Scanned Code Report

AUDITAGENT

Powered by NETHERMIND SECURITY

Code Info

Basic Scan

Scan ID 1	Date September 12, 2025
Organization RozoAI	Repository rozo-rewards-miniapp
Branch main	Commit Hash 01f470ca...99eb43f4

Contracts in scope

contracts/CashbackRewards.sol

Code Statistics

Findings 11	Contracts Scanned 1	Lines of Code 150

Findings Summary

Total Findings

- High Risk (0)
- Medium Risk (0)
- Low Risk (3)
- Info (3)
- Best Practices (5)

Code Summary

The CashbackRewards protocol implements a decentralized cashback and rewards system using USDC as the payment token. It facilitates transactions between users and registered merchants, rewarding users with points for their purchases.

The system is managed by an owner who configures merchants, their specific cashback rates, and a global platform fee. When a user initiates a purchase, the contract pulls the required USDC from their wallet. It then automatically calculates and deducts the platform fee, determines the cashback amount based on the merchant's rate, and transfers the net amount to the merchant's destination address.

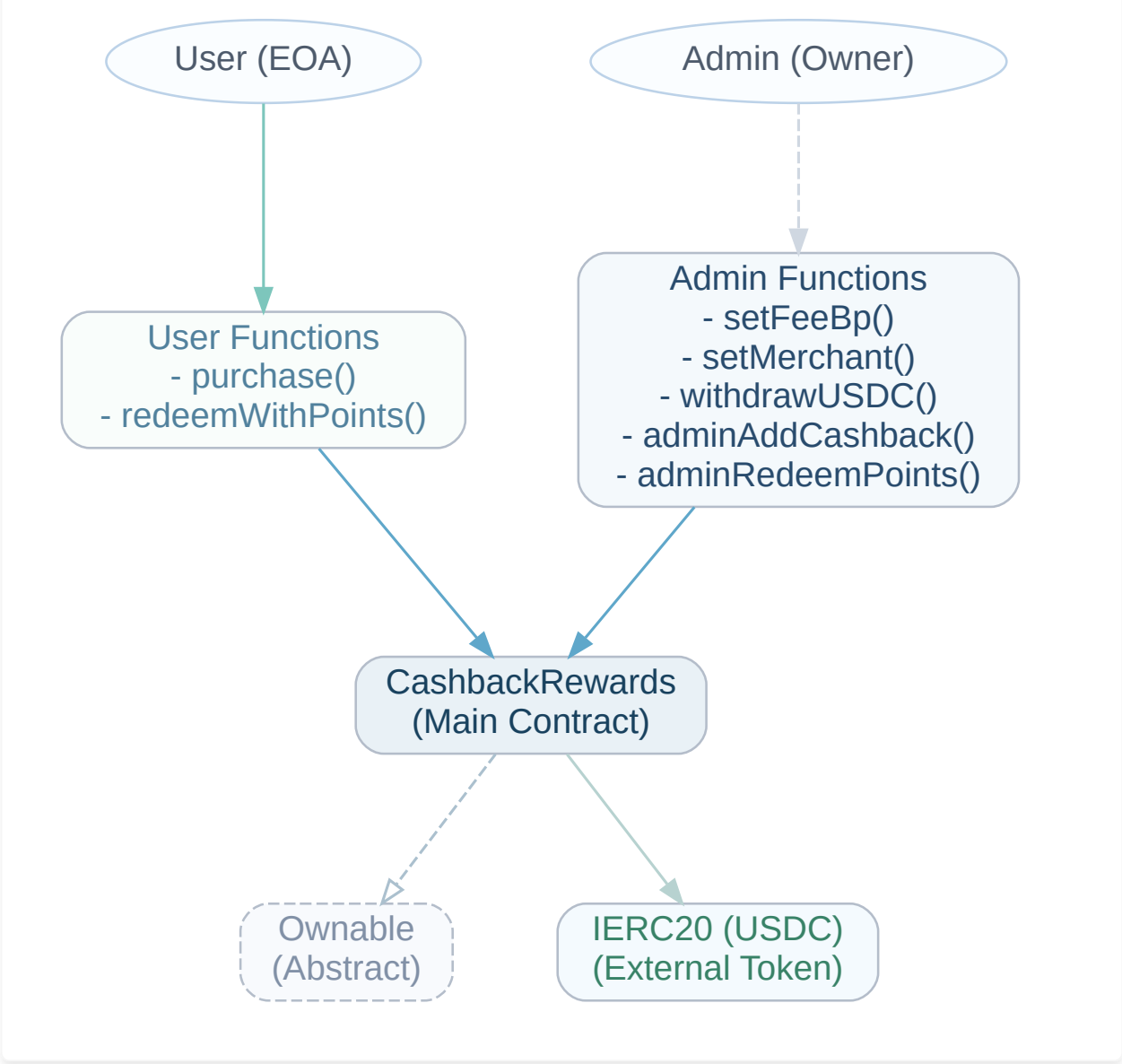
The calculated cashback is converted into points and credited to the user's balance within the contract. Users can accumulate these points and later redeem them to pay for items from any registered merchant. When points are redeemed, the contract pays the merchant the equivalent value in USDC from its own balance.

