Protocol Summary

This project involves the Hawk High School, an educational platform where students enroll, undergo reviews from teachers, and ultimately graduate if they meet specific criteria. The protocol is upgradeable using the UUPSUpgradeable library from OpenZeppelin. At the end of each school session (lasting 4 weeks), the system is upgraded, and wages are paid to teachers and the principal based on the school fees paid.

Key Functions

- Principal Actions:

Enroll students and manage school operations.

Hire and fire teachers.

Call the graduateAndUpgrade() function to upgrade the system and distribute wages.

Receives 5% of the bursary as their salary.

- Teachers Actions:

Review students weekly.

Receive 35% of the bursary as their wages.

- Student Actions:

Pay a school fee upon enrollment.

Receive reviews each week.

Must meet the cutOffScore to graduate at the end of the school session.

If a student does not meet the cutoff, they are not upgraded.

Invariants

A school session lasts 4 weeks.

Wages are paid after calling graduateAndUpgrade() by the principal.

Principal’s wage: 5% of the total bursary.

Teachers’ wage: 35% of the total bursary.

The remaining 60% reflects the bursary after the upgrade.

Students must receive all 4 weekly reviews before the system upgrade can occur.

The system upgrade cannot happen if any student has not been reviewed in all 4 weeks.

Disclaimer

Risk Classification

Impact High Medium Low 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

- Roles

Principal: Deployer of the system, responsible for managing the school session, enrolling students, and upgrading the system. Receives 5% of the total bursary as salary.

Teachers: Review students each week and are entitled to 35% of the total bursary as their wages.

Student: Pays the school fee upon enrollment and must meet the cutOffScore to graduate.

Issues found

Severity	Number of issues found
High	2
Medium	1
Low	1
Total	4

Findings

Highs

[H-1] Absence of Graduation Logic in `LevelOne::graduateAndUpgrade` Function Causing Broken System Upgrade

Summary

The `LevelOne::graduateAndUpgrade` function lacks logic to handle student graduation, which allows the protocol to be upgraded without controlling which students should graduate.

Vulnerability Details

In the `LevelOne::graduateAndUpgrade` function, there is no code to manage the graduation of students, which leads to an upgrade of the contract without controlling who should graduate. A student that does not have four reviews or that does not pass the cutOffScore mark is not marked as not graduated, which goes against the protocol intentention. Also, the system does not check that the session has ended to update the system.

This breaks the protocol's invariants and omits a crucial function from the system.

Impact

This issue severely breaks the protocol's intended behavior by bypassing a critical function of the system. This will occur every time the system is upgraded.

As will be shown in the Proof of Code, `principal` actor is able to call `LevelOne::graduateAndUpdate` function in an undesirable scenario in which the student `fin` does not outperform `LevelOne::cutOffScore`, does not have four reviews in `LevelOne::reviewCount` and in which the session has not ended yet.

Proof of Concept

This test demonstrates a vulnerability that allows the contract upgrade process to proceed even when the necessary conditions for student evaluation are not met. The goal is to bypass validation checks during the session period and confirm that an upgrade can occur undesirably. The steps are as follows:

1. Increase the `cutOffScore` to 100 within the `schoolInSession` modifier in the `LevelOneGraduateTest.t.sol` contract. This raises the minimum score required for a student to qualify for graduation.
2. Set the session duration to 1 week, ensuring that the session is still active and has not yet ended (`sessionEnd` has not occurred).
3. Give the student `fin` only 1 negative review, resulting in:
   - Fewer than 4 total reviews (fails the `reviewCount` invariant)
   - A `studentScore` lower than the `cutOffScore`
4. Trigger the contract upgrade as the `principal`, and verify that the upgrade proceeds despite the unmet conditions, thus confirming the vulnerability.

