Analysis Report By Favour

Summary

Issue Summary

Category	No. of Issues
High	2
Low	9

High Issues

H-1: Uninitialized State Variables

Severity

Impact: High Likelihood: High

Description

The AutheoRewardDistribution contract contains multiple critical state variables that are not properly initialized in the constructor. This could lead to undefined behavior, potential security vulnerabilities, and incorrect reward calculations.

Proof of Concept

Affected Variables:

```
    // Token configuration
    IERC20 public immutable Autheo;  // Line 16

    // Tracking variables
    uint256 public totalDappRewardsIds;  // Line 41

    // Bug bounty user counts
    uint256 public totalLowBugBountyUserNumber;    // Line 51
    uint256 public totalMediumBugBountyUserNumber; // Line 52
    uint256 public totalHighBugBountyUserNumber;   // Line 53

    // Current IDs for different user categories
    uint256 public dappUserCurrentId;              // Line 68
    uint256 public contractDeployerCurrentId;      // Line 69
    uint256 public lowBugBountyCurrentId;          // Line 70
    uint256 public mediumBugBountyCurrentId;       // Line 71
    uint256 public highBugBountyCurrentId;         // Line 72
```

Current Constructor Implementation:

```
    constructor() Ownable(msg.sender) {
        totalSupply = 1050000000000000000000000;
        isTestnet = true; // Start in testnet mode
}
```

Proof of Concept

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.28;

import "./AutheoRewardDistribution.sol";

contract AutheoRewardDistributionExploit {
    AutheoRewardDistribution public rewardContract;

    function setup() public {
        // Deploy the contract without initializing critical variables
        rewardContract = new AutheoRewardDistribution();
    }

    function exploit() public {
        // 1. Demonstrate Autheo token address is zero
        address autheoAddress = address(rewardContract.Autheo());
        require(autheoAddress == address(0), "Autheo token should be uninitialized");

        // 2. Attempt reward calculations with uninitialized user counts
        try rewardContract.claimReward(false, false, true) {
            revert("Should fail due to division by zero");
        } catch Error(string memory reason) {
            // Expected to fail due to division by zero in reward calculations
            require(
                keccak256(bytes(reason)) == keccak256(bytes("Division by zero")),
                "Expected division by zero error"
            );
        }

        // 3. Demonstrate ID tracking issues
        uint256 initialDappUserId = rewardContract.dappUserCurrentId();
        require(initialDappUserId == 0, "DappUserCurrentId should be uninitialized");
    }
}

// Test Contract
contract TestAutheoRewardDistribution {
    AutheoRewardDistribution public rewardContract;

    function testMissingSetters() public {
        // 1. Deploy contract
        rewardContract = new AutheoRewardDistribution();

        // 2. Test Autheo token initialization
        address autheoToken = address(rewardContract.Autheo());
        assert(autheoToken == address(0)); // Will pass as there's no setter

        // 3. Test bug bounty user counts
        uint256 lowBountyUsers = rewardContract.totalLowBugBountyUserNumber();
        uint256 mediumBountyUsers = rewardContract.totalMediumBugBountyUserNumber();
        uint256 highBountyUsers = rewardContract.totalHighBugBountyUserNumber();

        assert(lowBountyUsers == 0);
        assert(mediumBountyUsers == 0);
        assert(highBountyUsers == 0);


    }
}
```

Impact Demonstration:

- Token Address Issue:

```
    // The Autheo token address is never initialized
    IERC20 public immutable Autheo;

    // This could lead to failed token operations
    function emergencyWithdraw(address token) external onlyOwner {
        uint256 balance = IERC20(token).balanceOf(address(this));
        // Will fail if Autheo is used and not initialized
}
```

- Reward Calculation Issues:

```
    function __bugBountyClaim(address _user) private {
        // These calculations use uninitialized variables
        lowRewardPerUser = ((totalBugBountyAllocation * LOW_PERCENTAGE) / 10000) /
            totalLowBugBountyUserNumber; // Could divide by zero

        mediumRewardPerUser = ((totalBugBountyAllocation * MEDIUM_PERCENTAGE) / 10000)
            totalMediumBugBountyUserNumber; // Could divide by zero

        highRewardPerUser = ((totalBugBountyAllocation * HIGH_PERCENTAGE) / 10000) /
            totalHighBugBountyUserNumber; // Could divide by zero
    }
```

- ID Tracking Issues:

```
    function registerDappUsers(address[] memory _dappRewardsUsers, bool[] memory _userU
        // dappUserCurrentId is uninitialized
        dappUserCurrentId++; // Increments from 0, might not be intended behavior
}
```

Proposed Fix:

```
    constructor(address _autheoToken) Ownable(msg.sender) {
    require(_autheoToken != address(0), "Invalid token address");

    // Initialize token
    Autheo = IERC20(_autheoToken);

    // Initialize total supply
    totalSupply = 1050000000000000000000000000;

    // Initialize tracking variables
    totalDappRewardsIds = 0;

    // Initialize bug bounty user counts
    totalLowBugBountyUserNumber = 0;
    totalMediumBugBountyUserNumber = 0;
    totalHighBugBountyUserNumber = 0;

    // Initialize current IDs
    dappUserCurrentId = 0;
    contractDeployerCurrentId = 0;
    lowBugBountyCurrentId = 0;
    mediumBugBountyCurrentId = 0;
    highBugBountyCurrentId = 0;

