QONE Token Issuance

AUDIT REPORT

Version 1.0.0

Serial No. 20260203000012018

Presented by Fairyproof

February 3, 2026

01. Introduction

This document includes the results of the audit performed by the Fairyproof team on the QONE Token Issuance project.

Audit Start Time:

February 1, 2026

Audit End Time:

February 3, 2026

Audited Code's Github Repository:

https://github.com/01-Quantum/qlabs-contracts

Audited Code's Github Commit Number When Audit Started:

4dc0b3c47ea37ee069a1a5561379867f1435fc8f

Audited Code's Github Commit Number When Audit Ended:

4dc0b3c47ea37ee069a1a5561379867f1435fc8f

Audited Source Files:

The source files audited include all the files as follows:

```
./contracts/
├── DummyAuthorizer.sol
├── interfaces
│   └── IPQCAuthorizer.sol
├── QONE-V2.sol
└── TokenVestingV2.sol

2 directories, 4 files


Files not included:`Falcon512Authorizer.sol` and `falcon_512_test.sol`
```

The goal of this audit is to review QONE's solidity implementation for its Token Issuance function, study potential security vulnerabilities, its general design and architecture, and uncover bugs that could compromise the software in production.

We make observations on specific areas of the code that present concrete problems, as well as general observations that traverse the entire codebase horizontally, which could improve its quality as a whole.

This audit only applies to the specified code, software or any materials supplied by the QONE team for specified versions. Whenever the code, software, materials, settings, environment etc is changed, the comments of this audit will no longer apply.

— Disclaimer

Note that as of the date of publishing, the contents of this report reflect the current understanding of known security patterns and state of the art regarding system security. You agree that your access and/or use, including but not limited to any associated services, products, protocols, platforms, content, and materials, will be at your sole risk.

The review does not extend to the compiler layer, or any other areas beyond the programming language, or other programming aspects that could present security risks. If the audited source files are smart contract files, risks or issues introduced by using data feeds from offchain sources are not extended by this review either.

Given the size of the project, the findings detailed here are not to be considered exhaustive, and further testing and audit is recommended after the issues covered are fixed.

To the fullest extent permitted by law, we disclaim all warranties, expressed or implied, in connection with this report, its content, and the related services and products and your use thereof, including, without limitation, the implied warranties of merchantability, fitness for a particular purpose, and non-infringement.

We do not warrant, endorse, guarantee, or assume responsibility for any product or service advertised or offered by a third party through the product, any open source or third-party software, code, libraries, materials, or information linked to, called by, referenced by or accessible through the report, its content, and the related services and products, any hyperlinked websites, any websites or mobile applications appearing on any advertising, and we will not be a party to or in any way be responsible for monitoring any transaction between you and any third-party providers of products or services.

FOR AVOIDANCE OF DOUBT, THE REPORT, ITS CONTENT, ACCESS, AND/OR USAGE THEREOF, INCLUDING ANY ASSOCIATED SERVICES OR MATERIALS, SHALL NOT BE CONSIDERED OR RELIED UPON AS ANY FORM OF FINANCIAL, INVESTMENT, TAX, LEGAL, REGULATORY, OR OTHER ADVICE.

— Methodology

The above files' code was studied in detail in order to acquire a clear impression of how the its specifications were implemented. The codebase was then subject to deep analysis and scrutiny, resulting in a series of observations. The problems and their potential solutions are discussed in this document and, whenever possible, we identify common sources for such problems and comment on them as well.

The Fairyproof auditing process follows a routine series of steps:

1. Code Review, Including:

- Project Diagnosis

Understanding the size, scope and functionality of your project's source code based on the specifications, sources, and instructions provided to Fairyproof.

- Manual Code Review

Reading your source code line-by-line to identify potential vulnerabilities.

- Specification Comparison

Determining whether your project's code successfully and efficiently accomplishes or executes its functions according to the specifications, sources, and instructions provided to Fairyproof.

2. Testing and Automated Analysis, Including:

- Test Coverage Analysis

Determining whether the test cases cover your code and how much of your code is exercised or executed when test cases are run.

- Symbolic Execution

Analyzing a program to determine the specific input that causes different parts of a program to execute its functions.

3. Best Practices Review

Reviewing the source code to improve maintainability, security, and control based on the latest established industry and academic practices, recommendations, and research.

— Structure of the document

This report contains a list of issues and comments on all the above source files. Each issue is assigned a severity level based on the potential impact of the issue and recommendations to fix it, if applicable. For ease of navigation, an index by topic and another by severity are both provided at the beginning of the report.

— Documentation

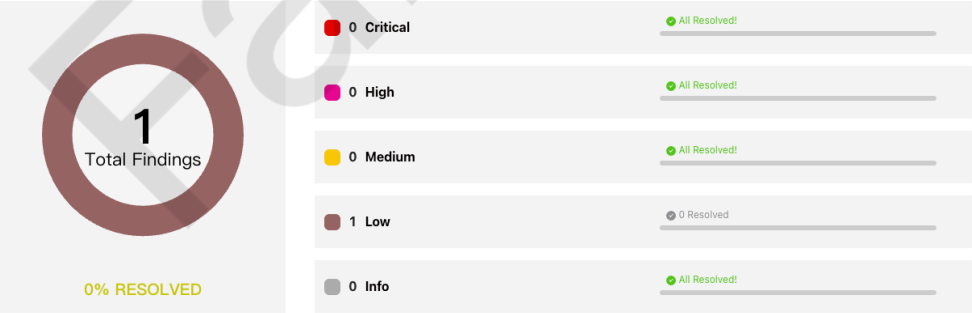
For this audit, we used the following source(s) of truth about how the token issuance function should work:

Source Code: https://github.com/01-Quantum/qlabs-contracts

These were considered the specification, and when discrepancies arose with the actual code behavior, we consulted with the QONE team or reported an issue.

— Comments from Auditor

Serial Number	Auditor	Audit Time	Result
2026020300012018	Fairyproof Security Team	Feb 1, 2026 - Feb 3, 2026	Low Risk



Summary:

The Fairyproof security team used its auto analysis tools and manual work to audit the project. During the audit, one issue of low-severity was uncovered. The QONE team acknowledged all the issues.

02. About Fairyproof

Fairyproof is a leading technology firm in the blockchain industry, providing consulting and security audits for organizations. Fairyproof has developed industry security standards for designing and deploying blockchain applications.

03. Introduction to QONE

QONE Token is a fixed-supply (1 billion tokens) ERC20 token with an integrated upgradeable authorization mechanism for token transfers. The token implements a two-phase security model:

Phase 1 (Current): Utilizes a DummyAuthorizer that approves all transfers, allowing the token to function as a standard ERC20 without any transfer restrictions.

Phase 2 (Future): The project plans to upgrade to a more sophisticated authorizer to control token transfer permissions. This design is similar to an upgradeable whitelist/blacklist mechanism but uses an external authorization contract instead of simple address lists.

The key design feature is that the token contract address remains unchanged between phases—only the authorizer contract is upgraded—ensuring continuity for DEX liquidity pools, vesting contracts, and wallet integrations.

The above description is quoted from relevant documents of QONE.

04. Major functions of audited code

The audited code includes two functions:

1. **Token Issuance**

The audited code mainly implements a token issuance function. Here are the details:

- Token Standard: ERC20
- Token Name: QONE
- Token Symbol: QONE
- Decimals: 18
- Current Supply: 1_000_000_000
- Max Supply: 1_000_000_000
- Blacklist: Yes

Note:

The audited token implementation is **QONE**, a fixed-supply ERC20 token that integrates an upgradeable external transfer authorization module based on the `IPQCAuthorizer` interface. In a typical Phase 1 deployment, a DummyAuthorizer is used that approves all transfers, allowing the token to behave as a standard ERC20 while keeping the upgrade path to a stricter PQC authorizer.

To make future authorizer upgrades safer, QONE builds in the following mechanisms:

- A 48-hour timelock for authorizer upgrades (propose → wait → finalize)
- Interface validation for new authorizers (basic `IPQCAuthorizer` sanity check)

- A `probePendingAuthorizer` view function to test authorization results of the pending authorizer for arbitrary accounts before finalizing the upgrade
- The ability to cancel a pending upgrade before the timelock expires

These mechanisms are designed to reduce the risk that a misconfigured or buggy authorizer causes a global transfer lock when moving from the permissive DummyAuthorizer configuration to a production PQC authorizer, and they are covered by additional unit tests in the test suite (QONE.t.sol).

2. **Token Vesting**

The TokenVesting contract implements a Merkle-tree-based linear token vesting mechanism supporting 6 distinct vesting schedule categories for different token allocation roles. The contract adopts a ***self-service initialization and claiming model***, where users must provide Merkle proofs to initialize their allocations and claim unlocked tokens independently.

