Hawk High First Flight Audit Report

May 1,2025 -> May 8, 2025

Prepared by: cosminmarian53

Table of Contents

Protocol Summary

At the end of the school session (4 weeks), the system is upgraded to a new one.

- A school session lasts 4 weeks
- For the sake of this project, assume USDC has 18 decimals
- Wages are to be paid only when the `graduateAndUpgrade()` function is called by the `principal`
- Payment structure is as follows:
  - `principal` gets 5% of `bursary`
  - `teachers` share of 35% of bursary
  - remaining 60% should reflect in the bursary after upgrade
- Students can only be reviewed once per week
- Students must have gotten all reviews before system upgrade. System upgrade should not occur if any student has not gotten 4 reviews (one for each week)
- Any student who doesn't meet the `cutOffScore` should not be upgraded
- System upgrade cannot take place unless school's `sessionEnd` has reached

Risk Classification

	Impact		
	High	Medium	Low

		Impact		
	High	H	H/M	M
Likelihood	Medium	H/M	M	M/L
	Low	M	M/L	L

We use the CodeHawks severity matrix to determine severity. See the documentation for more details.

Audit Details

Scope

```
├── src
│   ├── LevelOne.sol
│   └── LevelTwo.sol
```

- Chain: EVM Compatible
- Token: USDC

Roles

- `Principal`: In charge of hiring/firing teachers, starting the school session, and upgrading the system at the end of the school session. Will receive 5% of all school fees paid as his wages. can also expel students who break rules.
- `Teachers`: In charge of giving reviews to students at the end of each week. Will share in 35% of all school fees paid as their wages.
- `Student`: Will pay a school fee when enrolling in Hawk High School. Will get a review each week. If they fail to meet the cutoff score at the end of a school session, they will be not graduated to the next level when the `Principal` upgrades the system.

Executive Summary

Participating in this “first flight” has been a pivotal milestone in my security-auditing journey. Fresh from completing Cyfrin Updraft’s Smart Contract Security and Auditing course, I dove straight into this codebase—and was thrilled to uncover a slew of vulnerabilities. Finding so many issues felt both rewarding and exhilarating, and it’s only fueled my excitement to keep auditing.

Issues found

Number of findings by me in the contest:

- High: 2/4
- Medium: 2/2
- Low: 4/11

Findings

High

H-01 Missing UUPS Upgrade Invocation Halts Graduation

Summary

(src/LevelOne.sol#261-278)

```
function graduateAndUpgrade(address _levelTwo, bytes memory)
    public onlyPrincipal
{
    _authorizeUpgrade(_levelTwo);
    // … transfers …
    // ✖ no upgradeTo or upgradeToAndCall called
}
```

Vulnerability Details

The function calls `_authorizeUpgrade(_levelTwo)` to verify permissions but never actually invokes the UUPS upgrade mechanism (`_upgradeToAndCall` or the public `upgradeToAndCall`) to point the proxy at the new implementation. As a result, the proxy's implementation slot remains set to **LevelOne**, so **LevelTwo**'s logic (including its `graduate()` entrypoint) is never executed.

Impact

Enrolled students can never transition to Level Two, and any graduation-specific state or payouts in **LevelTwo** never occur.

Calls to `graduateAndUpgrade` succeed (no revert), giving the false impression that the upgrade—and thus graduation—has happened, while in reality nothing changes.

Tools Used

- Foundry
- Manual Review

Proof Of Concept

- A unit test (`test_storage_collision` demonstrated that after calling `graduateAndUpgrade`, reading `sessionEnd()` still shows the old value (zero), proving no upgrade occurred.

- **OpenZeppelin UUPS Docs**

  - Reference for the correct usage of `upgradeToAndCall` in UUPS-style upgradeable contracts.

Recommendations

1. **Invoke the Upgrade**

Replace the line `_authorizeUpgrade(_levelTwo)` call with a full UUPS upgrade, for example:

```
- _authorizeUpgrade(_levelTwo);
+ _upgradeToAndCall(_levelTwo, data, false);
// — or simply —
+ upgradeToAndCall(_levelTwo, data);
```

This ensures the proxy's implementation pointer is updated before executing any new logic.

1. **Add a Test Guard**

Write a unit test that asserts the proxy's implementation address has changed after `graduateAndUpgrade`, preventing regressions in future versions.