Entry Points and Actors

The primary actors interacting with the protocol are Users.

- **Users:**
- `purchase(uint256 merchantId, uint256 amount)`: Allows a user to make a purchase from a registered merchant using USDC, which earns them cashback points.
- `redeemWithPoints(uint256 merchantId, uint256 itemPrice)`: Enables a user to use their accumulated points to pay for an item, with the contract transferring the equivalent USDC value to the merchant.

Code Diagram



1 of 11 Findings

contracts/CashbackRewards.sol

No token recovery mechanism

Low Risk

If other ERC20 tokens are accidentally sent to the contract, there's no way to recover them. A recovery function would help ensure that tokens other than USDC aren't permanently stuck in the contract.

```
// Missing function to recover non-USDC tokens
function recoverToken(IERC20 token, uint256 amount) external onlyOwner {
    require(token != usdc, "Cannot recover primary token");
    require(token.transfer(owner(), amount), "Transfer failed");
}
```

Without such a function, any tokens mistakenly sent to the contract address will be permanently locked.

Severity Note:
- Users may mistakenly transfer non-USDC tokens to the contract address or tokens may be airdropped to it.
- The protocol intentionally only supports USDC for its flows, and no external module exists to recover other tokens.

2 of 11 Findings

contracts/CashbackRewards.sol

Incorrect points calculation in adminRedeemPoints leads to excessive point deduction • Low Risk

The `adminRedeemPoints` function calculates the required points differently than the user-facing `redeemWithPoints` function, resulting in users being charged 1,000,000 times more points when an admin redeems points on their behalf.

In `adminRedeemPoints`:

```
uint256 requiredPoints = itemPrice * POINTS_PER_USDC;
```

Whereas in `redeemWithPoints`:

```
uint256 requiredPoints = itemPrice * POINTS_PER_USDC / UNIT;
```

The missing division by `UNIT` (which equals 1,000,000) means that if an admin redeems points for a user to purchase an item worth 1 USDC, the user will lose 100,000,000 points instead of 100 points. This is a critical issue as it would effectively drain users' point balances if the admin function is ever used, potentially affecting all users in the system.

Severity Note:
- itemPrice is denominated in 6-decimal USDC units (UNIT = 1,000,000)
- POINTS_PER_USDC = 100 means 100 points = 1 USDC
- Owner is a trusted role; onlyOwner is enforced by OZ Ownable

3 of 11 Findings

contracts/CashbackRewards.sol

Precision Loss in Point Calculation Allows Users to Redeem Items for Free

Low Risk

The `redeemWithPoints` function calculates the number of points required for an item using integer division, which leads to significant precision loss. This can result in the required points being calculated as zero for non-zero item prices, allowing users to acquire items for free.

