Protocol Audit Report

Table of Contents

Protocol Summary

Disclaimer

I make 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.

Risk Classification

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

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

Audit Details

Scope

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

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.

Invariants

- 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

Issues found

Severtity	Number of issues found
High	8
Medium	5
Low	2
Total	15

Findings

High

[H-1] Unprotected Initialization.

Description: The logic of `LevelOne::initialize` function is marked public initializer, but there is no call to `_disableInitializers()` in the implementation's constructor. That means anyone can call initialize on the logic contract and become principal, set arbitrary schoolFees or usdc, and then drain or destroy funds.

Impact: This runs `initialize()` in the implementation contract's own storage, sets '_initialized = 1' in the logic contract, and potentially sets 'owner = attacker'. An attacker can hijack the contract by re-initializing the logic implementation, minting themselves ownership and privileges, and either steal USDC or upgrade to malicious code. Also, the attacker can interfere with reinitializer(x) logic.

Proof of Concept: Include the following test in the `LevelOneAndGraduateTest.t.sol` file:

```
function testReinitializeLogic() public {
    // Initialize the proxy with the first implementation
    deployBot = new DeployLevelOne();
    proxyAddress = deployBot.deployLevelOne();

    address attacker = makeAddr("attacker");

    LevelOne levelOneImplementation = new LevelOne();
    // Initialize directly, not via proxy
    LevelOne(address(levelOneImplementation)).initialize(attacker, schoolFees,
address(usdc));

    assertEq(levelOneImplementation.getPrincipal(), attacker);
}
```

Even if the proxy was correctly initialized first, the logic contract is now initialized with its own state (could confuse tools like Etherscan, OpenZeppelin Defender) and has its own owner (someone may accidentally interact with the wrong address).

Recommended Mitigation: In the implementation contract's constructor, call `_disableInitializers()` to ensure initialize cannot be called on the logic contract.

```
constructor() {
+   _disableInitializers();
}
```

This prevents anyone from initializing the logic contract itself and ensures only the proxy can be initialized.

[H-2] Incorrect Teacher-Pay Calculation.

Description:

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

This computes 35% of bursary per teacher, not total. Distributing that to N teachers pays out bursary _ 0.35 _ N, which can exceed available funds and revert.

Impact: With multiple teachers, total payouts can exceed bursary, causing underflow.

Proof of Concept: Include the following test in the `LevelOneAndGraduateTest.t.sol` file:

```
function testOverpaymentRevert() public {
    // add teachers
    vm.startPrank(principal);
    levelOneProxy.addTeacher(alice);
    levelOneProxy.addTeacher(bob);
    levelOneProxy.addTeacher(harriet);
    vm.stopPrank();

    // add student
    vm.startPrank(clara);
    usdc.approve(address(levelOneProxy), schoolFees);
    levelOneProxy.enroll();
    vm.stopPrank();

    vm.prank(principal);
    levelOneProxy.startSession(70);

    levelTwoImplementation = new LevelTwo();
    levelTwoImplementationAddress = address(levelTwoImplementation);
    bytes memory data = abi.encodeCall(LevelTwo.graduate, ());

    // ERC20InsufficientBalance revert
    vm.startPrank(principal);
    vm.expectRevert();
    levelOneProxy.graduateAndUpgrade(levelTwoImplementationAddress, data);
    vm.stopPrank();
}
```

Recommended Mitigation:

```
function graduateAndUpgrade(address _levelTwo, bytes memory) public onlyPrincipal
{
    …
    uint256 totalTeachers = listOfTeachers.length;
+   uint256 totalTeacherShare = (bursary * TEACHER_WAGE) / PRECISION;
+   uint256 payPerTeacher = totalTeacherShare / totalTeachers;
-   uint256 payPerTeacher = (bursary * TEACHER_WAGE) / PRECISION;
    …
}
```

[H-3] Upgrade logic problem. (Logic Bypass + Invalid Graduation).

Description: Although `LevelOne::startSession` sets 'cutOffScore', `LevelOne::graduateAndUpgrade` never checks whether a student's score meets it. Every student end up treated the same and the '_levelTwo' address is never used to actually bridge or upgrade students.

Impact: Poor-performing students pass automatically and upgrade logic is effectively dead code. Bursary distribution can occur even if no one qualifies, and the 'Graduated' events 'levelTwo' address is never used.