Proof of Code

To confirm the vulnerability, apply the following changes in the `LevelOneGraduateTest.t.sol` contract:

1. Update the `schoolInSession` modifier by setting the `cutOffScore` to 100:

```diff
    modifier schoolInSession() {
           _teachersAdded();
           _studentsEnrolled();

           vm.prank(principal);
-          levelOneProxy.startSession(70);
+          levelOneProxy.startSession(100);

           _;
```

```
        }
```

2. Add the following test function to simulate the undesired upgrade:

```
    function test_confirm_can_graduate_without_following_invariants() public
schoolInSession {
            uint256 currentSessionTime = block.timestamp + 1 weeks;
            vm.warp(currentSessionTime);
            vm.prank(bob);
            levelOneProxy.giveReview(fin, false);

            console2.log("studentScore: ", levelOneProxy.studentScore(fin));
            // levelOneProxy.reviewCount(fin) will return 0 as giveReview does not
update reviewCount after being called
            console2.log("reviewCount: ", levelOneProxy.reviewCount(fin));
            console2.log("cutOffScore: ", levelOneProxy.cutOffScore());
            console2.log("sessionEnd: ", levelOneProxy.sessionEnd());
            console2.log("Actual Time: ", currentSessionTime);
            assertGt(levelOneProxy.cutOffScore(), levelOneProxy.studentScore(fin));
            assertGt(levelOneProxy.sessionEnd(), currentSessionTime);

            levelTwoImplementation = new LevelTwo();
            levelTwoImplementationAddress = address(levelTwoImplementation);

            bytes memory data = abi.encodeCall(LevelTwo.graduate, ());
            vm.startPrank(principal);
            levelOneProxy.graduateAndUpgrade(levelTwoImplementationAddress, data);
            vm.stopPrank();

            LevelTwo levelTwoProxy = LevelTwo(proxyAddress);
    }
```

Tools Used

The issue was discovered through a manual code review.

Recommendations

It is recommended to split the graduation and upgrade functionalities into two distinct functions for better clarity and security of the code.

1. In the `LevelOne.sol` contract, add an array to store all graduated students and a mapping to register whether a student has graduated:

   ```diff address principal; bool inSession; uint256 schoolFees; uint256 public immutable reviewTime = 1 weeks; uint256 public sessionEnd; uint256 public bursary; uint256 public cutOffScore; mapping(address => bool) public isTeacher; mapping(address => bool) public isStudent; mapping(address => uint256)
   ```

public studentScore; mapping(address => uint256) public reviewCount; mapping(address => uint256) public lastReviewTime; + mapping(address => bool) public isGraduated; // False by default. address[] listOfStudents; address[] listOfTeachers; + address[] graduatedStudents; ```

2. Modify the `LevelOne::Graduated` event and add an `UpgradeToLevelTwo` event:

```
     event TeacherAdded(address indexed);
     event TeacherRemoved(address indexed);
     event Enrolled(address indexed);
     event Expelled(address indexed);
     event SchoolInSession(uint256 indexed startTime, uint256 indexed
endTime);
     event ReviewGiven(address indexed student, bool indexed review, uint256
indexed studentScore);
-    event Graduated(address indexed levelTwo);
+    event Graduated(address indexed);
+    event UpgradeToLevelTwo(address indexed levelTwo, bytes data);
```

3. Ensure that `LevelOne::reviewCount[_student]` is updated each time a review is submitted within the `LevelOne::giveReview` function.
   Note: This is an issue which is discussed in detail in a separate submission.

```
function giveReview(address _student, bool review) public onlyTeacher {
        if (!isStudent[_student]) {
            revert HH__StudentDoesNotExist();
        }
        require(reviewCount[_student] < 5, "Student review count exceeded!!!");
        require(block.timestamp >= lastReviewTime[_student] + reviewTime, "Reviews
can only be given once per week");

        // where `false` is a bad review and true is a good review
        if (!review) {
            studentScore[_student] -= 10;
        }

        // Update last review time
        lastReviewTime[_student] = block.timestamp;
+       // Update review Count
+       reviewCount[_student]++;

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

4.Add a `LevelOne::graduateStudent` function in the `LevelOne.sol` contract that adheres to the protocol's invariants:

```
    function graduateStudent(address _student) public onlyPrincipal {
        require(block.timestamp >= sessionEnd, "Session has not ended yet");
        require(reviewCount[_student] == 4, "Student must have exactly 4 reviews");
        require(studentScore[_student] >= cutOffScore, "Score below cut-off");
        require(!isGraduated[_student], "Already graduated");

        isGraduated[_student] = true;
        graduatedStudents.push(_student);

        emit Graduated(_student);
  }
```

5.Update the `LevelOne::graduateAndUpgrade` function:

It is recommended to modify the **`LevelOne::graduateAndUpgrade`** function to call the new function **`graduateStudent`** before proceeding with the contract upgrade. This ensures that only students who have completed all the necessary requirements are graduated before the contract upgrade takes place.

By calling **`graduateStudent`**, you can ensure that all necessary graduation conditions are met before allowing any changes or upgrades to the contract, preserving the integrity of the system.

After this, the invariants will be checked, and the **`LevelOne::graduateAndUpgrade`** function should be updated as follows:

```diff
-  function graduateAndUpgrade(address _levelTwo, bytes memory ) public onlyPrincipal {
+  function Upgrade(address _levelTwo, bytes memory ) public onlyPrincipal {
        if (_levelTwo == address(0)) {
            revert HH__ZeroAddress();
        }

        uint256 totalTeachers = listOfTeachers.length;

        uint256 payPerTeacher = (bursary * TEACHER_WAGE) / PRECISION;
        uint256 principalPay = (bursary * PRINCIPAL_WAGE) / PRECISION;

+       event UpgradeToLevelTwo(_levelTwo, "");
        _authorizeUpgrade(_levelTwo);

        for (uint256 n = 0; n < totalTeachers; n++) {
            usdc.safeTransfer(listOfTeachers[n], payPerTeacher);
        }

        usdc.safeTransfer(principal, principalPay);
    }
```

Optionally, you can add the new variables to the `LevelTwo.sol` contract if they will be used in the updated implementation.

If you do, consider maintaining the same variable declaration order as in the `LevelOne.sol` contract, and ensure that variables from the `LevelOne.sol` contract are included to avoid storage collisions during the upgrade.

Note: You can check the storage slots by running the following commands in your console:

- `forge inspect LevelOne storage` to inspect the storage of `LevelOne.sol`.
- `forge inspect LevelTwo storage` to inspect the storage of `LevelTwo.sol`.

```diff
    address principal;
    bool inSession;
+   uint256 schoolFees;
+   uint256 private reviewTime = 1 weeks;
    uint256 public sessionEnd;
    uint256 public bursary;
    uint256 public cutOffScore;
    mapping(address => bool) public isTeacher;
    mapping(address => bool) public isStudent;
    mapping(address => uint256) public studentScore;
+   mapping(address => uint256) public reviewCount;
+   mapping(address => uint256) public lastReviewTime;
+   mapping(address => bool) public isGraduated; // False by default.
    address[] listOfStudents;
    address[] listOfTeachers;
+   address[] graduatedStudents;
```

[H-2] Miscalculation of `payPerTeacher` that cause a Denial of Sevice if more than two teachers are added

Summary

The `LevelOne::graduateAndUpgrade` function contains a critical miscalculation in the `payPerTeacher` formula. If more than two teachers are added, the total funds required to pay them exceed the available `bursary`, causing the transaction to revert. This effectively locks the `LevelOne::graduateAndUpgrade` function, resulting in a denial of service.

Vulnerability Details

Inside the `LevelOne::graduateAndUpgrade` function, the contract loops through the `listOfTeachers` array and attempts to distribute teacher salaries using a `payPerTeacher` amount. This value is calculated as `payPerTeacher = (bursary * TEACHER_WAGE) / PRECISION`, where `TEACHER_WAGE = 35` and `PRECISION = 100`.

However, this calculation assumes only one teacher will be paid the entire teacher allocation of the `bursary`. When multiple teachers are present, this formula is applied to each teacher individually, leading to total payouts that exceed the available `bursary`. As a result, if more than two teachers are added, the third or subsequent transfer fails due to insufficient funds, causing the transaction to revert.

Impact

This miscalculation leads to a Denial of Service. When the condition is triggered (more than two teachers added), the `LevelOne::graduateAndUpgrade` function becomes permanently unusable. This prevents the contract from distributing `schoolFees` and upgrading to the next level, effectively locking the system.

Proof of Concept

To demonstrate the vulnerability:

1. Add a third teacher to the contract.
2. Call the `graduateAndUpgrade` function.
3. Observe the transaction revert due to an out-of-funds error when paying the third teacher.
4.

Proof of Code

```diff
    address alice;
    address bob;
+   address keating;
```

```diff
   alice = makeAddr("first_teacher");
   bob = makeAddr("second_teacher");
+  keating = makeAddr("third_teacher"); // DoS teacher
```