The calculation for `requiredPoints` is:
`uint256 requiredPoints = itemPrice * POINTS_PER_USDC / UNIT;`

Where `POINTS_PER_USDC` is 100 and `UNIT` is 1,000,000. Due to Solidity's integer arithmetic, if `itemPrice * 100` is less than `1,000,000`, the result will truncate to zero. This condition holds true for any `itemPrice` less than 10,000 USDC units (equivalent to 0.01 USDC).

A malicious user can call `redeemWithPoints` with an `itemPrice` between 1 and 9,999. The contract will calculate `requiredPoints` as 0. The check `require(pointsBalance[msg.sender] >= requiredPoints, "not enough points")` will pass even if the user has zero points. The contract then proceeds to transfer the `itemPrice` in USDC to the merchant from its own balance. This vulnerability can be exploited repeatedly to drain the contract's USDC reserves intended for redemptions.

```
// File: contracts/CashbackRewards.sol

function redeemWithPoints(uint256 merchantId, uint256 itemPrice) external {
    uint256 requiredPoints = itemPrice * POINTS_PER_USDC / UNIT; // If itemPrice < 10000,
requiredPoints is 0
    require(pointsBalance[msg.sender] >= requiredPoints, "not enough points");
    Merchant memory m = merchants[merchantId];
    require(m.active, "merchant not active");

    pointsBalance[msg.sender] -= requiredPoints;

    // pool pays merchant in USDC for the redeemed value
    require(usdc.transfer(m.destination, itemPrice), "redeem transfer failed"); // Contract
pays for a free item

    emit Redeem(msg.sender, merchantId, itemPrice, requiredPoints);
}
```

Severity Note:
- Merchants are whitelisted by the owner; typical users cannot set destination addresses.
- There is no gas subsidy; attackers pay gas per call.
- Typical items are priced >= $0.01; sub-cent redemptions are edge cases or require intentional abuse.

4 of 11 Findings

contracts/CashbackRewards.sol

Missing Input Validation in `setMerchant`

Info

The `setMerchant` function, callable only by the owner, lacks critical input validation for the merchant's parameters, which can lead to permanent loss of funds or denial of service.

1. **Zero Destination Address:** The function does not check if `_destination` is `address(0)`. If the owner sets a merchant's destination to the zero address, all funds transferred to that merchant during purchases or redemptions will be permanently lost.
2. **Unbounded Cashback Rate:** The function does not validate the `_cashbackBp` parameter. An owner could set a cashback rate greater than 100% (i.e., `_cashbackBp > 10000`). If this happens, the calculation for `netToMerchant` in the `purchase` function (`amount - fee - cashback`) would underflow and revert, effectively causing a denial of service for all purchases made to that merchant.

```
// File: contracts/CashbackRewards.sol

function setMerchant(uint256 _merchantId, address _destination, uint256 _cashbackBp)
external onlyOwner {
    // No validation for _destination != address(0)
    // No validation for _cashbackBp <= 10000
    merchants[_merchantId] = Merchant({
        merchantId: _merchantId,
        destination: _destination,
        cashbackBp: _cashbackBp,
        active: true
    });
    emit MerchantUpdated(_merchantId, _destination, _cashbackBp);
}
```

Severity Note:
- USDC (and the configured IERC20) reverts on transfer to the zero address (true for OpenZeppelin ERC20 and Circle's USDC).

5 of 11 Findings

contracts/CashbackRewards.sol

No mechanism to deactivate merchants

Info

The contract checks for merchant active status in several functions:

```
require(m.active, "merchant not active");
```

However, there is no function to deactivate a merchant once they've been activated. The `setMerchant` function always sets the `active` flag to `true`:

```
merchants[_merchantId] = Merchant({
    merchantId: _merchantId,
    destination: _destination,
    cashbackBp: _cashbackBp,
    active: true
});
```

This is problematic because it means the protocol cannot quickly disable a merchant in case of suspicious activity, business relationship termination, or if a merchant's account is compromised. The only way to effectively disable a merchant would be to change their parameters to values that make transactions impossible or impractical, which is not a clean solution.