The vesting mechanism consists of three phases:

1. ***TGE Unlock***: A fixed percentage of tokens unlocks immediately at TGE (Token Generation Event)
2. ***Cliff Period***: A lock-up period where no additional tokens unlock beyond the TGE portion
3. ***Linear Vesting***: After the cliff ends, remaining tokens unlock uniformly over time

Formula:

```
Vested Amount = TGE Unlock + (Remaining Pool × Time Elapsed / Vesting
Duration)
Claimable Amount = Vested Amount - Already Claimed
```

Vesting Categories (TokenVestingV2, 6 schedules)

Category	Role	TGE Unlock	Cliff Period	Linear Vesting Period	Total Duration
0	Token Sale Tier 1	15%	3 months	9 months	12 months
1	Token Sale Tier 2	15%	0	12 months	12 months
2	Token Sale Tier 3	100%	0	0 (instant)	0
3	Team & Advisors	20%	6 months	24 months	30 months
4	Liquidity/Treasury	50%	0	36 months	36 months
5	Community Airdrops	9%	0	24 months	24 months

05. Coverage of issues

The issues that the Fairyproof team covered when conducting the audit include but are not limited to the following ones:

- Access Control
- Admin Rights
- Arithmetic Precision
- Code Improvement
- Contract Upgrade/Migration

- Delete Trap
- Design Vulnerability
- DoS Attack
- EOA Call Trap
- Fake Deposit
- Function Visibility
- Gas Consumption
- Implementation Vulnerability
- Inappropriate Callback Function
- Injection Attack
- Integer Overflow/Underflow
- IsContract Trap
- Miner's Advantage
- Misc
- Price Manipulation
- Proxy selector clashing
- Pseudo Random Number
- Re-entrancy Attack
- Replay Attack
- Rollback Attack
- Shadow Variable
- Slot Conflict
- Token Issuance
- Tx.origin Authentication
- Uninitialized Storage Pointer

06. Severity level reference

Every issue in this report was assigned a severity level from the following:

Critical severity issues need to be fixed as soon as possible.

High severity issues will probably bring problems and should be fixed.

Medium severity issues could potentially bring problems and should eventually be fixed.

Low severity issues are minor details and warnings that can remain unfixed but would be better fixed at some point in the future.

Informational is not an issue or risk but a suggestion for code improvement.

07. Major areas that need attention

Based on the provided source code the Fairyproof team focused on the possible issues and risks related to the following functions or areas.

- Function Implementation

We checked whether or not the functions were correctly implemented.
We didn't find issues or risks in these functions or areas at the time of writing.

- Access Control

We checked each of the functions that could modify a state, especially those functions that could only be accessed by owner or administrator
We found one issue, for more details please refer to [FP-1] in "09. Issue description".

- Token Issuance & Transfer

We examined token issuance and transfers for situations that could harm the interests of holders.
We didn't find issues or risks in these functions or areas at the time of writing.

- State Update

We checked some key state variables which should only be set at initialization.
We didn't find issues or risks in these functions or areas at the time of writing.

- Asset Security

We checked whether or not all the functions that transfer assets were safely handled.
We didn't find issues or risks in these functions or areas at the time of writing.

- Miscellaneous

We checked the code for optimization and robustness.
We didn't find issues or risks in these functions or areas at the time of writing.

08. List of issues by severity

Index	Title	Issue/Risk	Severity	Status
FP-1	Authorizer Upgrade May Result in Transfer Lock if Misconfigured	Access Control	Low	Acknowledged

09. Issue descriptions

[FP-1] Authorizer Upgrade May Result in Transfer Lock if Misconfigured

Access Control Low Acknowledged

Issue/Risk: Access Control

Description:

The QONE token implements a two-phase authorization model where all transfers must pass the `pqcAuthorizer.isAuthorized()` check. In Phase 1, the DummyAuthorizer always returns true, allowing unrestricted transfers. However, when upgrading to Phase 2, if the new authorizer is misconfigured or contains bugs that cause isAuthorized() to:

- Always return false
- Revert unexpectedly
- Consume excessive gas

All token transfers will be frozen, as the `_update()` function enforces authorization for every transfer:

```
function _update(
        address from,
        address to,
        uint256 amount
    ) internal override {
        // Skip authorization check for minting
        if (from != address(0)) {
            require(
                pqcAuthorizer.isAuthorized(from),
                "QONE: sender not authorized (PQC)"
            );
        }

        super._update(from, to, amount);
    }
```

Current Status: This is rated as Low Risk because the project is currently in Phase 1 with no immediate plans to upgrade the authorizer. However, this becomes a critical consideration when transitioning to Phase 2.

Additional Consideration: If QONE.owner() does not match the owner address expected by the new authorizer, subsequent authorizer upgrades may fail, creating a permanent lock situation.

Recommendation:

1. Thorough Testing: Before upgrading to a new authorizer in Phase 2, conduct extensive testing on testnets to ensure:

The authorizer correctly implements the IPQCAuthorizer interface
isAuthorized() returns expected results for both authorized and unauthorized users
Gas consumption is reasonable
Edge cases (zero address, contract addresses, etc.) are handled properly

2. Multi-Signature Control: Transfer ownership of the QONE contract to a multi-signature wallet (e.g., Gnosis Safe with 3/5 or 4/7 threshold) before Phase 2 deployment to prevent single-point-of-failure during critical upgrades.
3. Owner Consistency Verification: Ensure that QONE.owner() matches the owner address recognized by the new authorizer to prevent future upgrade failures.

Update/Status:

The QONE team has acknowledged this issue.

10. Recommendations to enhance the overall security

We list some recommendations in this section. They are not mandatory but will enhance the overall security of the system if they are adopted.

- Consider managing the owner's access control with great care and transfering it to a multi-sig wallet or DAO when necessary.

11. Appendices

11.1 Unit Test

1. QONE.t.sol

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.20;

import "forge-std/Test.sol";
import "../src/QONE-V2.sol";
import "../src/DummyAuthorizer.sol";
import "../src/interfaces/IPQCAuthorizer.sol";

/**
 * @title QONETest
 * @notice Comprehensive test suite for the QONE token system with timelock
 * @dev Tests cover:
 *      - Token deployment and initial state
 *      - ERC20 standard functionality
 *      - PQC authorization with DummyAuthorizer
 *      - Timelock-protected authorizer upgrades
 *      - Edge cases and security (including FP-1 audit issue)
 */
contract QONETest is Test {
    QONE public token;
```

```
    DummyAuthorizer public dummyAuthorizer;

    address public owner;
    address public user1;
    address public user2;
    address public user3;

    // Events to test
    event Transfer(address indexed from, address indexed to, uint256 value);
    event PQCAuthorizerUpdated(address indexed newAuthorizer);
    event PQCAuthorizerUpgradeProposed(address indexed pendingAuthorizer,
uint256 unlockTime);
    event PQCAuthorizerUpgradeCancelled(address indexed oldPending);
    event UpgradeAuthorized(
        address indexed tokenContract,
        address indexed oldAuthorizer,
        address indexed newAuthorizer
    );

    function setUp() public {
        owner = address(this);
        user1 = address(0x1);
        user2 = address(0x2);
        user3 = address(0x3);

        // Deploy DummyAuthorizer
        dummyAuthorizer = new DummyAuthorizer(owner);

        // Deploy QONE token
        token = new QONE(owner, address(dummyAuthorizer));
    }

    /*//////////////////////////////////////////////////////////////
                        DEPLOYMENT TESTS
    //////////////////////////////////////////////////////////////*/

    function test_Deployment() public {
        assertEq(token.name(), "QONE");
        assertEq(token.symbol(), "QONE");
        assertEq(token.decimals(), 18);
        assertEq(token.totalSupply(), 1_000_000_000 * 1e18);
        assertEq(token.balanceOf(owner), 1_000_000_000 * 1e18);
        assertEq(address(token.pqcAuthorizer()), address(dummyAuthorizer));
        assertEq(token.owner(), owner);
        assertEq(token.TIMELOCK_DURATION(), 48 hours);
        assertEq(token.pendingPQCAuthorizer(), address(0));
        assertEq(token.upgradeTimelock(), 0);
    }

    function test_RevertWhen_DeployWithZeroAuthorizer() public {
        vm.expectRevert("QONE: authorizer is zero address");
        new QONE(owner, address(0));
    }

    /*//////////////////////////////////////////////////////////////
                        ERC20 FUNCTIONALITY TESTS
    //////////////////////////////////////////////////////////////*/

    function test_Transfer() public {
        uint256 amount = 1000 * 1e18;

        vm.expectEmit(true, true, false, true);
        emit Transfer(owner, user1, amount);
```