Proof of Concept: Include the following test in the `LevelOneAndGraduateTest.t.sol` file:

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

    vm.prank(principal);
    levelOneProxy.startSession(200); // impossible score

    // even though studentScore < cutOffScore, graduation still proceeds
    levelTwoImplementation = new LevelTwo();
    levelTwoImplementationAddress = address(levelTwoImplementation);
    bytes memory data = abi.encodeCall(LevelTwo.graduate, ());

    vm.prank(principal);
    levelOneProxy.graduateAndUpgrade(levelTwoImplementationAddress, data);
    // No revert, bursary is distributed
}
```

Recommended Mitigation:

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

+   for (uint i = 0; i < listOfStudents.length; i++) {
+       address student = listOfStudents[i];
+       if (studentScore[student] >= cutOffScore) {
+           // optionally expel / refund / emit failure event
+       } else { LevelTwo(_levelTwo).upgradeToAndCall(student, data); }
+   }

    …
}
```

[H-4] Misused UUPS Upgrade Flow (Redundant _authorizeUpgrade + Missing upgradeToAndCall).

Description: Inside `LevelOne::graduateAndUpgrade`, the code calls `_authorizeUpgrade(_levelTwo)` directly instead of performing 'upgradeToAndCall' on the proxy. The signature doesn't accept the 'bytes data' parameter and the redundant `_authorizeUpgrade(_levelTwo)` is never sufficient to change implementation.

Impact: Students never actually get upgraded to LevelTwo, the intended proxy call is never invoked. This breaks the upgrade flow and leaves the contract in an inconsistent state.

Proof of Concept: Include the following test in the `LevelOneAndGraduateTest.t.sol` file:

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

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

    LevelTwo levelTwoProxy = LevelTwo(proxyAddress);

    vm.expectRevert();
    levelTwoProxy.TEACHER_WAGE_L2();
}
```

Recommended Mitigation:

- Change signature to function `graduateAndUpgrade(address _levelTwo, bytes memory data)`
- Remove direct `_authorizeUpgrade(_levelTwo)` call
- Use `upgradeToAndCall(_levelTwo, data)` on the proxy so that `_authorizeUpgrade` is invoked internally and the new implementation is executed immediately.

```
- function graduateAndUpgrade(address _levelTwo, bytes memory) public
onlyPrincipal {
+ function graduateAndUpgrade(address _levelTwo, bytes memory data) public
onlyPrincipal {
      …
-     _authorizeUpgrade(_levelTwo);
+     upgradeToAndCall(_levelTwo, data);
      …
}
```

[H-5] Unprotected Reinitializer on `LevelTwo::graduate()` (Access Control Bypass).

Description: `LevelTwo::graduate()` is marked public reinitializer(2) with no access control, so anyone can invoke it once the proxy is at version < 2

Impact: An attacker can prematurely trigger graduation logic or malicious hooks in a future implementation, potentially manipulating state or skipping required flows.

Recommended Mitigation: Restrict to principal:

```
- function graduate() public reinitializer(2) {
+ function graduate() public onlyPrincipal reinitializer(2) {
}
```

[H-6] Missing UUPS Inheritance in LevelTwo.

Description: `LevelTwo` imports only 'Initializable' and lacks 'UUPSUpgradeable' inheritance and `__UUPSUpgradeable_init()`

Impact: The intended UUPS proxy pattern is broken: no '_authorizeUpgrade' hook is available, and the proxy cannot be safely upgraded to a next implementation if needed.

Proof of Concept:

> Note: this PoC assumes that the 'Misused UUPS Upgrade Flow' issue has already been fixed, so that graduateAndUpgrade gets as far as splitting by totalTeachers instead of reverting earlier.

```
- function graduateAndUpgrade(address _levelTwo, bytes memory) public
onlyPrincipal {
+ function graduateAndUpgrade(address _levelTwo, bytes memory data) public
onlyPrincipal {
      …
-     _authorizeUpgrade(_levelTwo);
+     upgradeToAndCall(_levelTwo, data);
      …
}
```

After this fix include the following test in the `LevelOneAndGraduateTest.t.sol` file:

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

    vm.startPrank(principal);
    vm.expectRevert();
    levelOneProxy.graduateAndUpgrade(levelTwoImplementationAddress, "");
    vm.stopPrank();
}
```

Attempting to compile or upgrade the proxy to LevelTwo will fail due to missing functions.

