depositAmount);
        emit TokenHandler__TokenDeposited(address(i_stableToken), user,
            depositAmount);
    }
```

[I-02] DcaManager.sol :: Replace <= with == for zero-value checks

Description

In both `_validateDeposit()` and `_withdrawToken()`, the code checks that the amount is not zero. However, it currently uses `<= 0` for this validation. Since the value being checked is a `uint256` (which cannot be negative), using `== 0` is more appropriate and more gas-efficient. Replacing `<= 0` with `== 0` will reduce gas usage slightly on each transaction.

```
function _validateDeposit(address token, uint256 depositAmount) internal {

@>      if (depositAmount <= 0) revert
    DcaManager__DepositAmountMustBeGreaterThanZero();

        if (!s_tokenIsDeposited[msg.sender][token]) {
            s_tokenIsDeposited[msg.sender][token] = true;
            s_usersDepositedTokens[msg.sender].push(token);
        }
        if (!s_userRegistered[msg.sender]) {
            s_userRegistered[msg.sender] = true;
            s_users.push(msg.sender);
        }
    }
```

```
function _withdrawToken(address token, uint256 scheduleIndex, uint256
    withdrawalAmount) internal {

@>      if (withdrawalAmount <= 0) revert
    DcaManager__WithdrawalAmountMustBeGreaterThanZero();
        uint256 tokenBalance = s_dcaSchedules[msg.sender][token][scheduleIndex].
            tokenBalance;
        if (withdrawalAmount > tokenBalance) {
            revert DcaManager__WithdrawalAmountExceedsBalance(token,
                withdrawalAmount, tokenBalance);
        }
        DcaDetails storage dcaSchedule = s_dcaSchedules[msg.sender][token][
            scheduleIndex];
        dcaSchedule.tokenBalance -= withdrawalAmount;
        _handler(token, dcaSchedule.lendingProtocolIndex).withdrawToken(msg.sender,
            withdrawalAmount);
        emit DcaManager__TokenWithdrawn(msg.sender, token, withdrawalAmount);
        emit DcaManager__TokenBalanceUpdated(
            token, dcaSchedule.scheduleId, s_dcaSchedules[msg.sender][token][
                scheduleIndex].tokenBalance
        );
    }
```

Recommendations

Replace `<=` with `==`.