For more information reffer to https://github.com/crytic/slither/wiki/Detector-Documentation#unprotected-upgradeable-contract

H-02 Missing cutOffScore enforcement allows ineligible students to graduate

`LevelOne::graduateAndUpgrade()` upgrades the proxy to `LevelTwo` and finalises the academic session but **never checks each student's** `studentScore` **against** `cutOffScore`. Because `cutOffScore` is only set (in `LevelOne::startSession`) and never read/compared, the invariant *"Any student whose score is below the cut-off must not be upgraded"* is violated.

Impact

- **Logical integrity broken** — students who failed the session still exist in `LevelTwo`, undermining the core business rule of the system.
- **Follow-on bugs** — later LevelTwo logic may assume every student passed the cut-off, leading to incorrect calculations or privilege escalation.

Severity is marked High as one of the fundamental business rules is bypassed.

Proof of Concept

Add to `LevelOneAndGraduateTest.t.sol` and run `forge test --match-test test_student_below_cutoff_still_graduates -vvv` => The test will pass, proving that the student Harriet is still present in LevelTwo despite having a score of 60 (cutoff is 70).

```
function test_student_below_cutoff_still_graduates()
    public
    schoolInSession
{
    levelTwoImplementation = new LevelTwo();
    levelTwoImplementationAddress = address(levelTwoImplementation);

    bytes memory data = abi.encodeCall(LevelTwo.graduate, ());

    // Lower Harriet’s score from 100 → 60 (< cutOffScore)
    vm.startPrank(alice);
    for (uint256 i; i < 4; i++) {
        vm.warp(block.timestamp + 1 weeks);
        levelOneProxy.giveReview(harriet, false);
    }
    vm.stopPrank();

    assertLt(
        levelOneProxy.studentScore(harriet),
        levelOneProxy.cutOffScore()
    );

    vm.prank(principal);
    levelOneProxy.graduateAndUpgrade(levelTwoImplementationAddress, data);

    LevelTwo levelTwoProxy = LevelTwo(proxyAddress);

    // Harriet should not be a student anymore; assert should fail but
passes
    assertTrue(levelTwoProxy.isStudent(harriet));
}
```

Recommended Mitigation

Multiple mitigation options exist: either revert or automatically expel students who are under the cutoff at the time of the upgrade/graduation.

1. **Revert if any student is below the cut-off** Add a pre-check before the upgrade:

   ```
   function _checkCutOff() internal view {
       for (uint256 i; i < listOfStudents.length; ++i) {
           require(studentScore[listOfStudents[i]] >= cutOffScore,
                   "Some students below cut-off");
       }
   }
   // call _checkCutOff() inside graduateAndUpgrade()
   ```

2. **Automatic expulsion option** Iterate through `listOfStudents` and `levelOne::expel()` every address with a score below `cutOffScore` before authorising the upgrade. This keeps the

single-transaction UX while upholding the invariant:

```
function _expelFailures() internal {
    uint256 i = 0;
    while (i < listOfStudents.length) {
        address s = listOfStudents[i];
        if (studentScore[s] < cutOffScore) {
            expel(s);
        } else {
            ++i;
        }
    }
}
// call _expelFailures() inside graduateAndUpgrade()
```

Either approach ensures no student with a score lower than the cutoff can pass into Level Two.

Medium

M-01 Unprotected Initializer Allows Full Takeover

Summary

The issue happens here:

(src/LevelOne.sol#125–141)

```
functioninitialize(address _principal, uint256 _schoolFees, address
_usdcAddress)
public initializer
{ … }
```

Vulnerability Details

Lack of `onlyProxy` or constructor lock: No `onlyProxy` guard and no `_disableInitializers()` in a constructor, so **anyone** can call `initialize()` on the implementation or proxy first.

- **Attacker as principal**: Once called, the attacker becomes `principal` and satisfies `onlyPrincipal`, enabling them to call `upgradeToAndCall` on the proxy to any malicious implementation.

Impact

Anyone can call initialize on the logic contract, and destruct the contract.