Recommended Mitigation:

```
+ import {UUPSUpgradeable} from "@openzeppelin/contracts-
upgradeable/proxy/utils/UUPSUpgradeable.sol";
```

```
- contract LevelTwo is Initializable {
+ contract LevelTwo is Initializable, UUPSUpgradeable {
+ constructor() { _disableInitializers(); }
+ function _authorizeUpgrade(address newImplementation) internal override
onlyPrincipal {}
```

[H-7] Incorrect Residual Bursary Handling.

Description: After distributing 5% to the principal and 35% to teachers, the remaining 60% of the bursary should persist in the 'bursary' state variable so it can be carried over or audited. The current implementation computes only the two payout shares and never recalculates or stores the residual.

Impact: Funds corresponding to the 60% residual go untracked:

- The bursary state variable remains stale (usually still its original value or zero), breaking the invariant

```
principalPay + teacherTotal + residual == bursary
```

- Over time, "lost" bursary accumulates invisibly, preventing accurate accounting or reuse in future sessions.

Proof of Concept:

> Note: this PoC assumes that the 'Incorrect Teacher-Pay Calculation' issue has already been fixed, so that graduateAndUpgrade gets as far as splitting by totalTeachers instead of reverting earlier.

```
function graduateAndUpgrade(address _levelTwo, bytes memory) public onlyPrincipal
{
    …
    uint256 totalTeachers = listOfTeachers.length;
+   uint256 totalTeacherShare = (bursary * TEACHER_WAGE) / PRECISION;
+   uint256 payPerTeacher = totalTeacherShare / totalTeachers;
-   uint256 payPerTeacher = (bursary * TEACHER_WAGE) / PRECISION;
    …
}
```

After this fix include the following test in the `LevelOneAndGraduateTest.t.sol` file:

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

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

    // Expected residual = 60% of 30e18 = 18e18
    uint256 expectedResidual = (30e18 * 60) / 100;
    // But bursary() remains its prior value (or zero), not the expected residual
    assertTrue(levelOneProxy.bursary() != expectedResidual, "Residual 60% should
be stored in bursary");
}
```

Recommended Mitigation: After executing all transfers, compute and persist the 60% residual explicitly:

```
function graduateAndUpgrade(address _levelTwo, bytes memory) public onlyPrincipal
{
     …
+    uint256 residual = bursary - principalPay - (payPerTeacher *
listOfTeachers.length);
+    bursary = residual;
     …
}
```

[H-8] Storage Layout Mismatch (State Variable Order Inconsistency + Storage Collision).

Description: The two implementations, 'LevelOne' and 'LevelTwo', declare their state variables in different orders. When you perform a UUPS upgrade, the proxy's storage is interpreted by the new implementation according to its own layout. Mismatched layouts cause storage collisions—slots written by one implementation map to the wrong variables in the next.

Impact:

- Corrupted state: Data intended for 'bursary' in LevelOne end up in 'cutOffScore' in LevelTwo, leading to wildly incorrect behavior.
- Security bypasses: Critical invariants break, and funds or upgrades can be executed under the wrong conditions.
- Fund loss or lock: Misinterpreted 'bursary' values can result in under- or over-payment, or leave residual balances stranded.

Proof of Concept:

LevelOne storage:

Name	Type	Slot	Offset	Bytes
principal	address	0	0	20
inSession	bool	0	20	1
schoolFees	uint256	1	0	32
sessionEnd	uint256	2	0	32
bursary	uint256	3	0	32
cutOffScore	uint256	4	0	32
isTeacher	mapping(address => bool)	5	0	32

Name	Type	Slot	Offset	Bytes
isStudent	mapping(address => bool)	6	0	32
studentScore	mapping(address => uint256)	7	0	32
reviewCount	mapping(address => uint256)	8	0	32
lastReviewTime	mapping(address => uint256)	9	0	32
listOfStudents	address[]	10	0	32
listOfTeachers	address[]	11	0	32
usdc	contract IERC20	12	0	20

LevelTwo storage:

Name	Type	Slot	Offset	Bytes
principal	address	0	0	20
inSession	bool	0	20	1
sessionEnd	uint256	1	0	32
bursary	uint256	2	0	32
cutOffScore	uint256	3	0	32
isTeacher	mapping(address => bool)	4	0	32
isStudent	mapping(address => bool)	5	0	32
studentScore	mapping(address => uint256)	6	0	32
listOfStudents	address[]	7	0	32
listOfTeachers	address[]	8	0	32
usdc	contract IERC20	9	0	20

> Note: this PoC assumes that the 'Missing UUPS Inheritance in LevelTwo' issue has already been fixed, so that graduateAndUpgrade gets as far as splitting by totalTeachers instead of reverting earlier.