```diff
     function test_confirm_add_teacher() public {
        vm.startPrank(principal);
        levelOneProxy.addTeacher(alice);
        levelOneProxy.addTeacher(bob);
+       levelOneProxy.addTeacher(keating);
        vm.stopPrank();

        assert(levelOneProxy.isTeacher(alice) == true);
        assert(levelOneProxy.isTeacher(bob) == true);
-       assert(levelOneProxy.getTotalTeachers() == 2);
+       assert(levelOneProxy.getTotalTeachers() == 3);
    }
```

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

        bytes memory data = abi.encodeCall(LevelTwo.graduate, ());
        vm.prank(principal);
        levelOneProxy.graduateAndUpgrade(levelTwoImplementationAddress, data);

        LevelTwo levelTwoProxy = LevelTwo(proxyAddress);
    }
```

Tools Used

This issue was found by manual review

Recommendations

Combining two critical operations (contract upgrade and funds distribution) in a single function increases the risk of cascading failures. In this case, a miscalculation in teacher payments can revert the entire transaction, permanently blocking the contract upgrade.

1. Consider splitting the logic of the upgrade and the bursary distribution in different functions. `LevelOne::graduateAndUpgrade` will keep only the upgrade logic:

```diff
function graduateAndUpgrade(address _levelTwo, bytes memory) public onlyPrincipal
{
        if (_levelTwo == address(0)) {
            revert HH__ZeroAddress();
        }

-        uint256 totalTeachers = listOfTeachers.length;

-        uint256 payPerTeacher = (bursary * TEACHER_WAGE) / PRECISION;
-        uint256 principalPay = (bursary * PRINCIPAL_WAGE) / PRECISION;



        _authorizeUpgrade(_levelTwo);


-        for (uint256 n = 0; n < totalTeachers; n++) {
-            usdc.safeTransfer(listOfTeachers[n], payPerTeacher);
-        }

-        usdc.safeTransfer(principal, principalPay);
    }
```

2. To ensure fair and efficient distribution of the `bursary`, a proper formula will be introduced in a new function `bursaryDistribution`. Additionally, the current approach of paying teachers via a loop presents scalability challenges as the number of teachers grows, the transaction may become prohibitively expensive or exceed the gas limit, leading to another Denial of Service.

To address this, the following optimizations will be implemented:

- Storing Payments in a Mapping: Instead of iterating over the `listOfTeachers` array to distribute funds in a single transaction, a mapping will store each teacher's allocated payment.
- Self-Service Claim Mechanism: Teachers will be able to independently withdraw their salaries using a new `claimTeacherPay` function. This eliminates the need for costly batch transactions and reduces gas consumption.

This approach improves scalability and ensures the contract remains efficient even as the number of teachers increases.

```
function bursaryDistribution() public onlyPrincipal {
        uint256 teacherTotalPay = (bursary * TEACHER_WAGE) / PRECISION;
        uint256 payPerTeacher = teacherTotalPay / listOfTeachers.length;

        for (uint256 i = 0; i < listOfTeachers.length; i++) {
            pendingTeacherPay[listOfTeachers[i]] = payPerTeacher;
        }

        uint256 principalPay = (bursary * PRINCIPAL_WAGE) / PRECISION;
        usdc.safeTransfer(principal, principalPay);
    }

    function claimTeacherPay() public onlyTeacher {
        uint256 amount = pendingTeacherPay[msg.sender];
        require(amount > 0, "No payment due");
        pendingTeacherPay[msg.sender] = 0;
        usdc.safeTransfer(msg.sender, amount);
    }
```

2. If the desire is to preserve the upgrade and payment logic within the `LevelOne::graduateAndUpgrade` function, the following changes will be implemented:

   2.1. Introduce a `teachersTotalPay` variable, which will replicate the logic from `LevelOne::payPerTeacher` to determine the total disbursable amount for teachers.