    // Set initial state
    isTestnet = true;
}
```

Mitigation

1. Add proper setter functions for all critical state variables
2. Implement initialization checks
3. Add events for tracking initialization
4. Add proper validation in setter functions
5. Consider using an initialization pattern with access control

H-2: Unprotected Native ETH Functions

Severity

Impact: High Likelihood: High

Description

The AutheoRewardDistribution contract has two functions that handle native ETH transfers without proper access controls:

1. claimReward() - Allows claiming rewards in native ETH
2. withdraw() - Allows withdrawing contract's ETH balance

1. Affected Functions

```
// Unprotected withdraw function
function withdraw() external {
    payable(msg.sender).transfer(address(this).balance);
}

// Unprotected claim function
function claimReward(
    bool _contractDeploymentClaim,
    bool _dappUserClaim,
    bool _bugBountyClaim
) external nonReentrant whenNotPaused whenTestnetInactive {
    // No proper validation of claim eligibility
}
```

Impact

- Unauthorized Access: Any external address can call these functions
- Fund Drainage: Contract's entire ETH balance can be stolen
- Denial of Service: Legitimate users may be unable to claim rewards
- Financial Loss: Project funds intended for rewards can be compromised

Root Cause Analysis

1. Missing Access Controls:

- No ownership checks
- No role-based access control
- No whitelisting validation
- No balance/allowance checks

2. Architectural Flaws:
    - Direct ETH transfers without validation
    - No rate limiting mechanisms
    - Lack of withdrawal limits
    - Insufficient state validation

Attack Vectors

1. Direct Withdrawal Attack:
    - Attacker calls unprotected withdraw() function
    - Contract balance is transferred to attacker
    - No authorization required
2. Unauthorized Claim Attack:
    - Attacker exploits claimReward() function
    - False parameters bypass checks
    - Drain funds before valid claims

Recommendations

1. Implement Access Controls

```
// Add modifiers
modifier onlyAuthorized() {
    require(isAuthorized[msg.sender], "Unauthorized");
    _;
}

// Add role-based access
modifier onlyClaimant() {
    require(isEligibleClaimer[msg.sender], "Not eligible");
    _;
}
```

2. Add Security Checks

- Implement withdrawal limits

- Add timelock mechanisms
- Include balance validations
- Verify claim eligibility

3. Best Practices

- Use OpenZeppelin's AccessControl
- Implement multi-signature requirements
- Add emergency pause functionality
- Include event logging

4. Additional Safeguards

- Rate limiting
- Withdrawal delays
- Balance thresholds
- Transaction monitoring

L-1: Unsafe ERC20 Operations in AutheoRewardDistribution

The contract uses unsafe ETH transfer operations (transfer()) that could lead to potential vulnerabilities.

- Location: Line 613 - payable(msg.sender).transfer(address(this).balance);
- The contract uses .transfer() which has a fixed gas limit of 2300 gas

Impact

1. Failed Transfers:
    - Transfers may fail if recipient is a contract with expensive fallback
    - Gas costs changes could break functionality
2. Stuck Funds:
    - ETH could become locked in contract if transfer fails
    - No fallback mechanism for failed transfers

Recommendation

1. Replace .transfer() with .call{value: amount}("")
2. Implement proper return value checking

3. Use OpenZeppelin's SafeERC20 library
4. Add reentrancy protection

L-2: Solidity pragma should be specific, not wide

Consider using a specific version of Solidity in your contracts instead of a wide version. For example, instead of `pragma solidity ^0.8.0;` , use `pragma solidity 0.8.0;`

▶ 1 Found Instances

L-3: `public` functions not used internally could be marked `external`

Instead of marking a function as `public` , consider marking it as `external` if it is not used internally.

▶ 5 Found Instances

L-4: Define and use `constant` variables instead of using literals

If the same constant literal value is used multiple times, create a constant state variable and reference it throughout the contract.

▶ 3 Found Instances

L-5: Event is missing `indexed` fields

Index event fields make the field more quickly accessible to off-chain tools that parse events. However, note that each index field costs extra gas during emission, so it's not necessarily best to index the maximum allowed per event (three fields). Each event should use three indexed fields if there are three or more fields, and gas usage is not particularly of concern for the events in question. If there are fewer than three fields, all of the fields should be indexed.

▶ 6 Found Instances

L-6: Modifiers invoked only once can be shoe-horned

into the function

▶ 1 Found Instances

L-7: Large literal values multiples of 10000 can be replaced with scientific notation

Use `e` notation, for example: `1e18` , instead of its full numeric value.

▶ 5 Found Instances

L-8: Loop contains `require` / `revert` statements

Avoid `require` / `revert` statements in a loop because a single bad item can cause the whole transaction to fail. It's better to forgive on fail and return failed elements post processing of the loop

▶ 5 Found Instances

L-9: OpenZeppelin Library Structure Issues

Overview

This PoC documents two library-related issues in the AutheoRewardDistribution contract:

- Usage of deprecated StringsUpgradeable library
- Incorrect import path for ReentrancyGuard

Issue Details

1. StringsUpgradeable Deprecation
   - Current Import: The contract attempts to use StringsUpgradeable
   - Status: StringsUpgradeable is no longer available in OpenZeppelin
   - Correct Usage: Should use `@openzeppelin/contracts/utils/Strings.sol`
2. ReentrancyGuard Location
   - Current Import: `import "@openzeppelin/contracts/security/ReentrancyGuard.sol"`
   - Correct Path: `import "@openzeppelin/contracts/utils/ReentrancyGuard.sol"`
   - Issue: ReentrancyGuard has been moved to utils directory

Impact

- Build failures due to missing imports
- Potential dependency conflicts
- Incorrect library versioning