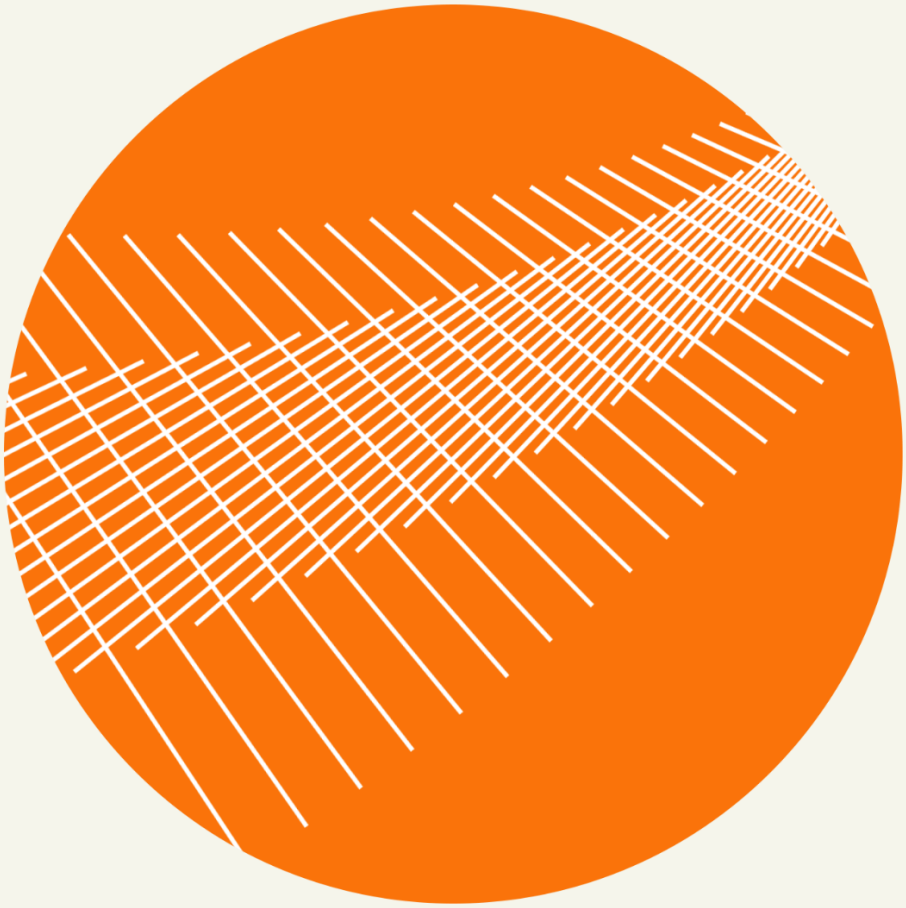
CTDSEC

YOU ARE PROTECTED



TERMS

Smart contract security audit

QFlow - PASS

Table of Contents

1.0 Introduction

1.1 Project engagement

During September of 2026, QFlow team engaged CTDSec to audit smart contracts that they created. The engagement was technical in nature and focused on identifying security flaws in the design and implementation of the contracts.

1.2 Disclaimer

It should be noted that this audit is not an endorsement of the reliability or effectiveness of the contract, rather limited to an assessment of the logic and implementation. In order to ensure a secure contract that's able to withstand the network's fast-paced and rapidly changing environment, we at CTDSec recommend that QFlow team put in place a bug bounty program to encourage further and active analysis of the smart contract.

2.0 Coverage

2.1 Target Code and Revision

For this audit, we performed research, investigation, and review of the QFlow contracts followed by issue reporting, along with mitigation and remediation instructions outlined in this report. The following code files are considered in-scope for the review:

Source file:

https://bscscan.com/address/0xdC0F4179492356F7bf567F80dFB350346F5B9B9A#code

2.2 Attacks made to the contract

In order to check for the security of the contract, we tested several attacks in order to make sure that the contract is secure and follows best practices.