```diff
+ import {UUPSUpgradeable} from "@openzeppelin/contracts-upgradeable/proxy/utils/UUPSUpgradeable.sol";

- contract LevelTwo is Initializable {
+ contract LevelTwo is Initializable, UUPSUpgradeable {
+ constructor() { _disableInitializers(); }
+ function _authorizeUpgrade(address newImplementation) internal override onlyPrincipal {}
```

After this fix include the following test in the `LevelOneAndGraduateTest.t.sol` file:

```
function testBursaryAppearsAsSessionEndInV2() public schoolInSession {
    uint256 bursaryV1 = levelOneProxy.bursary();

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

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

    LevelTwo levelTwoProxy = LevelTwo(proxyAddress);

    uint256 bursaryV2 = levelTwoProxy.bursary();

    vm.expectRevert();
    assertEq(bursaryV1, bursaryV2, "Bursary should be equal to bursary in V2");
}
```

Recommended Mitigation: Adopt a stable storage layout pattern across all versions:

1. Keep variable order identical in both implementations.
2. Insert new state variables only at the end of the layout.
3. Reserve gaps for future growth, e.g.:

```
// after existing variables
uint256[50] private __gap;
```

This ensures that every storage slot's meaning remains consistent through each upgrade, preventing collisions and preserving all invariants.

Medium

[M-1] Missing Session Guard on 'removeTeacher' (Improper Access Control + Logic Corruption).

Description: `LevelOne::removeTeacher` lacks the 'notYetInSession' modifier. A malicious or mistaken principal can remove teachers in the middle of an active session.

Impact: If all teachers are removed mid-session, `LevelOne::graduateAndUpgrade` will compute 'totalTeachers == 0', causing a division-by-zero or a zero-teacher payout—either way, graduation will revert and funds lock up.

Proof of Concept:

> Note: this PoC assumes that the 'Incorrect Teacher-Pay Calculation' issue has already been fixed, so that graduateAndUpgrade gets as far as splitting by totalTeachers instead of reverting earlier.

```
function graduateAndUpgrade(address _levelTwo, bytes memory) public onlyPrincipal
{
    …
    uint256 totalTeachers = listOfTeachers.length;
+   uint256 totalTeacherShare = (bursary * TEACHER_WAGE) / PRECISION;
+   uint256 payPerTeacher = totalTeacherShare / totalTeachers;
-   uint256 payPerTeacher = (bursary * TEACHER_WAGE) / PRECISION;
    …
}
```

After this fix include the following test in the `LevelOneAndGraduateTest.t.sol` file:

```
function testGraduateReverts() public schoolInSession {
    vm.startPrank(principal);
    levelOneProxy.removeTeacher(alice);
    levelOneProxy.removeTeacher(bob);
    vm.stopPrank();

    levelTwoImplementation = new LevelTwo();
    levelTwoImplementationAddress = address(levelTwoImplementation);
    bytes memory data = abi.encodeCall(LevelTwo.graduate, ());

    vm.startPrank(principal);
    vm.expectRevert();
    levelOneProxy.graduateAndUpgrade(levelTwoImplementationAddress, data);
    vm.stopPrank();
}
```

Recommended Mitigation: Add 'notYetInSession' to 'removeTeacher':

```
- function removeTeacher(address _teacher) public onlyPrincipal {
+ function removeTeacher(address _teacher) public onlyPrincipal notYetInSession {
    …
}
```

[M-2] Missing 'reviewCount' Increment & Underflow.

Description: In `LevelOne::giveReview`, 'reviewCount[_student]' is checked but never incremented. Teachers can issue unlimited "bad" reviews. Every time '!review', 'studentScore' drops by 10, and can underflow below zero-causing a revert in Solidity 0.8+ and permanent lock.

Impact: A teacher can repeatedly call `giveReview(student, false)` (respecting the 1-week interval) until 'studentScore' underflows, blocking further reviews or graduation and locking that student in the system.

Proof of Concept: Include the following test in the `LevelOneAndGraduateTest.t.sol` file:

```
function testUnderflowReview() public schoolInSession {
    // studentScore == 100, each bad review -10
    for (uint256 i = 0; i < 10; i++) {
        vm.warp(block.timestamp + levelOneProxy.reviewTime());
        vm.prank(alice);
        levelOneProxy.giveReview(harriet, false);
    }

    // next call underflows
    vm.warp(block.timestamp + levelOneProxy.reviewTime());
    vm.startPrank(alice);
    vm.expectRevert();
    levelOneProxy.giveReview(harriet, false);
    vm.stopPrank();
}
```

Recommended Mitigation:

```
function giveReview(address _student, bool review) public onlyTeacher {
    …
    require(reviewCount[_student] < 5, "Student review count exceeded!!!");
    require(block.timestamp >= lastReviewTime[_student] + reviewTime, "Reviews can
only be given once per week");
+   reviewCount[_student] += 1;

    if (!review) {
+       require(studentScore[_student] >= 10, "Score cannot go negative");
        studentScore[_student] -= 10;
    }
    …
}
```

[M-3] Students Graduate with Fewer Than Four Reviews.

Description: `LevelOne::graduateAndUpgrade` does not verify that each enrolled student has received the mandated four weekly reviews before allowing graduation. Although 'giveReview' tracks individual review calls, there is no check in the graduation logic to ensure 'reviewCount[student] >= 4' (or exactly 4) prior to performing upgrades and payouts.

Impact: A principal can invoke `graduateAndUpgrade` after only one, two, or three reviews per student (or even zero), allowing under-reviewed and potentially under-qualified students to graduate. This completely subverts the project's 4-week, 4-review business invariant and may lead to undeserved upgrades, misallocated bursaries, and violation of academic rules.

Proof of Concept: Include the following test in the `LevelOneAndGraduateTest.t.sol` file:

```
function testGraduateWithOnlyOneReview() public schoolInSession {
    vm.warp(block.timestamp + 1 weeks);
    vm.prank(alice);
    levelOneProxy.giveReview(harriet, true);

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

    vm.prank(principal);
    levelOneProxy.graduateAndUpgrade(levelTwoImplementationAddress, "");
    // graduation bypassed the 4-review requirement
    assertTrue(true);
}
```

Recommended Mitigation: Before any upgrade or payout in `graduateAndUpgrade`, enforce per-student review counts:

```
function graduateAndUpgrade(address _levelTwo, bytes calldata data)
    public onlyPrincipal notYetInSession
{
+   for (uint i = 0; i < listOfStudents.length; i++) {
+       address s = listOfStudents[i];
+       require(reviewCount[s] == 4,
+           "Each student must receive 4 weekly reviews");
+       require(studentScore[s] >= cutOffScore,
+           "Student did not meet cutoff score");
   }
    …
}
```

[M-4] Missing Session Duration Enforcement.

Description: `LevelOne::graduateAndUpgrade` does not verify that the current block timestamp has reached the scheduled 'sessionEnd' (which should be set to start + 4 weeks). Without a 'require(block.timestamp >= sessionEnd)' guard, the principal can prematurely call graduation logic at any time.

Impact: A principal can graduate and pay out students and teachers before the full 4-week session has elapsed, completely circumventing the intended duration requirement. This allows under-reviewed students to advance, accelerates payouts improperly, and breaks the core business rule that each session must run its full term.

Proof of Concept: Include the following test in the `LevelOneAndGraduateTest.t.sol` file:

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

    vm.prank(alice);
    levelOneProxy.giveReview(harriet, true);
```

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

        // attempt to graduate early — currently succeeds but should revert
        vm.prank(principal);
        levelOneProxy.graduateAndUpgrade(levelTwoImplementationAddress, "");
        // the session-duration invariant bypassed
        assertTrue(true);
    }
```

Recommended Mitigation: Add a timestamp check at the top of graduateAndUpgrade to enforce the 4-week session length:

```
function graduateAndUpgrade(address _levelTwo, bytes memory data) public
onlyPrincipal {
{
+   require(block.timestamp >= sessionEnd, "Session not ended");
    …
}
```

[M-5] Session Flags Not Reset Before Upgrade.

Description: After `graduateAndUpgrade` completes, the contract does not clear or reset session-related state variables (inSession) - to allow a fresh 4-week cycle in the upgraded implementation. Currently, none of these flags are reset, so the proxy's storage carries stale session state into V2.

Impact: Graduation logic in V2 may erroneously think the prior session is still active.

Proof of Concept:

> Note: this PoC assumes that the 'Missing UUPS Inheritance in LevelTwo' issue has already been fixed, so that graduateAndUpgrade gets as far as splitting by totalTeachers instead of reverting earlier.