6 of 11 Findings

contracts/CashbackRewards.sol

Admin Can Render Protocol Insolvent by Minting Unbacked Points and Withdrawing Funds

Info

The protocol's solvency depends on its USDC balance being sufficient to cover all outstanding points liabilities. However, two administrative functions, `adminAddCashback` and `withdrawUSDC`, break this fundamental invariant, allowing the owner to make the protocol insolvent.

1. **Unbacked Point Minting**: The `adminAddCashback` function allows the owner to add points to any user's balance arbitrarily. These points represent a liability for the protocol, as they can be redeemed for USDC. However, the function does not require a corresponding deposit of USDC, effectively creating liabilities out of thin air.

2. **Unchecked Fund Withdrawal**: The `withdrawUSDC` function allows the owner to withdraw USDC from the contract. There is no check to ensure that the remaining USDC balance is sufficient to cover the total outstanding points held by users.

An owner can exploit these functions to drain the protocol. For example, the owner could use `adminAddCashback` to grant themselves a large number of points and then use `withdrawUSDC` to remove all the USDC from the contract. This would leave legitimate users with earned points that are worthless because the contract lacks the funds to honor redemptions.

```
// File: contracts/CashbackRewards.sol

// Admin function to add cashback points to a specific user
function adminAddCashback(address user, uint256 pointsToAdd) external onlyOwner {
    require(user != address(0), "invalid user address");
    require(pointsToAdd > 0, "points must be greater than 0");

    pointsBalance[user] += pointsToAdd; // Creates liability without corresponding asset
    emit AdminCashbackAdded(user, pointsToAdd);
}

// Admin function to withdraw USDC balance
function withdrawUSDC(uint256 amount) external onlyOwner {
    require(amount > 0, "amount must be greater than 0");
    require(amount <= 1000000000, "amount must be less than 1000 USDC");
    require(usdc.balanceOf(address(this)) >= amount, "insufficient USDC balance");

    require(usdc.transfer(msg.sender, amount), "withdraw failed"); // Removes assets without
reducing liabilities
    emit AdminWithdraw(msg.sender, amount);
}
```

7 of 11 Findings

contracts/CashbackRewards.sol

Redundant merchant ID storage

Best Practices

The `Merchant` struct includes a `merchantId` field that duplicates the key in the `merchants` mapping. This redundancy wastes storage space and could lead to inconsistencies if the stored ID doesn't match the mapping key.

```
struct Merchant {
    uint256 merchantId; // Redundant as it duplicates the key in merchants mapping
    address destination;
    uint256 cashbackBp;
    bool active;
}

mapping(uint256 => Merchant) public merchants;
```

The `merchantId` field in the struct is unnecessary since the ID is already accessible as the key in the mapping.

8 of 11 Findings

contracts/CashbackRewards.sol

Unused welcome bonus functionality

Best Practices

The contract includes a `hasClaimed` mapping and a `WelcomeBonusClaimed` event, suggesting an intended welcome bonus feature, but there's no function to actually claim this bonus or that uses this mapping.

```
mapping(address => bool) public hasClaimed; // track if user has claimed welcome bonus

event WelcomeBonusClaimed(address indexed user, uint256 points);
```

This appears to be incomplete or abandoned functionality that should either be implemented or removed to avoid confusion.

9 of 11 Findings — contracts/CashbackRewards.sol

Hardcoded USDC decimals assumption

• Best Practices

The contract uses a hardcoded constant `UNIT = 1000000` which assumes USDC has 6 decimals. This might not be true on all chains or if a different token is used in the future.

```
uint256 public constant UNIT = 1000000; // 1 USDC = 1000000 USDC units
```

It would be more flexible to query the token's decimals or allow this value to be configured based on the specific token being used.