№	Issue description.
1	Compiler warnings.
2	Race conditions and Reentrancy. Cross-function race conditions.
3	Possible delays in data delivery.
4	Oracle calls.
5	Front running.
6	Timestamp dependence.
7	Integer Overflow and Underflow.
8	DoS with Revert.
9	DoS with block gas limit.
10	Methods execution permissions.
11	Economy model. If application logic is based on an incorrect economic model, the application would not function correctly and participants would incur financial losses. This type of issue is most often found in bonus rewards systems, Staking and Farming contracts, Vault and Vesting contracts, etc.
12	The impact of the exchange rate on the logic.
13	Private user data leaks.
14	Malicious Event log.
15	Scoping and Declarations.
16	Uninitialized storage pointers.
17	Arithmetic accuracy.
18	Design Logic.

19	Cross-function race conditions.
20	Safe Zeppelin module.
21	Fallback function security.
22	Overpowered functions / Owner privileges

3.0 Security Issues

3.1 High severity issues [0]

No high severity issues were found

3.2 Medium severity issues [0]

No medium severity issues were found

3.3 Low severity issues [1]

1. Direct transfers become stranded

Location:

Funding and transfer handling [line 82 and 114]

Issue:

The contract maintains two separate values:

```
balanceOf(address(this))
epochPool
```

fundEpochPool() increases both values:

```
super._transfer(_msgSender(), address(this), amount);
epochPool += amount;
```

An ordinary ERC-20 transfer to the token contract changes only its token balance:

```
qflow.transfer(address(qflow), amount);
```

It does not increase epochPool.

executeEpoch() calculates burns exclusively from epochPool, so it cannot burn the untracked surplus. The contract also has no recovery function and cannot call its own transfer() or approve() functions as address(this).

Consequently, directly transferred QFLOW becomes permanently inaccessible.

Recommendation:

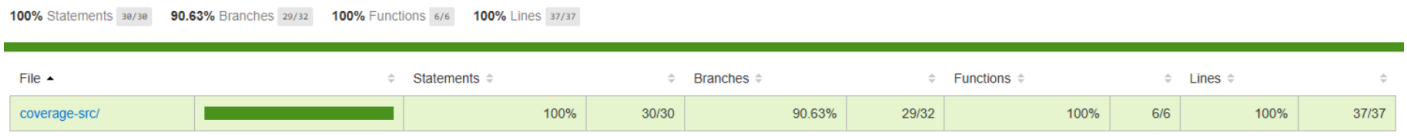
Reject direct transfers or explicitly account for donations.

3.4 Informational Findings [0]

No informational severity issues were found

4.0 Testing coverage

During the testing phase, custom use cases were written to cover all the logic of contracts in python language. *Check “5 Annexes” to see the testing code.*



File	Statements		Branches		Functions		Lines	
coverage-src/	100%	30/30	90.63%	29/32	100%	6/6	100%	37/37