```
        token.transfer(user1, amount);

        assertEq(token.balanceOf(user1), amount);
        assertEq(token.balanceOf(owner), token.TOTAL_SUPPLY() - amount);
    }

    function test_TransferFrom() public {
        uint256 amount = 1000 * 1e18;

        // Approve user2 to spend owner's tokens
        token.approve(user2, amount);
        assertEq(token.allowance(owner, user2), amount);

        // User2 transfers tokens from owner to user1
        vm.prank(user2);
        token.transferFrom(owner, user1, amount);

        assertEq(token.balanceOf(user1), amount);
        assertEq(token.balanceOf(owner), token.TOTAL_SUPPLY() - amount);
        assertEq(token.allowance(owner, user2), 0);
    }

    function test_MultipleTransfers() public {
        uint256 amount = 500 * 1e18;

        // Owner -> User1
        token.transfer(user1, amount);
        assertEq(token.balanceOf(user1), amount);

        // User1 -> User2
        vm.prank(user1);
        token.transfer(user2, amount / 2);
        assertEq(token.balanceOf(user2), amount / 2);
        assertEq(token.balanceOf(user1), amount / 2);

        // User2 -> User3
        vm.prank(user2);
        token.transfer(user3, amount / 4);
        assertEq(token.balanceOf(user3), amount / 4);
    }

    /*//////////////////////////////////////////////////////////////
                    TIMELOCK AUTHORIZER UPGRADE TESTS
    //////////////////////////////////////////////////////////////*/

    function test_ProposePQCAuthorizer() public {
        DummyAuthorizer newAuthorizer = new DummyAuthorizer(owner);
        uint256 expectedUnlockTime = block.timestamp +
token.TIMELOCK_DURATION();

        vm.expectEmit(true, false, false, true);
        emit PQCAuthorizerUpgradeProposed(address(newAuthorizer),
expectedUnlockTime);

        token.proposePQCAuthorizer(address(newAuthorizer));

        assertEq(token.pendingPQCAuthorizer(), address(newAuthorizer));
        assertEq(token.upgradeTimelock(), expectedUnlockTime);
    }

    function test_RevertWhen_ProposePQCAuthorizerZeroAddress() public {
        vm.expectRevert("QONE: new authorizer is zero address");
```

```
        token.proposePQCAuthorizer(address(0));
    }

    function test_RevertWhen_ProposePQCAuthorizerWhilePendingExists() public {
        DummyAuthorizer newAuthorizer1 = new DummyAuthorizer(owner);
        DummyAuthorizer newAuthorizer2 = new DummyAuthorizer(owner);

        token.proposePQCAuthorizer(address(newAuthorizer1));

        vm.expectRevert("QONE: pending authorizer exists");
        token.proposePQCAuthorizer(address(newAuthorizer2));
    }

    function test_RevertWhen_ProposePQCAuthorizerByNonOwner() public {
        DummyAuthorizer newAuthorizer = new DummyAuthorizer(owner);

        vm.prank(user1);
        vm.expectRevert();
        token.proposePQCAuthorizer(address(newAuthorizer));
    }

    function test_CancelPendingUpdate() public {
        DummyAuthorizer newAuthorizer = new DummyAuthorizer(owner);
        token.proposePQCAuthorizer(address(newAuthorizer));

        assertEq(token.pendingPQCAuthorizer(), address(newAuthorizer));

        vm.expectEmit(true, false, false, false);
        emit PQCAuthorizerUpgradeCancelled(address(newAuthorizer));

        token.cancelPendingUpdate();

        assertEq(token.pendingPQCAuthorizer(), address(0));
        assertEq(token.upgradeTimelock(), 0);
    }

    function test_RevertWhen_CancelPendingUpdateWithNoPending() public {
        vm.expectRevert("QONE: no pending authorizer");
        token.cancelPendingUpdate();
    }

    function test_FinalizePQCAuthorizer() public {
        DummyAuthorizer newAuthorizer = new DummyAuthorizer(owner);

        // Propose
        token.proposePQCAuthorizer(address(newAuthorizer));

        // Fast forward past timelock
        vm.warp(block.timestamp + token.TIMELOCK_DURATION());

        vm.expectEmit(true, false, false, false);
        emit PQCAuthorizerUpdated(address(newAuthorizer));

        // Finalize
        token.finalizePQCAuthorizer("");

        assertEq(address(token.pqcAuthorizer()), address(newAuthorizer));
        assertEq(token.pendingPQCAuthorizer(), address(0));
        assertEq(token.upgradeTimelock(), 0);
    }

    function test_RevertWhen_FinalizePQCAuthorizerNoPending() public {
        vm.expectRevert("QONE: no pending authorizer");
```

```
        token.finalizePQCAuthorizer("");
    }

    function test_RevertWhen_FinalizePQCAuthorizerBeforeTimelock() public {
        DummyAuthorizer newAuthorizer = new DummyAuthorizer(owner);

        token.proposePQCAuthorizer(address(newAuthorizer));

        // Try to finalize immediately
        vm.expectRevert("QONE: timelock not expired");
        token.finalizePQCAuthorizer("");

        // Try after 24 hours (still before 48 hour timelock)
        vm.warp(block.timestamp + 24 hours);
        vm.expectRevert("QONE: timelock not expired");
        token.finalizePQCAuthorizer("");
    }

    function test_TransfersWorkDuringPendingUpgrade() public {
        DummyAuthorizer newAuthorizer = new DummyAuthorizer(owner);
        token.proposePQCAuthorizer(address(newAuthorizer));

        // Transfers should still work with current authorizer
        uint256 amount = 1000 * 1e18;
        token.transfer(user1, amount);
        assertEq(token.balanceOf(user1), amount);
    }

    /*//////////////////////////////////////////////////////////////
                [FP-1] AUTHORIZER MISCONFIGURATION TESTS
    //////////////////////////////////////////////////////////////*/

    function test_ProbeFunction_ValidAuthorizer() public {
        DummyAuthorizer newAuthorizer = new DummyAuthorizer(owner);
        token.proposePQCAuthorizer(address(newAuthorizer));

        (QONE.ProbeStatus status, address pending) =
token.probePendingAuthorizer(user1);

        assertEq(uint8(status), uint8(QONE.ProbeStatus.OK));
        assertEq(pending, address(newAuthorizer));
    }

    function test_ProbeFunction_BrokenAuthorizer() public {
        // BrokenAuthorizer will be rejected during proposal due to interface
check
        BrokenAuthorizer brokenAuth = new BrokenAuthorizer();

        vm.expectRevert("QONE: invalid authorizer interface");
        token.proposePQCAuthorizer(address(brokenAuth));
    }

    function test_ProbeFunction_AlwaysFalseAuthorizer() public {
        AlwaysFalseAuthorizer alwaysFalse = new AlwaysFalseAuthorizer(owner);
        token.proposePQCAuthorizer(address(alwaysFalse));

        (QONE.ProbeStatus status, address pending) =
token.probePendingAuthorizer(user1);

        assertEq(uint8(status), uint8(QONE.ProbeStatus.NOT_AUTHORIZED));
        assertEq(pending, address(alwaysFalse));
    }
```

```
    function test_RevertWhen_ProbeWithNoPendingAuthorizer() public {
        vm.expectRevert("QONE: no pending authorizer");
        token.probePendingAuthorizer(user1);
    }

    function test_CompleteUpgradeFlow_DetectMisconfiguration() public {
        // Step 1: Propose a misconfigured authorizer
        AlwaysFalseAuthorizer badAuthorizer = new
AlwaysFalseAuthorizer(owner);
        token.proposePQCAuthorizer(address(badAuthorizer));

        // Step 2: Test with probe BEFORE finalizing
        (QONE.ProbeStatus status,) = token.probePendingAuthorizer(owner);
        assertEq(uint8(status), uint8(QONE.ProbeStatus.NOT_AUTHORIZED));

        // Step 3: Cancel the bad upgrade
        token.cancelPendingUpdate();

        // Step 4: Propose a good authorizer
        DummyAuthorizer goodAuthorizer = new DummyAuthorizer(owner);
        token.proposePQCAuthorizer(address(goodAuthorizer));

        // Step 5: Verify with probe
        (status,) = token.probePendingAuthorizer(owner);
        assertEq(uint8(status), uint8(QONE.ProbeStatus.OK));

        // Step 6: Wait for timelock and finalize
        vm.warp(block.timestamp + token.TIMELOCK_DURATION());
        token.finalizePQCAuthorizer("");

        // Step 7: Verify transfers still work
        token.transfer(user1, 1000 * 1e18);
        assertEq(token.balanceOf(user1), 1000 * 1e18);
    }

    function test_DetectBrokenInterface_BeforeUpgrade() public {
        BrokenAuthorizer brokenAuth = new BrokenAuthorizer();