Proof Of Code

Add the following test to `LevelOneAndGraduateTest.t.sol`:

```
function test_anyone_can_initialize_implementation() public {
        // 1) Grab the raw implementation address (uninitialized)
        LevelOne impl = LevelOne(levelOneImplementationAddress);

        // 2) Prepare a “malicious” principal and new params
        address attacker = makeAddr("attacker");
        uint256 fakeFees = 1;

        // 3) Attacker calls initialize(...) on the implementation
        vm.prank(attacker);
        impl.initialize(attacker, fakeFees, address(usdc));

        // 4) Assert that the implementation’s storage was overwritten
        assertEq(impl.getPrincipal(), attacker);
        assertEq(
            impl.getSchoolFeesCost(),
            fakeFees
        );
        assertEq(
            impl.getSchoolFeesToken(),
            address(usdc)
        );
    }
```

Tools Used

- Slither
- Foundry
- Manual Review

Recommendations

1. Add a constructor to ensure `initialize` cannot be called on the logic contract

2. Add `onlyProxy` modifier to `initialize` so it can only be called via the proxy

```
...
+//@custom:oz-upgrades-unsafe-allow constructor
+  constructor() {
+    _disableInitializers();
+  }
....
- function initialize(address _principal, uint256 _schoolFees, address
_usdcAddress) public initializer {..}
+ function initialize(address _principal, uint256 _schoolFees, address
_usdcAddress) public onlyProxy initializer {
```

M-02 Storage-Layout Collision Risk on Future Upgrade

Summary

The LevelOne → LevelTwo UUPS upgradeable contracts suffer from a **storage-layout collision**: after slot 0, every variable in LevelTwo shifts and reinterprets the proxy's existing storage, corrupting state. This latent bug lies dormant until the proxy is correctly upgraded, at which point `schoolFees` (slot 1) becomes `sessionEnd`, `sessionEnd` (slot 2) becomes `bursary`, and so on, silently overwriting balances, scores, and mappings. If exploited or simply triggered by a future upgrade, this will break business logic and could cause substantial token loss. Mitigation requires aligning storage (inheritance or gaps) and adding automated CI checks.

Vulnerability Details

Storage Layout Comparison

Slot	LevelOne	LevelTwo
0	`address principalbool bool inSession`	`address principalbool bool inSession`
1	`uint256 schoolFees`	`uint256 sessionEnd`
2	`uint256 sessionEnd`	`uint256 bursary`
3	`uint256 bursary`	`uint256 cutOffScore`
4	`uint256 cutOffScore`	`mapping(address=>bool) isTeacher`
…	…	…

Because LevelTwo neither inherits LevelOne nor reserves unused slots, **every** subsequent variable is repurposed on upgrade, leading to **storage collision**

Impact

Critical fields like `bursary` (accumulated USDC) may map to a student's cut-off score or teacher wage slots, enabling arbitrary transfers or lost funds.

Student scores, session flags, and teacher/ student rosters vanish or invert, halting core school operations.

Tools Used

- Foundry
- Manual Review

Proof of Concept

How It Happens

1. **Delegatecall Semantics**: A proxy's `delegatecall` means the implementation reads/writes the proxy's storage.
2. **Mismatched Definitions**: Changing order, removing, or adding fields without reserving gaps shifts all downstream slots.
3. **Silent Corruption**: No compiler or runtime error—existing state is simply misinterpreted under the new layout.

   If the upgrade to LevelTwo issue would be fixed, then the following test holds true:

```
 function test_storage_collision() public {
        // 1. Read slot 1 (schoolFees) before upgrade
        bytes32 rawBefore = vm.load(proxyAddress, bytes32(uint256(1)));
        uint256 feesBefore = uint256(rawBefore);
        assertEq(feesBefore, schoolFees);

        // 2. Perform the UUPS upgrade to LevelTwo
        levelTwoImplementation = new LevelTwo();
        bytes memory data = abi.encodeCall(LevelTwo.graduate, ());
        vm.prank(principal);
        levelOneProxy.graduateAndUpgrade(address(levelTwoImplementation),
data);

        LevelTwo levelTwoProxy = LevelTwo(proxyAddress);

        // 3. Read slot 1 again (now interpreted as sessionEnd)
        bytes32 rawAfter = vm.load(address(levelTwoProxy),
bytes32(uint256(1)));
        uint256 sessionEndSlot = uint256(rawAfter);
        // It must still equal the same schoolFees value => collision
        assertEq(sessionEndSlot, schoolFees);
    }
```