10 of 11 Findings — contracts/CashbackRewards.sol

Inconsistent Validation for `merchantId`

• Best Practices

The contract has inconsistent validation rules for `merchantId`. The `adminRedeemPoints` function requires `merchantId > 0`, but the user-facing functions `purchase` and `redeemWithPoints` have no such check. This allows users to interact with a merchant at `merchantId = 0`, but prevents the admin from performing redemptions for that same merchant. This inconsistency can lead to confusion and unexpected behavior. It is best practice to either disallow `merchantId = 0` throughout the contract or allow it consistently.

```
// File: contracts/CashbackRewards.sol

function adminRedeemPoints(address user, uint256 merchantId, uint256 itemPrice) external
onlyOwner {
    // ...
    require(merchantId > 0, "invalid merchant id"); // Check is present
    // ...
}

function purchase(uint256 merchantId, uint256 amount) external {
    Merchant memory m = merchants[merchantId]; // No check for merchantId > 0
    require(m.active, "merchant not active");
    // ...
}
```

11 of 11 Findings

contracts/CashbackRewards.sol

Misleading Code Comment on Points Value

Best Practices

A code comment incorrectly describes the value of points. The comment states `// user -> points (100 points = 1 USDC unit)`. A "USDC unit" in this contract is defined by the `UNIT` constant as 1/1,000,000th of a USDC. However, the actual logic implemented in the contract equates 100 points to 1 full USDC (`1,000,000` USDC units). This discrepancy can cause significant confusion for developers or auditors reviewing the code. The comment should be corrected to reflect the actual implementation, for example: `// 100 points = 1 USDC`.

```
// File: contracts/CashbackRewards.sol

// ...
uint256 public constant UNIT = 1000000; // 1 USDC = 1000000 USDC units

// ...
mapping(address => uint256) public pointsBalance; // user -> points (100 points = 1 USDC
unit)
// ...
```

Disclaimer

Kindly note, the Audit Agent is currently in beta stage and no guarantee is being given as to the accuracy and/or completeness of any of the outputs the Audit Agent may generate, including without limitation this Report. The results set out in this Report may not be complete nor inclusive of all vulnerabilities. The Audit Agent is provided on an 'as is' basis, without warranties or conditions of any kind, either express or implied, including without limitation as to the outputs of the code scan and the security of any smart contract verified using the Audit Agent.

Blockchain technology remains under development and is subject to unknown risks and flaws. This Report does not indicate the endorsement of any particular project or team, nor guarantee its security. Neither you nor any third party should rely on this Report in any way, including for the purpose of making any decisions to buy or sell a product, service or any other asset.

To the fullest extent permitted by law, Nethermind disclaims any liability in connection with this Report, its content, and any related services and products and your use thereof, including, without limitation, the implied warranties of merchantability, fitness for a particular purpose, and non-infringement. Nethermind does not warrant, endorse, guarantee, or assume responsibility for any product or service advertised or offered by a third party through the product, any open source or third-party software, code, libraries, materials, or information linked to, called by, referenced by or accessible through the report, its content, and the related services and products, any hyperlinked websites, any websites or mobile applications appearing on any advertising, and Nethermind will not be a party to or in any way be responsible for monitoring any transaction between you and any third-party providers of products or services. As with the purchase or use of a product or service through any medium or in any environment, you should use your best judgment and exercise caution where appropriate.

FOR AVOIDANCE OF DOUBT, THE REPORT, ITS CONTENT, ACCESS, AND/OR USAGE THEREOF, INCLUDING ANY ASSOCIATED SERVICES OR MATERIALS, SHALL NOT BE CONSIDERED OR RELIED UPON AS ANY FORM OF FINANCIAL, INVESTMENT, TAX, LEGAL, REGULATORY, OR OTHER ADVICE.