5.0 Annexes

QFLOW Testing:

```
const assert = require('node:assert/strict');

const { ethers, network } = require('hardhat');



const DAY = 24 * 60 * 60;

const base = n => ethers.parseEther(String(n));

async function sent(promise) { return (await promise).wait(); }

async function fails(fn, fragment) {

  try { await fn(); assert.fail('Expected transaction to fail'); }

  catch (e) {

    if (e.message === 'Expected transaction to fail') throw e;

    if (fragment) assert.ok(String(e).includes(fragment), String(e));

  }

}

async function deploy(supply = base(2_000_000)) {

  const [owner] = await ethers.getSigners();

  const factory = await
ethers.getContractFactory('audit/coverage-src/QFlow.sol:QFlow');

  const token = await factory.deploy(supply, owner.address);

  await token.waitForDeployment();

  return token;
```

```
}

async function jump(seconds) { await network.provider.send('evm_increaseTime',
[seconds]); await network.provider.send('evm_mine'); }


describe('QFlow functional and adversarial coverage', function () {

  let owner, user, spender, attacker;

  beforeEach(async function () { [owner, user, spender, attacker] = await
ethers.getSigners(); });


  it('reproduces the pasted 30-day, 30-call one-transaction burn exactly', async
function () {

    const q = await deploy();

    await sent(q.setEpochConfig(1000, DAY));

    const first = await q.nextEpochAt();

    await jump(30 * DAY);

    await fails(() => q.connect(attacker).executeEpoch(), 'Nothing to burn');

    assert.equal(await q.nextEpochAt(), first);

    await sent(q.fundEpochPool(base(1_000_000)));

    const caller = await (await
ethers.getContractFactory('EpochCaller.sol:EpochCaller')).connect(attacker).deplo
y();

    await caller.waitForDeployment();

    await sent(caller.connect(attacker).batch(await q.getAddress(), 30));

    let expected = base(1_000_000);

    for (let i = 0; i < 30; i++) expected -= expected * 1000n / 10000n;
```

```
    assert.equal(expected, 42391158275216203514296n);

    assert.equal(await q.epochPool(), expected);

    assert.equal(await q.balanceOf(await q.getAddress()), expected);

    assert.equal(await q.totalSupply(), base(2_000_000) - (base(1_000_000) -
expected));

    assert.equal(await q.balanceOf(await caller.getAddress()), 0n);

    assert.equal(await q.nextEpochAt(), first + BigInt(30 * DAY));

  });


  it('confirms direct transfers strand excess outside the burn pool', async
function () {

    const q = await deploy();

    await sent(q.connect(owner).transfer(await q.getAddress(), base(100)));

    assert.equal(await q.balanceOf(await q.getAddress()), base(100));

    assert.equal(await q.epochPool(), 0n);

    await sent(q.setEpochConfig(10000, 1));

    await jump(1);

    await fails(() => q.connect(attacker).executeEpoch(), 'Nothing to burn');

    await sent(q.fundEpochPool(base(10)));

    await sent(q.connect(attacker).executeEpoch());

    assert.equal(await q.epochPool(), 0n);

    assert.equal(await q.balanceOf(await q.getAddress()), base(100));

    await fails(() => q.connect(attacker).transferFrom(q.target,
attacker.address, base(1)), 'insufficient allowance');

  });
```

```
  it('enforces ownership and two-step transfer', async function () {
    const q = await deploy();
    await fails(() => q.connect(attacker).setEpochConfig(1000, 1), 'caller is not the owner');
    await fails(() => q.connect(attacker).fundEpochPool(base(1)), 'caller is not the owner');
    await fails(() => q.connect(attacker).setTransferBurnBps(10), 'caller is not the owner');
    await sent(q.transferOwnership(user.address));
    await fails(() => q.connect(attacker).acceptOwnership(), 'caller is not the new owner');
    await sent(q.connect(user).acceptOwnership());
    assert.equal(await q.owner(), user.address);
    await fails(() => q.setTransferBurnBps(10), 'caller is not the owner');
    await sent(q.connect(user).setTransferBurnBps(10));
  });

  it('enforces configuration bounds and epoch timing', async function () {
    const q = await deploy();
    const factory = await ethers.getContractFactory('QFlow.sol:QFlow');
    await fails(() => factory.deploy(base(1), ethers.ZeroAddress), 'Owner is zero');
    await fails(() => q.setTransferBurnBps(1001), 'Too high');
    await fails(() => q.setEpochConfig(10001, 1), 'Too high');
```

```
    await fails(() => q.setEpochConfig(1, 0), 'Duration = 0');

    await fails(() => q.fundEpochPool(0), 'Amount = 0');

    await fails(() => q.executeEpoch(), 'Epoch burn disabled');

    await sent(q.setEpochConfig(5000, DAY));

    await sent(q.fundEpochPool(base(100)));

    await fails(() => q.executeEpoch(), 'Too early');

    await jump(DAY);

    await sent(q.connect(attacker).executeEpoch());

    assert.equal(await q.epochPool(), base(50));