Recommendations

1. Either have `LevelTwo` inherit `LevelOne` so state variables preserve exact slot positions, like this:

```
contract LevelTwo is LevelOne { … }
```

1. Or use blank storage variables in order to align storage state.

Low

L-01 Missing Session-End Check Allows Premature Graduation and Upgrade

Summary

The `graduateAndUpgrade` function in the `LevelOne` contract lacks any enforcement that the school session has ended before allowing an upgrade. As a result, the contract can be upgraded—and graduation logic executed—immediately after deployment, bypassing the intended time-based lockout controlled by `sessionEnd`. This single flaw breaks the invariant that graduation may only occur once the session has concluded.

Vulnerability Details

In the UUPS proxy pattern, the implementation contract bears responsibility for both business and upgrade logic, delegating calls from the proxy to itself for `upgradeTo` and `_authorizeUpgrade` handling Documentation - OpenZeppelin Docs. While the contract correctly restricts `_authorizeUpgrade` to the `principal`, it entirely omits any check of `block.timestamp` against `sessionEnd` within either `graduateAndUpgrade` or `_authorizeUpgrade`. Consequently, a call to

```
function graduateAndUpgrade(address _levelTwo, bytes memory) public
onlyPrincipal { … }
```

can be made at any time, irrespective of whether the period defined by

```
        sessionEnd = block.timestamp + 4 weeks;
```

has elapsed. This defeats the very purpose of `sessionEnd`, rendering the time-based control over graduation a no-op and allowing premature or repeated upgrades.

Impact

A malicious or compromised `principal` can trigger graduation immediately, without waiting the intended 4-week term.

Recommendations

In order to fix this issue, consider using a modifier that checks whether the session has ended or not:

```
modifier sessionEnded() {
    require(block.timestamp >= sessionEnd, "Session not ended");
    _;
}
```

L-02 Missing Review-Count Increment Breaks Weekly-Review Invariant

Summary

The `giveReview` function in the `LevelOne` contract enforces a one-week cooldown between reviews and purports to limit each student to five total reviews, but it never increments the `reviewCount` mapping. As a result, a teacher can call `giveReview` indefinitely on the same student—one time per week—completely bypassing the intended four-review cap.

Vulnerability Details

The code checks two conditions before recording a review: that the caller is a teacher and that fewer than five reviews have been given (`require(reviewCount[_student] < 5, "Student review count exceeded!!!")`), and that at least one week has passed since the last review (`require(block.timestamp >= lastReviewTime[_student] + reviewTime, "Reviews can only be given once per week")`). However, after passing these checks and adjusting the student's score and `lastReviewTime`, the function never increments `reviewCount[_student]`. Consequently, the five-review limit is never reached and the count check remains perpetually true.

Impact

A malicious or compromised teacher can indefinitely penalize or reward a student beyond the intended limit, potentially manipulating student scores and graduation eligibility. The effect undermines fairness and trust in the system's academic logic but does not directly steal funds. Exploitation requires only teacher privileges and weekly calls, making it straightforward once a teacher account is available.

Proof Of Concept

The following test proves that we can review a student more than 4 times without reverting:

```
function test_reviewCountInvariant_broken() public {
    _teachersAdded();
    _studentsEnrolled();
    vm.prank(principal);
    levelOneProxy.startSession(50);

    for (uint i = 0; i < 10; i++) {
        vm.warp(block.timestamp + levelOneProxy.reviewTime());
        vm.prank(alice);
        levelOneProxy.giveReview(harriet, false);
    }
    assertTrue(true, "reviewCount never increments; more than 5 reviews
allowed");
}
```

Tools Used

- Foundry
- Manual Review

Recommendations

To restore both the weekly-rate and total-count invariants, update `giveReview` as follows:

1. Introduce a separate counter for weekly reviews, resetting it when a new week begins.

```
+ mapping(address => uint256) private weeklyReviewCount;
```