   2.2. Compute the individual `payPerTeacher` using a new proper formula.

   2.3. Retain the existing loop over the listOfTeachers array to distribute payments directly within the function.

```
 function graduateAndUpgrade(address _levelTwo, bytes memory) public onlyPrincipal
{
        if (_levelTwo == address(0)) {
            revert HH__ZeroAddress();
        }

        uint256 totalTeachers = listOfTeachers.length;

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

```diff
+       uint256 teacherTotalPay = (bursary * TEACHER_WAGE) / PRECISION;
+       uint256 payPerTeacher = teacherTotalPay / totalTeachers;
        uint256 principalPay = (bursary * PRINCIPAL_WAGE) / PRECISION;

        _authorizeUpgrade(_levelTwo);


        for (uint256 n = 0; n < totalTeachers; n++) {
            usdc.safeTransfer(listOfTeachers[n], payPerTeacher);
        }

        usdc.safeTransfer(principal, principalPay);
    }
```

Mediums

[M-1] Lack of Access Control in `LevelOne::initialize` Function, Allowing Anyone to Initialize and Upgrade `LevelOne` Contract

Summary

The `LevelOne::initialize` function lacks proper access control, allowing any user to call it and potentially take ownership of the contract, which could lead to its compromise or destruction.

Vulnerability Details

The `LevelOne::initialize` function is publicly accessible and does not restrict who can invoke it. As a result, any external actor can initialize the contract at any time.

Impact

This vulnerability allows a malicious actor to initialize the contract before its intended owner, potentially assigning themselves as the principal. Once in control, they can point the proxy to a malicious implementation, compromising the system's integrity and functionality.

Proof of concept

The vulnerability can be exploited through the following steps:

1. Deploy the `LevelOne` contract using a script (e.g., `DeployLevelOne.s.sol`) that does not perform initialization.
2. An unauthorized actor, such as a disgruntled teacher `alice`, notices the contract is uninitialized.
3. `alice` calls the initialize function and assigns herself as the `principal`.
4. With the `principal` role, `alice` executes `upgradeToAndCall`, directing the proxy to a malicious `EvilImplementation` contract that includes a dummy `doNothing` function or other undesired logic.

The proxy is now pointing to malicious code, and control of the contract has effectively been seized.

Proof of code

In the `DeployLevelOne.s.sol` deployment script, the `LevelOne` contract is deployed without calling the `initialize` function, leaving it vulnerable to unauthorized initialization:

```diff
function deployLevelOne() public returns (address) {
        usdc = new MockUSDC();

        vm.startBroadcast();
        levelOneImplementation = new LevelOne();
        proxy = new ERC1967Proxy(address(levelOneImplementation), "");
-       LevelOne(address(proxy)).initialize(principal, schoolFees, address(usdc));
        vm.stopBroadcast();

        return address(proxy);
    }
```

At the end of the `LevelOneAndGraduate.t.sol` file, a malicious implementation named `EvilImplementation` is defined. This contract contains a `doNothing` function to simulate arbitrary, undesired logic and mimics a valid UUPS proxy target by implementing `proxiableUUID`.

```
contract EvilImplementation {
    function proxiableUUID() public pure returns (bytes32) {
        return 0x360894a13ba1a3210667c828492db98dca3e2076cc3735a920a3ca505d382bbc;
    }

    function doNothing() external {
    }
}
```

Then, inside the `LevelOneAndGraduateTest` contract, a test case demonstrates how a supposedly unauthorized user `alice` can initialize the proxy and redirect it to the malicious implementation:

```
    function test_anyone_can_initialize_and_brick() public {

        vm.startPrank(alice);
        levelOneProxy.initialize(alice, schoolFees, address(usdc));
        assert(levelOneProxy.getPrincipal() == alice);

        EvilImplementation evil = new EvilImplementation();
        levelOneProxy.upgradeToAndCall(
            address(evil),
            abi.encodeWithSignature("doNothing()")
        );
        vm.stopPrank();
        vm.expectRevert();
        levelOneProxy.getPrincipal();
    }
}
```

Once this test passes, it proves that the contract was initialized by an unauthorized actor `alice` and then redirected to the malicious `EvilImplementation`. As a result, subsequent calls to functions defined in the original `LevelOne` contract (like `getPrincipal`) revert, since the proxy now points to a contract lacking those definitions.

Tools Used

Slither was used to identify this vulnerability.

Recommendations

To prevent unauthorized initialization and secure the upgradeable proxy pattern, it is strongly recommended to enforce access control on the `initialize` function of the `LevelOne` contract.

1. Import `OwnableUpgradeable` from OpenZeppelin to enable access control:

```
    import {UUPSUpgradeable} from "@openzeppelin/contracts-upgradeable/proxy/utils/UUPSUpgradeable.sol";
    import {Initializable} from "@openzeppelin/contracts-upgradeable/proxy/utils/Initializable.sol";
    import {IERC20} from "@openzeppelin/contracts/token/ERC20/IERC20.sol";
    import {SafeERC20} from "@openzeppelin/contracts/token/ERC20/utils/SafeERC20.sol";
+   import {OwnableUpgradeable} from "@openzeppelin/contracts-upgradeable/access/OwnableUpgradeable.sol";
```

2. Call `__Ownable_init()` within the initialize function, before initializing UUPS:

```
    function initialize(address _principal, uint256 _schoolFees, address _usdcAddress) public initializer {
        if (_principal == address(0)) {
            revert HH__ZeroAddress();
        }
        if (_schoolFees == 0) {
            revert HH__ZeroValue();
        }
        if (_usdcAddress == address(0)) {
            revert HH__ZeroAddress();
        }

        principal = _principal;
        schoolFees = _schoolFees;
        usdc = IERC20(_usdcAddress);
+       __Ownable_init(msg.sender);
        __UUPSUpgradeable_init();
    }
```

Note: `__Ownable_init()` sets the caller `msg.sender` as the initial owner. If needed, you can pass a custom address instead of `msg.sender`.

[L-1] `LevelOne::giveReview` function does not update `LevelOne::reviewCount`, breaking the function requirements and making the contract vulnerable.

Summary

The `LevelOne::giveReview` function incorrectly handles the update of the `LevelOne::reviewCount` for students, which can break the intended checks within the function.

Specifically, the function includes a requirement that `LevelOne::reviewCount` must be less than five. However, this counter is never updated when a review is submitted, rendering the check ineffective.

Vulnerability Details

`LevelOne::giveReview` function is supposed to allow the teachers to submit only one review per student per week. However, it does not update `LevelOne::reviewCount` mapping as expected. Although other variables in the contact such as `LevelOne::lastReviewTime` or `LevelOne::sessionEnd` provide protection to it indirectly, the function should still update `LevelOne::reviewCount` every time it is called. This omission creates an inconsistency in the contract's state.

Impact

Although the current implementation restricts teachers from calling `LevelOne::giveReview` more than once per week, which decreases the likelihood of this issue being exploited, the logic of the contract becomes fragile. If variables like `LevelOne::sessionEnd` or `LevelOne::lastReviewTime` are modified, the protection breaks down.

The intended restriction (`reviewCount < 5`) is entirely bypassed, rendering the `reviewCount` mapping functionally useless.

This issue can be confirmed by adding the following test to the `LevelOneAndGraduateTest.t.sol` contract. The console will show that `"reviewCount:"`, `"reviewCount2:"`, and `"reviewCount3:"` all return `0`.

Proof of Concept:

1. Teacher `alice` submits three reviews for student `fin`.
2. After the third review, the console shows `reviewCount(fin)` is still `0`, despite three reviews being recorded.

Proof of Code

```
function test_reviewCount_does_not_increase() public schoolInSession {
    vm.warp(block.timestamp + 1 weeks);

    vm.prank(alice);
    levelOneProxy.giveReview(fin, false);
```

```
        console2.log("reviewCount: ", levelOneProxy.reviewCount(fin));

        vm.warp(block.timestamp + 1 weeks);

        vm.prank(alice);
        levelOneProxy.giveReview(fin, false);

        console2.log("reviewCount2: ", levelOneProxy.reviewCount(fin));

        vm.warp(block.timestamp + 1 weeks);

        vm.prank(alice);
        levelOneProxy.giveReview(fin, false);

        console2.log("reviewCount3: ", levelOneProxy.reviewCount(fin));
    }
```

Tools Used

This issue was found by manual review.

Recommendations

Consider to modify `LevelOne::giveReview` function to properly update `LevelOne::reviewCount` value every time the function is called:

```
function giveReview(address _student, bool review) public onlyTeacher {
        if (!isStudent[_student]) {
            revert HH__StudentDoesNotExist();
        }
        require(reviewCount[_student] < 5, "Student review count exceeded!!!");
        require(block.timestamp >= lastReviewTime[_student] + reviewTime, "Reviews
can only be given once per week");

        // where `false` is a bad review and true is a good review
        if (!review) {
            studentScore[_student] -= 10;
        }

        // Update last review time
        lastReviewTime[_student] = block.timestamp;
+       // Update
+       reviewCount[_student]++;

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