```
+ import {UUPSUpgradeable} from "@openzeppelin/contracts-
upgradeable/proxy/utils/UUPSUpgradeable.sol";

- contract LevelTwo is Initializable {
+ contract LevelTwo is Initializable, UUPSUpgradeable {
+ constructor() { _disableInitializers(); }
+ function _authorizeUpgrade(address newImplementation) internal override
onlyPrincipal {}
```

After this fix include the following test in the `LevelOneAndGraduateTest.t.sol` file:

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

```
        vm.prank(principal);
        levelOneProxy.graduateAndUpgrade(levelTwoImplementationAddress, "");

        LevelTwo levelTwoProxy = LevelTwo(proxyAddress);

        assertTrue(levelTwoProxy.inSession(), "Session should have been reset");
    }
```

Recommended Mitigation: At end of `graduateAndUpgrade`:

```
+   inSession = false;

    // Perform the UUPS upgrade
     upgradeToAndCall(_levelTwo, data);
```

Low

[L-1] Precision Loss in 'principalPay'.

Description: Standard integer division discards the remainder ("dust"), leaving small amounts of USDC stranded in the contract.

```
uint256 principalPay = (bursary * PRINCIPAL_WAGE) / PRECISION;
```

Impact: Tiny balances accumulate and are never distributed, breaking the invariant 'principalPay + teacherTotal == bursary'. Over time, those "lost" tokens can grow significant.

Proof of Concept:

```
// enroll students so bursary == 101
// PRINCIPAL_WAGE = 5, so principalPay == 5 (101*5/100)
// dust == 101 - 5 - teacherTotal
// assert dust > 0 remains
```

Recommended Mitigation: Either use full-precision from OpenZeppelin:

```
Math.mulDiv(bursary, PRINCIPAL_WAGE, PRECISION)
```

or compute:

```
uint256 teacherTotal = payPerTeacher * totalTeachers;
uint256 principalPay = bursary - teacherTotal;
```

[L-2] Loop-Based Teacher Payout (External Call in Loop + Denial-of-Service).

Description: In `LevelOne::graduateAndUpgrade`, the contract iterates over 'listOfTeachers' and calls 'usdc.safeTransfer(teacher, payPerTeacher)' in a single loop. Each transfer is an external call: if any teacher's address reverts (e.g. a malicious contract or a frozen account) or if the loop runs out of gas due to a large list, the entire graduation operation will revert. This blocks payouts for all teachers and the principal, and can permanently lock funds in the contract.

Impact:

- A single malicious or misbehaving teacher can deny service to all other participants, preventing graduation and locking up the bursary.
- If the teacher list grows too large, the loop may consume more gas than the block limit, causing out-of-gas reverts and effectively a DoS.
- The principal cannot complete graduation, so neither teachers nor students can progress or be paid.

Recommended Mitigation: Use the pull-payment pattern instead of pushing payments in a loop. Record each teacher’s owed amount in a mapping and allow them to withdraw asynchronously:

```diff
contract LevelOne {
+   mapping(address => uint256) public pendingTeacherWithdrawals;

    function graduateAndUpgrade(address _levelTwo, bytes memory data) public
onlyPrincipal {
        for (uint i = 0; i < listOfTeachers.length; i++) {
-           usdc.safeTransfer(listOfTeachers[i], payPerTeacher);
+           pendingTeacherWithdrawals[listOfTeachers[i]] += payPerTeacher;
        }
        …
    }

+   // Withdraw teacher wages
+   function withdrawTeacherWages() external {
+       uint256 amount = pendingTeacherWithdrawals[msg.sender];
+       require(amount > 0, "No funds to withdraw");
+       pendingTeacherWithdrawals[msg.sender] = 0;
+       usdc.safeTransfer(msg.sender, amount);
+   }
}
```