1. Increment the total-review counter and enforce the five-review cap.

```
 function giveReview(address _student, bool review) public onlyTeacher {
     if (!isStudent[_student]) revert HH__StudentDoesNotExist();

     // Reset weekly counter if a full review interval has passed
-    require(block.timestamp >= lastReviewTime[_student] + reviewTime,
"Reviews can only be given once per week");
+    if (block.timestamp >= lastReviewTime[_student] + reviewTime) {
+        weeklyReviewCount[_student] = 0;
+    }
+    require(weeklyReviewCount[_student] < 1, "Only one review per week");

     require(reviewCount[_student] < 5, "Student review count
exceeded!!!");

     if (!review) {
         studentScore[_student] -= 10;
     }

     // Update timestamps and counters
     lastReviewTime[_student] = block.timestamp;
+    weeklyReviewCount[_student] += 1;
+    reviewCount[_student]      += 1;

     emit ReviewGiven(_student, review, studentScore[_student]);
 }
```

L-03 Session Flag Never Resets After Four Weeks

Summary

The `LevelOne` contract sets `sessionEnd = block.timestamp + 4 weeks` and flips `inSession` to `true` when `startSession` is invoked, but never resets `inSession` back to `false` once the four-week term expires. As a result, any on-chain workflow or external caller querying `getSessionStatus()` will always see the session as active, even long after the intended cutoff. This flaw breaks the invariant that "a school session lasts four weeks," leading to misleading state and potential downstream logic errors.

Vulnerability Details

When `startSession(uint256 _cutOffScore)` is called, the contract executes:

```
sessionEnd = block.timestamp + 4 weeks;
inSession  = true;
```

However, there is no code path—modifier, view helper, or external function—that ever flips `inSession` back to `false` when `block.timestamp` exceeds `sessionEnd`. Consequently, calls to:

```
function getSessionStatus() external view returns (bool) {
    return inSession;
}
```

will perpetually return `true`, regardless of the actual time elapsed. Any business logic relying on session closure—such as forbidding new enrollments or gating graduation—will operate under the false assumption that the session is still ongoing

Impact

The `inSession` flag remains `true` indefinitely once `startSession` is called, even after `sessionEnd` has passed, causing the reported session status to be misleading and invalidating any logic that depends on the session having actually ended.

Tools Used

- Foundry
- Manual Review

Proof of Concept

The following Forge test appended to your existing suite demonstrates that even after warping the blockchain time beyond five weeks, `getSessionStatus()` remains `true`:

```
function test_sessionNeverEndsAfterFourWeeks() public {
    _teachersAdded();
    _studentsEnrolled();

    // Start 4-week session now
    vm.prank(principal);
    levelOneProxy.startSession(80);

    // Immediately confirm session is live
    assertEq(levelOneProxy.getSessionStatus(), true);

    // Warp forward by 5 weeks (beyond sessionEnd)
    vm.warp(block.timestamp + 5 weeks);

    // Confirm session is STILL active even after the 4-week period
```

```
        assertEq(
            levelOneProxy.getSessionStatus(),true);
    }
```

Recommendations

Consider adding a state update in a modifier or dedicated function that resets `inSession` to `false` once `block.timestamp >= sessionEnd`. This approach ensures that after four weeks, the session truly ends, restoring the correct contract invariant and preventing downstream logic errors

L-04 Unbounded-Loop Gas-Exhaustion Leads To DoS in Administrative Functions

Summary

The contract implements linear searches over unbounded dynamic arrays in its `removeTeacher`, `expel`, and `graduateAndUpgrade` functions, causing gas consumption to grow proportionally with the number of entries and ultimately exceeding Ethereum's per-block gas limit, resulting in irreversible reverts and denial of service. This behavior is catalogued as "Denial of Service with Block Gas Limit" and falls under DoS through unbounded operations. Because these loops can disrupt core administrative and upgrade flows without direct theft of funds, this would be categorized as a Medium finding.

Vulnerability Details

Every invocation of `removeTeacher` iterates through `listOfTeachers` to locate and remove a teacher address, executing an O(N) search and swap on storage arrays. The `expel` function mirrors this pattern on the student list and the `graduateAndUpgrade` function similarly loops over teachers to distribute USDC payments in a push fashion, compounding gas costs on large cohorts. As the number of teachers or expelled students grows, these unbounded loops incrementally consume more gas until the operation surpasses the block gas limit and reverts.

Impact

Once array sizes cross the threshold where loop gas exceeds the block limit, the principal loses the ability to remove teachers, expel students, or perform payments towards himself and the teachers, effectively stranding USDC in the contract and halting protocol evolution.