        // Should revert during proposal due to interface check
        vm.expectRevert("QONE: invalid authorizer interface");
        token.proposePQCAuthorizer(address(brokenAuth));
    }

    function test_TransferLock_IfBadAuthorizerDeployed() public {
        // This test demonstrates the FP-1 issue if timelock didn't exist
        // We'll use a restrictive authorizer

        RestrictiveAuthorizer restrictive = new RestrictiveAuthorizer(owner,
user1);

        // Give user1 some tokens first
        token.transfer(user1, 10000 * 1e18);

        // Propose and finalize the restrictive authorizer
        token.proposePQCAuthorizer(address(restrictive));
        vm.warp(block.timestamp + token.TIMELOCK_DURATION());
        token.finalizePQCAuthorizer("");

        // Now user1 cannot transfer (they are blocked)
        vm.prank(user1);
        vm.expectRevert("QONE: sender not authorized (PQC)");
        token.transfer(user2, 1000 * 1e18);
```

```
        // But owner can still transfer
        token.transfer(user2, 1000 * 1e18);
        assertEq(token.balanceOf(user2), 1000 * 1e18);
    }

    function test_OwnerMismatch_PreventsFutureUpgrades() public {
        // FP-1: If new authorizer's owner doesn't match token owner,
        // future upgrades will fail

        address differentOwner = address(0x999);
        DummyAuthorizer mismatchedAuthorizer = new
DummyAuthorizer(differentOwner);

        // First upgrade succeeds (current authorizer checks token owner)
        token.proposePQCAuthorizer(address(mismatchedAuthorizer));
        vm.warp(block.timestamp + token.TIMELOCK_DURATION());
        token.finalizePQCAuthorizer("");

        assertEq(address(token.pqcAuthorizer()),
address(mismatchedAuthorizer));

        // Try to upgrade again - should fail because new authorizer
        // checks differentOwner, not current token owner
        DummyAuthorizer anotherAuthorizer = new DummyAuthorizer(owner);
        token.proposePQCAuthorizer(address(anotherAuthorizer));
        vm.warp(block.timestamp + token.TIMELOCK_DURATION());

        vm.expectRevert("QONE: PQC upgrade not authorized");
        token.finalizePQCAuthorizer("");
    }

    function test_MultipleProbeChecks_DifferentUsers() public {
        // Test that probe works for multiple users
        RestrictiveAuthorizer restrictive = new RestrictiveAuthorizer(owner,
user1);
        token.proposePQCAuthorizer(address(restrictive));

        // Probe for user1 (blocked)
        (QONE.ProbeStatus status1,) = token.probePendingAuthorizer(user1);
        assertEq(uint8(status1), uint8(QONE.ProbeStatus.NOT_AUTHORIZED));

        // Probe for user2 (not blocked)
        (QONE.ProbeStatus status2,) = token.probePendingAuthorizer(user2);
        assertEq(uint8(status2), uint8(QONE.ProbeStatus.OK));

        // Probe for owner (not blocked)
        (QONE.ProbeStatus status3,) = token.probePendingAuthorizer(owner);
        assertEq(uint8(status3), uint8(QONE.ProbeStatus.OK));
    }

    /*//////////////////////////////////////////////////////////////
                    FUZZING TESTS
    //////////////////////////////////////////////////////////////*/

    function testFuzz_Transfer(address to, uint256 amount) public {
        vm.assume(to != address(0));
        vm.assume(to != owner);
        amount = bound(amount, 0, token.TOTAL_SUPPLY());

        token.transfer(to, amount);
        assertEq(token.balanceOf(to), amount);
        assertEq(token.balanceOf(owner), token.TOTAL_SUPPLY() - amount);
    }
```

```
    function testFuzz_TimelockDuration(uint256 warpTime) public {
        DummyAuthorizer newAuthorizer = new DummyAuthorizer(owner);
        token.proposePQCAuthorizer(address(newAuthorizer));

        // Bound warp time to reasonable range
        warpTime = bound(warpTime, 0, 365 days);

        vm.warp(block.timestamp + warpTime);

        if (warpTime >= token.TIMELOCK_DURATION()) {
            // Should succeed
            token.finalizePQCAuthorizer("");
            assertEq(address(token.pqcAuthorizer()), address(newAuthorizer));
        } else {
            // Should fail
            vm.expectRevert("QONE: timelock not expired");
            token.finalizePQCAuthorizer("");
        }
    }

    /*//////////////////////////////////////////////////////////////
                    EDGE CASES
    //////////////////////////////////////////////////////////////*/

    function test_CancelAndRepropose() public {
        DummyAuthorizer auth1 = new DummyAuthorizer(owner);
        DummyAuthorizer auth2 = new DummyAuthorizer(owner);

        // Propose first
        token.proposePQCAuthorizer(address(auth1));
        assertEq(token.pendingPQCAuthorizer(), address(auth1));

        // Cancel
        token.cancelPendingUpdate();
        assertEq(token.pendingPQCAuthorizer(), address(0));

        // Propose second
        token.proposePQCAuthorizer(address(auth2));
        assertEq(token.pendingPQCAuthorizer(), address(auth2));

        // Finalize second
        vm.warp(block.timestamp + token.TIMELOCK_DURATION());
        token.finalizePQCAuthorizer("");
        assertEq(address(token.pqcAuthorizer()), address(auth2));
    }

    function test_ExactTimelockBoundary() public {
        DummyAuthorizer newAuthorizer = new DummyAuthorizer(owner);
        token.proposePQCAuthorizer(address(newAuthorizer));

        uint256 unlockTime = token.upgradeTimelock();

        // One second before unlock - should fail
        vm.warp(unlockTime - 1);
        vm.expectRevert("QONE: timelock not expired");
        token.finalizePQCAuthorizer("");

        // Exactly at unlock time - should succeed
        vm.warp(unlockTime);
        token.finalizePQCAuthorizer("");
        assertEq(address(token.pqcAuthorizer()), address(newAuthorizer));
    }
```

```
    function test_SequentialUpgrades() public {
        // First upgrade
        DummyAuthorizer auth1 = new DummyAuthorizer(owner);
        token.proposePQCAuthorizer(address(auth1));
        vm.warp(block.timestamp + token.TIMELOCK_DURATION());
        token.finalizePQCAuthorizer("");
        assertEq(address(token.pqcAuthorizer()), address(auth1));

        // Second upgrade
        DummyAuthorizer auth2 = new DummyAuthorizer(owner);
        token.proposePQCAuthorizer(address(auth2));
        vm.warp(block.timestamp + token.TIMELOCK_DURATION());
        token.finalizePQCAuthorizer("");
        assertEq(address(token.pqcAuthorizer()), address(auth2));

        // Third upgrade
        DummyAuthorizer auth3 = new DummyAuthorizer(owner);
        token.proposePQCAuthorizer(address(auth3));
        vm.warp(block.timestamp + token.TIMELOCK_DURATION());
        token.finalizePQCAuthorizer("");
        assertEq(address(token.pqcAuthorizer()), address(auth3));
    }
}

/**
 * @title BrokenAuthorizer
 * @notice Mock authorizer with broken interface (always reverts)
 */
contract BrokenAuthorizer {
    function isAuthorized(address) external pure returns (bool) {
        revert("Broken interface");
    }
}

/**
 * @title AlwaysFalseAuthorizer
 * @notice Mock authorizer that always returns false (misconfigured)
 */
contract AlwaysFalseAuthorizer is IPQCAuthorizer {
    address public immutable owner;

    constructor(address _owner) {
        owner = _owner;
    }

    function isAuthorized(address) external pure override returns (bool) {
        return false; // Always denies authorization
    }

    function authorizeUpgrade(
        address sender,
        address tokenContract,
        address currentAuthorizer,
        address newAuthorizer,
        bytes calldata proof
    ) external override returns (bool) {
        tokenContract;
        currentAuthorizer;
        newAuthorizer;
        proof;

        bool isApproved = (sender == owner);
```

```
        if (isApproved) {
            emit UpgradeAuthorized(tokenContract, currentAuthorizer,
newAuthorizer);
        }
        return isApproved;
    }
}

