RBD Labs

Hawk High Audit Report

Prepared by: RbdLabs

Table of Contents

Protocol Summary

The audit uncovered 4 High, 2 Medium, and 2 Low severity issues impacting protocol security, user experience, and financial accuracy. Key findings include Storage collision due to improper upgrade practices, leading to overwritten variables and broken logic graduation and upgrade logic bypassing protocol rules, miscalculation of teacher payments causing overallocation of bursary funds, and absence of a storage gap in the upgradeable contract, creating a risk of future storage collisions.

Disclaimer

The RBDLabs team makes all effort to find as many vulnerabilities in the code in the given time period, but holds no responsibilities for the findings provided in this document. A security audit by the team is not an endorsement of the underlying business or product. The audit was time-boxed and the review of the code was solely on the security aspects of the Solidity implementation of the contracts.

Issues found

Severtity	Number of issues found
High	4
Medium	1
Total	5

Findings

H-1: Contract locks Ether without a withdraw function

It appears that the contract includes a payable function to accept Ether but lacks a corresponding function to withdraw it, which leads to the Ether being locked in the contract. To resolve this issue, please implement a public or external function that allows for the withdrawal of Ether from the contract.

▶ 1 Found Instances

- Found in src/LevelOne.sol Line: 30

  ```
  contract LevelOne is Initializable, UUPSUpgradeable {
  ```

H-2: Storage Collision in Contract Upgrade

Description

In the original contract, the third storage slot holds uint256 schoolFees. In the upgraded contract, that slot is now uint256 public sessionEnd.

Impact

Data previously held in schoolFees will be interpreted as sessionEnd, leading to inconsistent behavior and potential logic failure.

Mitigation

When upgrading contracts, do not change, remove, or reorder variables. Always append new variables at the end of the contract to avoid storage slot collisions.

```
listOfStudents.push(msg.sender);
isStudent[msg.sender] = true;
studentScore[msg.sender] = 100;
bursary += schoolFees;

usdc.safeTransferFrom(msg.sender, address(this), schoolFees); //after state
change
```

H-3: Graduation and Upgrade Logic Violates Core Protocol Rules

Description

The graduateAndUpgrade() function allows the principal to authorize an upgrade and pay wages without validating critical conditions defined by the protocol. Specifically, it does not:

- Check whether the current time has passed sessionEnd

- Ensure all students have received 4 weekly reviews
- Validate that students meet the cutOffScore for graduation

Impact

- Premature upgrade before the session ends
- Wages paid without students completing the program
- Ineligible students potentially considered graduated
- Undermines system integrity and contract rules

Mitigation

Add the following checks before calling _authorizeUpgrade():

- require(block.timestamp >= sessionEnd, "...")
- Loop through listOfStudents and ensure:
  - reviewCount[student] == 4
  - studentScore[student] >= cutOffScore

H-4 incorrect teacher pay calculation

Description: Teachers receive 35% of the bursary each, which when multiplied by multiple teachers results in more than 35% total being distributed, leaving less than 60% for the bursary.

Impact: Significant fund misallocation that could deplete the bursary faster than intended, harming the school's financial stability.

Mitigation: Calculate teacher payments as 35% of bursary divided by number of teachers to ensure total teacher share is 35%.

```diff
- uint256 payPerTeacher = (bursary * TEACHER_WAGE) / PRECISION;
+ uint256 payPerTeacher = (bursary * TEACHER_WAGE) / (PRECISION * totalTeachers);
```

M-1 Missing Storage Gap in UUPSUpgradeable Contract May Cause Storage Collisions

Description

The UUPSUpgradeable contract does not include a reserved storage gap (__gap), which is critical for ensuring upgradeable contract safety. This omission can lead to storage collisions when new variables are added in future implementations, potentially overwriting existing storage and introducing unexpected behavior.

*** Impact*** Without a reserved storage gap, any newly added state variables in upgraded implementations may overlap with storage slots used by the inherited contracts. This can corrupt contract state, break logic, or introduce vulnerabilities that are hard to detect.

Mitigation Add a fixed-size storage gap at the end of the contract, as recommended by OpenZeppelin's UUPSUpgradeable pattern. For example:

uint256[50] private __gap; This allows for future upgrades without risking storage conflicts.