Tools Used

- Foundry
- Manual Review
- Aderyn

Proof Of Code

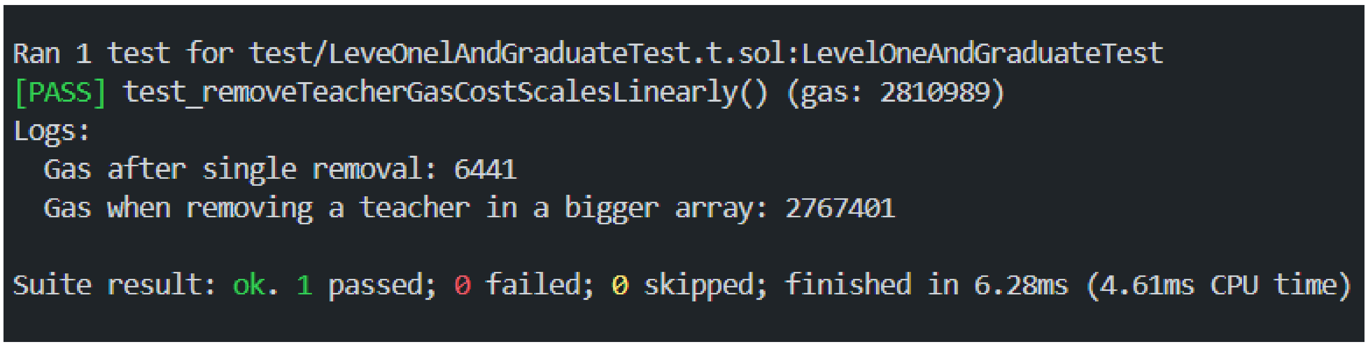
Place the following test into `LevelOneAndGraduateTest.t.sol`:

```
function test_removeTeacherGasCostScalesLinearly() public {
        // Add a single teacher so the array has length 1
        vm.txGasPrice(1);
        vm.prank(principal);
        levelOneProxy.addTeacher(alice);

        // 1) Attempt to remove a teacher in a 1-element array
        uint256 gasStart = gasleft();
        vm.prank(principal);
        levelOneProxy.removeTeacher(alice);
        uint256 gasAfterSingle = gasleft();
        uint256 gasUsedSingle = (gasStart - gasAfterSingle) * tx.gasprice;
        console.log("Gas after single removal:", gasUsedSingle);
        // 2) Bulk-add 50 more teachers to balloon the array
        uint256 gasStartFifty = gasleft();
        for (uint i = 0; i < 50; i++) {
            address t = makeAddr(string.concat("teacher", vm.toString(i)));
            vm.prank(principal);
            levelOneProxy.addTeacher(t);
        }
        // 3) Attempt the same removal in a 51-element array
        vm.prank(principal);
        vm.expectRevert(LevelOne.HH__TeacherDoesNotExist.selector);
        levelOneProxy.removeTeacher(address(1));

        uint256 gasAfterFifty = gasleft();
        uint256 gasUsedFifty = (gasStartFifty - gasAfterFifty) *
tx.gasprice;
        console.log("Gas when removing a teacher in a bigger array:",
gasUsedFifty);
        assert(gasUsedFifty > gasUsedSingle);
    }
```

Running this test will give us the following output:



```
Ran 1 test for test/LeveOnelAndGraduateTest.t.sol:LevelOneAndGraduateTest
[PASS] test_removeTeacherGasCostScalesLinearly() (gas: 2810989)
Logs:
  Gas after single removal: 6441
  Gas when removing a teacher in a bigger array: 2767401

Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 6.28ms (4.61ms CPU time)
```

We can see that gas consumption is significantly higher when removing a teacher because the function must loop through the entire array of fifty-plus addresses. The same issue arises in the student context: expelling a non-existent or end-of-array entry from a fifty-element student list would likewise incur a very costly operation.

Recommendations

Consider using a `mapping` for both teachers and students instead of an ever-growing `address[]` plus boolean flags, so that adding or removing an entry is just a constant-time write or delete and lookups never scan the array.

Altenatively, you can use OpenZeppelin's `EnumerableSet`—its add/remove/contains operations remain O(1) and you only pay the cost of turning its small snapshot into an array when necessary.