/**
 * @title RestrictiveAuthorizer
 * @notice Mock authorizer that blocks a specific address
 */
contract RestrictiveAuthorizer is IPQCAuthorizer {
    address public immutable owner;
    address public immutable blockedAddress;

    constructor(address _owner, address _blocked) {
        owner = _owner;
        blockedAddress = _blocked;
    }

    function isAuthorized(address account) external view override returns
(bool) {
        return account != blockedAddress;
    }

    function authorizeUpgrade(
        address sender,
        address tokenContract,
        address currentAuthorizer,
        address newAuthorizer,
        bytes calldata proof
    ) external override returns (bool) {
        tokenContract;
        currentAuthorizer;
        newAuthorizer;
        proof;

        bool isApproved = (sender == owner);
        if (isApproved) {
            emit UpgradeAuthorized(tokenContract, currentAuthorizer,
newAuthorizer);
        }
        return isApproved;
    }
}
```

2. TokenVestingV2.t.sol

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.20;

import "forge-std/Test.sol";
import "../src/TokenVestingV2.sol";
import "../src/QONE-V2.sol";
import "../src/DummyAuthorizer.sol";

/**
 * @title TokenVestingV2Test
 * @notice Comprehensive test suite for TokenVestingV2 with 6 categories
```

```
 * @dev Tests cover all vesting schedules
 */
contract TokenVestingV2Test is Test {
    TokenVestingV2 public vesting;
    QONE public token;
    DummyAuthorizer public authorizer;

    address public owner = address(1);
    address public beneficiary1 = address(2);
    address public beneficiary2 = address(3);
    address public beneficiary3 = address(4);
    address public beneficiary4 = address(5);
    address public beneficiary5 = address(6);
    address public beneficiary6 = address(7);

    uint256 public tgeTimestamp;

    // Per-category merkle roots (now 6 categories)
    bytes32[6] public merkleRoots;

    // Allocation amounts per tokenomics
    uint256 constant TIER1_AMOUNT = 50_000_000 * 1e18;        // Token Sale
Tier 1
    uint256 constant TIER2_AMOUNT = 100_000_000 * 1e18;       // Token Sale
Tier 2
    uint256 constant TIER3_AMOUNT = 50_000_000 * 1e18;        // Token Sale
Tier 3
    uint256 constant TEAM_AMOUNT = 150_000_000 * 1e18;        // Team &
Advisors
    uint256 constant TREASURY_AMOUNT = 110_000_000 * 1e18;    //
Liquidity/Treasury
    uint256 constant AIRDROP_AMOUNT = 330_000_000 * 1e18;     // Community
Airdrops

    // Event declarations for testing
    event MerkleRootUpdated(uint8 indexed category, bytes32 indexed newRoot);
    event AllocationInitialized(address indexed beneficiary, uint8 indexed
category, uint256 amount);
    event TokensClaimed(address indexed beneficiary, uint8 indexed category,
uint256 amount);
    event EmergencyWithdraw(address indexed owner, uint256 amount);
    event ExcessWithdrawn(address indexed owner, uint256 amount);

    function setUp() public {
        vm.startPrank(owner);

        // Deploy contracts
        authorizer = new DummyAuthorizer(owner);
        token = new QONE(owner, address(authorizer));

        // Set TGE to 1 day from now
        tgeTimestamp = block.timestamp + 1 days;

        // Deploy vesting contract
        vesting = new TokenVestingV2(
            address(token),
            tgeTimestamp,
            owner
        );

        // Create per-category merkle roots (6 categories now)
        merkleRoots[0] = keccak256(abi.encode(beneficiary1, TIER1_AMOUNT,
uint8(0)));
```

```
        merkleRoots[1] = keccak256(abi.encode(beneficiary2, TIER2_AMOUNT,
uint8(1)));
        merkleRoots[2] = keccak256(abi.encode(beneficiary3, TIER3_AMOUNT,
uint8(2)));
        merkleRoots[3] = keccak256(abi.encode(beneficiary4, TEAM_AMOUNT,
uint8(3)));
        merkleRoots[4] = keccak256(abi.encode(beneficiary5, TREASURY_AMOUNT,
uint8(4)));
        merkleRoots[5] = keccak256(abi.encode(beneficiary6, AIRDROP_AMOUNT,
uint8(5)));

        // Set merkle roots for each category
        for (uint8 i = 0; i < 6; i++) {
            vesting.setMerkleRoot(i, merkleRoots[i]);
        }

        // Transfer tokens to vesting contract
        uint256 totalVestingAmount = TIER1_AMOUNT + TIER2_AMOUNT +
TIER3_AMOUNT +
                                     TEAM_AMOUNT  + TREASURY_AMOUNT +
AIRDROP_AMOUNT;
        token.transfer(address(vesting), totalVestingAmount);

        vm.stopPrank();
    }

    // Helper to get empty proof for single-leaf merkle tree
    function _getProof() private pure returns (bytes32[] memory) {
        return new bytes32[](0);
    }

    /*//////////////////////////////////////////////////////////////
                        INITIAL STATE TESTS
    //////////////////////////////////////////////////////////////*/

    function testInitialState() public view {
        assertEq(address(vesting.token()), address(token));
        assertEq(vesting.tgeTimestamp(), tgeTimestamp);
        assertEq(vesting.totalAllocated(), 0);
        assertEq(vesting.totalClaimed(), 0);

        // Verify merkle roots for each category
        for (uint8 i = 0; i < 6; i++) {
            assertEq(vesting.merkleRoots(i), merkleRoots[i]);
        }
    }

    /*//////////////////////////////////////////////////////////////
                    VESTING SCHEDULE VERIFICATION TESTS
    //////////////////////////////////////////////////////////////*/

    function testVestingSchedules() public view {
        // Test Token Sale Tier 1: 15% TGE, 3mo cliff, 9mo vest
        TokenVestingV2.VestingSchedule memory schedule0 =
vesting.getVestingSchedule(0);
        assertEq(schedule0.tgePercent, 1500);
        assertEq(schedule0.cliffDuration, 90 days);
        assertEq(schedule0.vestingDuration, 270 days);

        // Test Token Sale Tier 2: 15% TGE, 0mo cliff, 12mo vest
        TokenVestingV2.VestingSchedule memory schedule1 =
vesting.getVestingSchedule(1);
        assertEq(schedule1.tgePercent, 1500);
```

```
        assertEq(schedule1.cliffDuration, 0);
        assertEq(schedule1.vestingDuration, 365 days);

        // Test Token Sale Tier 3: 100% TGE
        TokenVestingV2.VestingSchedule memory schedule2 =
vesting.getVestingSchedule(2);
        assertEq(schedule2.tgePercent, 10000);
        assertEq(schedule2.cliffDuration, 0);
        assertEq(schedule2.vestingDuration, 0);

        // Test Team & Advisors: 20% TGE, 6mo cliff, 24mo vest
        TokenVestingV2.VestingSchedule memory schedule3 =
vesting.getVestingSchedule(3);
        assertEq(schedule3.tgePercent, 2000); // Changed from 1500 to 2000
(20%)
        assertEq(schedule3.cliffDuration, 180 days);
        assertEq(schedule3.vestingDuration, 730 days);

        // Test Liquidity/Treasury: 50% TGE, 0mo cliff, 36mo vest
        TokenVestingV2.VestingSchedule memory schedule4 =
vesting.getVestingSchedule(4);
        assertEq(schedule4.tgePercent, 5000); // Changed from 3500 to 5000
(50%)
        assertEq(schedule4.cliffDuration, 0);
        assertEq(schedule4.vestingDuration, 1095 days);

        // Test Community Airdrops: 9% TGE, 0mo cliff, 24mo vest
        TokenVestingV2.VestingSchedule memory schedule5 =
vesting.getVestingSchedule(5);
        assertEq(schedule5.tgePercent, 900); // Changed from 500 to 900 (9%)
        assertEq(schedule5.cliffDuration, 0);
        assertEq(schedule5.vestingDuration, 730 days);
    }

    /*//////////////////////////////////////////////////////////////
                    ALLOCATION INITIALIZATION TESTS
    //////////////////////////////////////////////////////////////*/

    function testInitializeAllocation() public {
        bytes32[] memory proof = _getProof();

        vm.prank(beneficiary1);
        vesting.initializeMyAllocation(TIER1_AMOUNT, 0, proof);

        TokenVestingV2.Allocation memory allocation =
vesting.getAllocation(beneficiary1, 0);
        assertEq(allocation.totalAmount, TIER1_AMOUNT);
        assertEq(allocation.claimedAmount, 0);
        assertTrue(allocation.initialized);
        assertEq(vesting.totalAllocated(), TIER1_AMOUNT);
    }

    function testInitializeAllocationInvalidProof() public {
        bytes32[] memory invalidProof = new bytes32[](1);
        invalidProof[0] = bytes32(uint256(123));

        vm.prank(beneficiary1);
        vm.expectRevert(TokenVestingV2.InvalidProof.selector);
        vesting.initializeMyAllocation(TIER1_AMOUNT, 0, invalidProof);
    }

    function testInitializeAllocationTwice() public {
        bytes32[] memory proof = _getProof();
```

```
        vm.startPrank(beneficiary1);
        vesting.initializeMyAllocation(TIER1_AMOUNT, 0, proof);

        vm.expectRevert(TokenVestingV2.AlreadyInitialized.selector);
        vesting.initializeMyAllocation(TIER1_AMOUNT, 0, proof);
        vm.stopPrank();
    }

    /*//////////////////////////////////////////////////////////////
                    CLAIMING TESTS - BASIC
    //////////////////////////////////////////////////////////////*/

    function testClaimBeforeTGE() public {
        bytes32[] memory proof = _getProof();

        vm.startPrank(beneficiary1);
        vesting.initializeMyAllocation(TIER1_AMOUNT, 0, proof);

        vm.expectRevert(TokenVestingV2.NothingToClaim.selector);
        vesting.claim(0);
        vm.stopPrank();
    }

    function testClaimTGEUnlock() public {
        bytes32[] memory proof = _getProof();

        vm.startPrank(beneficiary1);
        vesting.initializeMyAllocation(TIER1_AMOUNT, 0, proof);

        // Fast forward to TGE
        vm.warp(tgeTimestamp);

        // Claim TGE unlock (15%)
        uint256 expectedTGE = (TIER1_AMOUNT * 1500) / 10000;

        uint256 claimable = vesting.getClaimableAmount(beneficiary1, 0);
        assertEq(claimable, expectedTGE);

        vesting.claim(0);
        assertEq(token.balanceOf(beneficiary1), expectedTGE);
        assertEq(vesting.getAllocation(beneficiary1, 0).claimedAmount,
expectedTGE);
        vm.stopPrank();
    }

    function testClaimDuringCliff() public {
        bytes32[] memory proof = _getProof();

        vm.startPrank(beneficiary1);
        vesting.initializeMyAllocation(TIER1_AMOUNT, 0, proof);

        // Fast forward to TGE + 45 days (middle of cliff)
        vm.warp(tgeTimestamp + 45 days);

        // Should only be able to claim TGE amount
        uint256 expectedTGE = (TIER1_AMOUNT * 1500) / 10000;

        assertEq(vesting.getClaimableAmount(beneficiary1, 0), expectedTGE);
        vesting.claim(0);
        assertEq(token.balanceOf(beneficiary1), expectedTGE);
        vm.stopPrank();
    }
```

```
    function testClaimFullyVested() public {
        // Test Tier 3: 100% TGE
        bytes32[] memory proof = _getProof();

        vm.startPrank(beneficiary3);
        vesting.initializeMyAllocation(TIER3_AMOUNT, 2, proof);

        vm.warp(tgeTimestamp);

        assertEq(vesting.getClaimableAmount(beneficiary3, 2), TIER3_AMOUNT);

        vesting.claim(2);
        assertEq(token.balanceOf(beneficiary3), TIER3_AMOUNT);
        assertEq(vesting.getAllocation(beneficiary3, 2).claimedAmount,
TIER3_AMOUNT);
        vm.stopPrank();
    }

    /*////////////////////////////////////////////////////////////
                UPDATED SCHEDULES TESTS (V2 CHANGES)
    ////////////////////////////////////////////////////////////*/

    function testTeamVestingSchedule_Updated() public {
        // Updated: 20% TGE (was 15%), 6mo cliff, 24mo vest
        bytes32[] memory proof = _getProof();

        vm.prank(beneficiary4);
        vesting.initializeMyAllocation(TEAM_AMOUNT, 3, proof);

        // At TGE, 20% claimable (changed from 15%)
        vm.warp(tgeTimestamp);
        uint256 tgeAmount = (TEAM_AMOUNT * 2000) / 10000; // 20%
        assertEq(vesting.getClaimableAmount(beneficiary4, 3), tgeAmount);

        // During cliff (e.g. 3 months), still only TGE amount
        vm.warp(tgeTimestamp + 90 days);
        assertEq(vesting.getClaimableAmount(beneficiary4, 3), tgeAmount);

        // At cliff end (6 months)
        vm.warp(tgeTimestamp + 180 days);
        assertEq(vesting.getClaimableAmount(beneficiary4, 3), tgeAmount);

        // Halfway through vesting (6 months cliff + 12 months)
        vm.warp(tgeTimestamp + 180 days + 365 days);
        uint256 vestingAmount = TEAM_AMOUNT - tgeAmount;
        uint256 halfVested = vestingAmount / 2;
        assertApproxEqRel(vesting.getClaimableAmount(beneficiary4, 3),
tgeAmount + halfVested, 0.01e18);

        // Fully vested (6 months cliff + 24 months vesting)
        vm.warp(tgeTimestamp + 180 days + 730 days + 1);
        assertEq(vesting.getClaimableAmount(beneficiary4, 3), TEAM_AMOUNT);
    }

    function testTreasuryVestingSchedule_Updated() public {
        // Updated: 50% TGE (was 35%), 0 cliff, 36mo vest
        bytes32[] memory proof = _getProof();

        vm.startPrank(beneficiary5);
        vesting.initializeMyAllocation(TREASURY_AMOUNT, 4, proof);

        // At TGE, 50% claimable (changed from 35%)
```

```
        vm.warp(tgeTimestamp);
        uint256 tgeAmount = (TREASURY_AMOUNT * 5000) / 10000; // 50%
        assertEq(vesting.getClaimableAmount(beneficiary5, 4), tgeAmount);
        vesting.claim(4);
        assertEq(token.balanceOf(beneficiary5), tgeAmount);

        // After 18 months (50% of vesting)
        vm.warp(tgeTimestamp + 547 days);
        uint256 vestingAmount = TREASURY_AMOUNT - tgeAmount;
        uint256 halfVested = vestingAmount / 2;

        uint256 additionalClaimable = vesting.getClaimableAmount(beneficiary5,
4);
        assertApproxEqRel(additionalClaimable, halfVested, 0.01e18);

        vm.stopPrank();
    }

    function testAirdropVestingSchedule_Updated() public {
        // Updated: 9% TGE (was 5%), 0 cliff, 24mo vest
        bytes32[] memory proof = _getProof();

        vm.prank(beneficiary6);
        vesting.initializeMyAllocation(AIRDROP_AMOUNT, 5, proof);

        // At TGE, 9% claimable (changed from 5%)
        vm.warp(tgeTimestamp);
        uint256 tgeAmount = (AIRDROP_AMOUNT * 900) / 10000; // 9%
        assertEq(vesting.getClaimableAmount(beneficiary6, 5), tgeAmount);

        // At 12 months (50% of 24mo vesting)
        vm.warp(tgeTimestamp + 365 days);
        uint256 vestingAmount = AIRDROP_AMOUNT - tgeAmount;
        uint256 halfVested = tgeAmount + (vestingAmount * 365) / 730;
        assertApproxEqRel(vesting.getClaimableAmount(beneficiary6, 5),
halfVested, 0.01e18);

        // Fully vested at 24 months
        vm.warp(tgeTimestamp + 730 days + 1);
        assertEq(vesting.getClaimableAmount(beneficiary6, 5), AIRDROP_AMOUNT);
    }

    /*//////////////////////////////////////////////////////////////
                    MULTIPLE CLAIMS TESTS
    //////////////////////////////////////////////////////////////*/

    function testMultipleClaimsOverTime() public {
        bytes32[] memory proof = _getProof();

        vm.startPrank(beneficiary2);
        vesting.initializeMyAllocation(TIER2_AMOUNT, 1, proof);

        // Claim at TGE
        vm.warp(tgeTimestamp);
        vesting.claim(1);
        uint256 balance1 = token.balanceOf(beneficiary2);

        // Claim at 3 months
        vm.warp(tgeTimestamp + 90 days);
        vesting.claim(1);
        uint256 balance2 = token.balanceOf(beneficiary2);
        assertGt(balance2, balance1);
```

```
        // Claim at 6 months
        vm.warp(tgeTimestamp + 180 days);
        vesting.claim(1);
        uint256 balance3 = token.balanceOf(beneficiary2);
        assertGt(balance3, balance2);

        // Claim at 12 months (fully vested)
        vm.warp(tgeTimestamp + 365 days + 1);
        vesting.claim(1);
        uint256 balance4 = token.balanceOf(beneficiary2);
        assertEq(balance4, TIER2_AMOUNT);

        // Try to claim again - should fail
        vm.expectRevert(TokenVestingV2.NothingToClaim.selector);
        vesting.claim(1);

        vm.stopPrank();
    }

    /*//////////////////////////////////////////////////////////////
                    ADMIN FUNCTIONS TESTS
    //////////////////////////////////////////////////////////////*/

    function testSetMerkleRoot() public {
        bytes32 newRoot = bytes32(uint256(123));

        vm.prank(owner);
        vesting.setMerkleRoot(0, newRoot);

        assertEq(vesting.merkleRoots(0), newRoot);
    }

    function testSetMerkleRootNotOwner() public {
        bytes32 newRoot = bytes32(uint256(123));

        vm.prank(beneficiary1);
        vm.expectRevert();
        vesting.setMerkleRoot(0, newRoot);
    }

    function testSetMerkleRootZero() public {
        vm.prank(owner);
        vm.expectRevert(TokenVestingV2.InvalidMerkleRoot.selector);
        vesting.setMerkleRoot(0, bytes32(0));
    }

    function testEmergencyWithdraw() public {
        uint256 withdrawAmount = 1000 * 1e18;

        uint256 balance_before = token.balanceOf(owner);

        vm.prank(owner);
        vesting.emergencyWithdraw(withdrawAmount);

        assertEq(token.balanceOf(owner) - balance_before, withdrawAmount);
    }

    function testEmergencyWithdrawAfterTGE() public {
        vm.warp(tgeTimestamp + 1);

        vm.prank(owner);
        vm.expectRevert(TokenVestingV2.TGEStarted.selector);
        vesting.emergencyWithdraw(1000 * 1e18);
```

```
    }

    function testWithdrawExcess() public {
        // Initialize only SOME allocations
        vm.prank(beneficiary1);
        vesting.initializeMyAllocation(TIER1_AMOUNT, 0, _getProof());
        vm.prank(beneficiary2);
        vesting.initializeMyAllocation(TIER2_AMOUNT, 1, _getProof());
        vm.prank(beneficiary3);
        vesting.initializeMyAllocation(TIER3_AMOUNT, 2, _getProof());

        uint256 ownerBalanceBefore = token.balanceOf(owner);
        uint256 totalInitialized = TIER1_AMOUNT + TIER2_AMOUNT + TIER3_AMOUNT;
        uint256 totalInContract = TIER1_AMOUNT + TIER2_AMOUNT + TIER3_AMOUNT +
                                  TEAM_AMOUNT + TREASURY_AMOUNT +
AIRDROP_AMOUNT;
        uint256 expectedExcess = totalInContract - totalInitialized;

        vm.prank(owner);
        vesting.withdrawExcess();

        assertEq(token.balanceOf(owner), ownerBalanceBefore + expectedExcess);
    }

    function testWithdrawExcessNoExcess() public {
        // Initialize all allocations
        vm.prank(beneficiary1);
        vesting.initializeMyAllocation(TIER1_AMOUNT, 0, _getProof());
        vm.prank(beneficiary2);
        vesting.initializeMyAllocation(TIER2_AMOUNT, 1, _getProof());
        vm.prank(beneficiary3);
        vesting.initializeMyAllocation(TIER3_AMOUNT, 2, _getProof());
        vm.prank(beneficiary4);
        vesting.initializeMyAllocation(TEAM_AMOUNT, 3, _getProof());
        vm.prank(beneficiary5);
        vesting.initializeMyAllocation(TREASURY_AMOUNT, 4, _getProof());
        vm.prank(beneficiary6);
        vesting.initializeMyAllocation(AIRDROP_AMOUNT, 5, _getProof());

        vm.prank(owner);
        vm.expectRevert(TokenVestingV2.NoExcessTokens.selector);
        vesting.withdrawExcess();
    }

    /*//////////////////////////////////////////////////////////////
                    VALIDATION TESTS
    //////////////////////////////////////////////////////////////*/

    function testInvalidCategory() public {
        bytes32[] memory proof = _getProof();

        vm.prank(beneficiary1);
        vm.expectRevert(TokenVestingV2.InvalidCategory.selector);
        vesting.initializeMyAllocation(TIER1_AMOUNT, 7, proof); // 7 is
invalid, max is 5
    }

    function testInvalidCategoryGetSchedule() public {
        vm.expectRevert(TokenVestingV2.InvalidCategory.selector);
        vesting.getVestingSchedule(7);
    }

    function testInvalidCategoryClaim() public {
```

```
        vm.prank(beneficiary1);
        vm.expectRevert(TokenVestingV2.InvalidCategory.selector);
        vesting.claim(7);
    }

    function testInitializeAllocationZeroAmount() public {
        bytes32[] memory proof = _getProof();

        vm.prank(beneficiary1);
        vm.expectRevert(TokenVestingV2.InvalidAmount.selector);
        vesting.initializeMyAllocation(0, 0, proof);
    }

    /*//////////////////////////////////////////////////////////////
                    EVENT TESTS
    //////////////////////////////////////////////////////////////*/

    function testEmergencyWithdrawEvent() public {
        uint256 withdrawAmount = 1000 * 1e18;

        vm.prank(owner);
        vm.expectEmit(true, false, false, true);
        emit EmergencyWithdraw(owner, withdrawAmount);
        vesting.emergencyWithdraw(withdrawAmount);
    }

    function testWithdrawExcessEvent() public {
        vm.prank(beneficiary1);
        vesting.initializeMyAllocation(TIER1_AMOUNT, 0, _getProof());

        uint256 totalInContract = TIER1_AMOUNT + TIER2_AMOUNT + TIER3_AMOUNT +
                                  TEAM_AMOUNT + TREASURY_AMOUNT +
AIRDROP_AMOUNT;
        uint256 expectedExcess = totalInContract - TIER1_AMOUNT;

        vm.prank(owner);
        vm.expectEmit(true, false, false, true);
        emit ExcessWithdrawn(owner, expectedExcess);
        vesting.withdrawExcess();
    }

    function testMerkleRootUpdatedEvent() public {
        bytes32 newRoot = bytes32(uint256(123));

        vm.prank(owner);
        vm.expectEmit(true, true, false, false);
        emit MerkleRootUpdated(0, newRoot);
        vesting.setMerkleRoot(0, newRoot);
    }

    function testAllocationInitializedEvent() public {
        bytes32[] memory proof = _getProof();

        vm.prank(beneficiary1);
        vm.expectEmit(true, true, false, true);
        emit AllocationInitialized(beneficiary1, 0, TIER1_AMOUNT);
        vesting.initializeMyAllocation(TIER1_AMOUNT, 0, proof);
    }

    function testTokensClaimedEvent() public {
        bytes32[] memory proof = _getProof();

        vm.prank(beneficiary1);
```

```
        vesting.initializeMyAllocation(TIER1_AMOUNT, 0, proof);

        vm.warp(tgeTimestamp);
        uint256 expectedTGE = (TIER1_AMOUNT * 1500) / 10000;

        vm.prank(beneficiary1);
        vm.expectEmit(true, true, false, true);
        emit TokensClaimed(beneficiary1, 0, expectedTGE);
        vesting.claim(0);
    }

    /*////////////////////////////////////////////////////////////
                COMPREHENSIVE VESTING PROGRESSION TESTS
    ////////////////////////////////////////////////////////////*/

    function testGetVestedAmountProgression() public {
        // Test Tier 2: 15% TGE, 12 month linear vesting
        bytes32[] memory proof = _getProof();

        vm.prank(beneficiary2);
        vesting.initializeMyAllocation(TIER2_AMOUNT, 1, proof);

        uint256 tgeAmount = (TIER2_AMOUNT * 1500) / 10000;
        uint256 vestingAmount = TIER2_AMOUNT - tgeAmount;

        // Before TGE
        assertEq(vesting.getVestedAmount(beneficiary2, 1), 0);

        // At TGE
        vm.warp(tgeTimestamp);
        assertEq(vesting.getVestedAmount(beneficiary2, 1), tgeAmount);

        // At 25% of vesting period (3 months)
        vm.warp(tgeTimestamp + 91 days);
        uint256 expectedVested25 = tgeAmount + (vestingAmount * 91) / 365;
        assertApproxEqRel(vesting.getVestedAmount(beneficiary2, 1),
expectedVested25, 0.01e18);

        // At 50% of vesting period (6 months)
        vm.warp(tgeTimestamp + 182 days);
        uint256 expectedVested50 = tgeAmount + (vestingAmount * 182) / 365;
        assertApproxEqRel(vesting.getVestedAmount(beneficiary2, 1),
expectedVested50, 0.01e18);

        // Fully vested
        vm.warp(tgeTimestamp + 365 days + 1);
        assertEq(vesting.getVestedAmount(beneficiary2, 1), TIER2_AMOUNT);
    }

    /*////////////////////////////////////////////////////////////
                ALL CATEGORIES INTEGRATION TEST
    ////////////////////////////////////////////////////////////*/

    function testAllCategoriesFullCycle() public {
        bytes32[] memory proof = _getProof();

        // Initialize all categories
        vm.prank(beneficiary1);
        vesting.initializeMyAllocation(TIER1_AMOUNT, 0, proof);
        vm.prank(beneficiary2);
        vesting.initializeMyAllocation(TIER2_AMOUNT, 1, proof);
        vm.prank(beneficiary3);
        vesting.initializeMyAllocation(TIER3_AMOUNT, 2, proof);
```

```
        vm.prank(beneficiary4);
        vesting.initializeMyAllocation(TEAM_AMOUNT, 3, proof);
        vm.prank(beneficiary5);
        vesting.initializeMyAllocation(TREASURY_AMOUNT, 4, proof);
        vm.prank(beneficiary6);
        vesting.initializeMyAllocation(AIRDROP_AMOUNT, 5, proof);

        uint256 totalExpected = TIER1_AMOUNT + TIER2_AMOUNT + TIER3_AMOUNT +
                                TEAM_AMOUNT + TREASURY_AMOUNT +
AIRDROP_AMOUNT;
        assertEq(vesting.totalAllocated(), totalExpected);

        // Warp to TGE and claim all TGE unlocks
        vm.warp(tgeTimestamp);

        vm.prank(beneficiary1);
        vesting.claim(0);
        vm.prank(beneficiary2);
        vesting.claim(1);
        vm.prank(beneficiary3);
        vesting.claim(2);
        vm.prank(beneficiary4);
        vesting.claim(3);
        vm.prank(beneficiary5);
        vesting.claim(4);
        vm.prank(beneficiary6);
        vesting.claim(5);

        // Verify TGE amounts
        assertEq(token.balanceOf(beneficiary1), (TIER1_AMOUNT * 1500) /
10000); // 15%
        assertEq(token.balanceOf(beneficiary2), (TIER2_AMOUNT * 1500) /
10000); // 15%
        assertEq(token.balanceOf(beneficiary3), TIER3_AMOUNT); // 100%
        assertEq(token.balanceOf(beneficiary4), (TEAM_AMOUNT * 2000) / 10000);
// 20%
        assertEq(token.balanceOf(beneficiary5), (TREASURY_AMOUNT * 5000) /
10000); // 50%
        assertEq(token.balanceOf(beneficiary6), (AIRDROP_AMOUNT * 900) /
10000); // 9%
    }
}
```

3. UnitTestOutput

```
Ran 31 tests for test/TokenVestingV2.t.sol:TokenVestingV2Test
[PASS] testAirdropVestingSchedule_Updated() (gas: 115644)
[PASS] testAllCategoriesFullCycle() (gas: 809096)
[PASS] testAllocationInitializedEvent() (gas: 89280)
[PASS] testClaimBeforeTGE() (gas: 94789)
[PASS] testClaimDuringCliff() (gas: 196324)
[PASS] testClaimFullyVested() (gas: 200491)
[PASS] testClaimTGEUnlock() (gas: 200033)
[PASS] testEmergencyWithdraw() (gas: 44950)
[PASS] testEmergencyWithdrawAfterTGE() (gas: 16716)
[PASS] testEmergencyWithdrawEvent() (gas: 41525)
[PASS] testGetVestedAmountProgression() (gas: 124095)
[PASS] testInitialState() (gas: 53538)
```

```
[PASS] testInitializeAllocation() (gas: 91606)
[PASS] testInitializeAllocationInvalidProof() (gas: 18828)
[PASS] testInitializeAllocationTwice() (gas: 89733)
[PASS] testInitializeAllocationZeroAmount() (gas: 12837)
[PASS] testInvalidCategory() (gas: 12714)
[PASS] testInvalidCategoryClaim() (gas: 16732)
[PASS] testInvalidCategoryGetSchedule() (gas: 10867)
[PASS] testMerkleRootUpdatedEvent() (gas: 22767)
[PASS] testMultipleClaimsOverTime() (gas: 254225)
[PASS] testSetMerkleRoot() (gas: 22246)
[PASS] testSetMerkleRootNotOwner() (gas: 14332)
[PASS] testSetMerkleRootZero() (gas: 14298)
[PASS] testTeamVestingSchedule_Updated() (gas: 126631)
[PASS] testTokensClaimedEvent() (gas: 190760)
[PASS] testTreasuryVestingSchedule_Updated() (gas: 206533)
[PASS] testVestingSchedules() (gas: 61113)
[PASS] testWithdrawExcess() (gas: 244171)
[PASS] testWithdrawExcessEvent() (gas: 125814)
[PASS] testWithdrawExcessNoExcess() (gas: 387478)
Suite result: ok. 31 passed; 0 failed; 0 skipped; finished in 1.62ms (2.92ms
CPU time)

Ran 29 tests for test/QONE.t.sol:QONETest
[PASS] testFuzz_TimelockDuration(uint256) (runs: 256, μ: 286763, ~: 276997)
[PASS] testFuzz_Transfer(address,uint256) (runs: 256, μ: 58552, ~: 58182)
[PASS] test_CancelAndRepropose() (gas: 514913)
[PASS] test_CancelPendingUpdate() (gas: 260588)
[PASS] test_CompleteUpgradeFlow_DetectMisconfiguration() (gas: 554135)
[PASS] test_Deployment() (gas: 46118)
[PASS] test_DetectBrokenInterface_BeforeUpgrade() (gas: 150335)
[PASS] test_ExactTimelockBoundary() (gas: 274860)
[PASS] test_FinalizePQCAuthorizer() (gas: 276063)
[PASS] test_MultipleProbeChecks_DifferentUsers() (gas: 336663)
[PASS] test_MultipleTransfers() (gas: 118160)
[PASS] test_OwnerMismatch_PreventsFutureUpgrades() (gas: 555494)
[PASS] test_ProbeFunction_AlwaysFalseAuthorizer() (gas: 294944)
[PASS] test_ProbeFunction_BrokenAuthorizer() (gas: 150358)
[PASS] test_ProbeFunction_ValidAuthorizer() (gas: 295257)
[PASS] test_ProposePQCAuthorizer() (gas: 297936)
[PASS] test_RevertWhen_CancelPendingUpdateWithNoPending() (gas: 13281)
[PASS] test_RevertWhen_DeployWithZeroAuthorizer() (gas: 111524)
[PASS] test_RevertWhen_FinalizePQCAuthorizerBeforeTimelock() (gas: 297687)
[PASS] test_RevertWhen_FinalizePQCAuthorizerNoPending() (gas: 13760)
[PASS] test_RevertWhen_ProbeWithNoPendingAuthorizer() (gas: 14029)
[PASS] test_RevertWhen_ProposePQCAuthorizerByNonOwner() (gas: 247276)
[PASS] test_RevertWhen_ProposePQCAuthorizerWhilePendingExists() (gas: 525464)
[PASS] test_RevertWhen_ProposePQCAuthorizerZeroAddress() (gas: 11595)
[PASS] test_SequentialUpgrades() (gas: 769170)
[PASS] test_Transfer() (gas: 57262)
[PASS] test_TransferFrom() (gas: 69912)
[PASS] test_TransferLock_IfBadAuthorizerDeployed() (gas: 375598)
[PASS] test_TransfersWorkDuringPendingUpgrade() (gas: 330942)
Suite result: ok. 29 passed; 0 failed; 0 skipped; finished in 15.00ms (32.77ms
CPU time)

Ran 2 test suites in 126.80ms (16.62ms CPU time): 60 tests passed, 0 failed, 0
skipped (60 total tests)
```

11.2 External Functions Check Points

1. Checkout.md

File: src/QONE-V2.sol

contract: QONE is ERC20, Ownable

(Empty fields in the table represent things that are not required or relevant)

Index	Function	StateMutability	Modifier	Param Check	IsUserInterface	Unit Test	Miscellaneous
1	proposePQCAuthorizer(address)		`onlyOwner`		False	Passed	
2	cancelPendingUpdate()		`onlyOwner`		False	Passed	
3	finalizePQCAuthorizer(bytes)		`onlyOwner`		False	Passed	
4	probePendingAuthorizer(address)	view					

File: src/TokenVestingV2.sol

contract: TokenVestingV2 is Ownable, ReentrancyGuard

(Empty fields in the table represent things that are not required or relevant)

Index	Function	StateMutability	Modifier	Param Check	IsUserInterface	Unit Test	Miscellaneous
1	setMerkleRoot(uint8,bytes32)		`onlyOwner, validCategory(category)`		False	Passed	
2	initializeMyAllocation(uint256,uint8,bytes32[])		`validCategory(category)`		Yes	Passed	
3	claim(uint8)		`nonReentrant, validCategory(category)`		Yes	Passed	
4	getClaimableAmount(address,uint8)	view				Passed	
5	getVestedAmount(address,uint8)	view				Passed	
6	getAllocation(address,uint8)	view				Passed	
7	getVestingSchedule(uint8)	view	`validCategory(category)`			Passed	
8	emergencyWithdraw(uint256)		`onlyOwner`		False	Passed	
9	withdrawExcess()		`onlyOwner`		False	Passed	

FAIRYPROOF

https://medium.com/@FairyproofT

https://twitter.com/FairyproofT

https://www.linkedin.com/company/fairyproof-tech

https://t.me/Fairyproof_tech

Reddit: https://www.reddit.com/user/FairyproofTech