    assert.equal(await q.totalSupply(), base(2_000_000) - base(50));

  });


  it('charges transfer fees and gross allowances without inflating supply', async
function () {

    const q = await deploy(base(1000));

    await sent(q.setTransferBurnBps(1000));

    await sent(q.transfer(user.address, base(100)));

    assert.equal(await q.balanceOf(user.address), base(90));

    assert.equal(await q.totalSupply(), base(990));

    await sent(q.approve(spender.address, base(100)));

    await sent(q.connect(spender).transferFrom(owner.address, attacker.address,
base(100)));

    assert.equal(await q.allowance(owner.address, spender.address), 0n);

    assert.equal(await q.balanceOf(attacker.address), base(90));
```

```
    assert.equal(await q.totalSupply(), base(980));

    await fails(() => q.connect(spender).transferFrom(owner.address,
attacker.address, 1), 'insufficient allowance');

    await sent(q.fundEpochPool(base(10)));

    assert.equal(await q.epochPool(), base(10));

    assert.equal(await q.balanceOf(await q.getAddress()), base(10));

  });


  it('checks a deterministic sequence of balance, supply and pool invariants',
async function () {

    const q = await deploy(base(1000));

    const participants = [owner, user, spender, attacker];

    let seed = 0x51f10a;

    const random = n => { seed ^= seed << 13; seed ^= seed >>> 17; seed ^= seed
<< 5; return (seed >>> 0) % n; };

    await sent(q.setTransferBurnBps(100));

    await sent(q.setEpochConfig(2500, 3));

    for (let i = 0; i < 80; i++) {

      const op = random(5);

      if (op <= 1) {

        const from = participants[random(4)];

        const to = participants[random(4)];

        const balance = await q.balanceOf(from.address);

        const amount = balance * BigInt(random(301)) / 1000n;

        await sent(q.connect(from).transfer(to.address, amount));
```

```
      } else if (op === 2) {
        const balance = await q.balanceOf(owner.address);
        const amount = balance / 100n;
        if (amount > 0n) await sent(q.fundEpochPool(amount));
      } else if (op === 3) {
        await jump(random(10));
        const deadline = await q.nextEpochAt();
        const now = BigInt((await ethers.provider.getBlock('latest')).timestamp);
        if (now >= deadline && (await q.epochPool()) * 2500n / 10000n > 0n) await
sent(q.connect(attacker).executeEpoch());
      } else {
        await sent(q.setTransferBurnBps(random(1001)));
      }
      const balances = await Promise.all([...participants.map(p =>
q.balanceOf(p.address)), q.balanceOf(await q.getAddress())]);
      assert.equal(balances.reduce((a, b) => a + b, 0n), await q.totalSupply(),
`supply invariant after step ${i}`);
      assert.ok(await q.epochPool() <= balances[4], `pool backing after step
${i}`);
    }
  });
});
```

```
const fs = require('node:fs');
```

```
const path = require('node:path');

const assert = require('node:assert/strict');

const crypto = require('node:crypto');

const solc = require('solc');

const ganache = require('ganache');

const { ethers } = require('ethers');

const root = path.resolve(__dirname, '..');

process.chdir(root);

const results = { compiler: solc.version(), seed: '0x51f10a', tests: [],
evidence: {} };

const original = fs.readFileSync('audit/original/QFlow.sol', 'utf8');

const source = fs.readFileSync('QFlow.sol', 'utf8');

assert.equal(source.replaceAll('./vendor/openzeppelin-contracts-4.9.6/contracts/'
, '@openzeppelin/contracts/').replaceAll('\r\n', '\n'),
original.replaceAll('\r\n', '\n'));

results.sourceSha256 =
crypto.createHash('sha256').update(fs.readFileSync('QFlow.sol')).digest('hex');

results.originalSha256 =
crypto.createHash('sha256').update(fs.readFileSync('audit/original/QFlow.sol')).d
igest('hex');

const batchSource = '// SPDX-License-Identifier: MIT\npragma solidity ^0.8.19;
interface IEpoch { function executeEpoch() external; } contract EpochCaller {
function batch(address token, uint256 count) external { for (uint256 i; i <
count; ++i) IEpoch(token).executeEpoch(); } }';

function compile(text) {

  const input = { language: 'Solidity', sources: { 'QFlow.sol': { content: text
}, 'audit/EpochCaller.sol': { content: batchSource } }, settings: { optimizer: {
enabled: true, runs: 200 }, evmVersion: 'paris', outputSelection: { '*': { '*':
```

```
['abi', 'evm.bytecode.object', 'evm.deployedBytecode.object'], '': ['ast'] } } }
};

  const out = JSON.parse(solc.compile(JSON.stringify(input), { import: p => {

    const local = p.replace('@openzeppelin/contracts/',
'vendor/openzeppelin-contracts-4.9.6/contracts/');

    try { return { contents: fs.readFileSync(local, 'utf8') }; } catch { return {
error: 'Missing import: ' + local }; }

  } }));

  const errors = (out.errors || []).filter(x => x.severity === 'error');

  assert.equal(errors.length, 0, JSON.stringify(errors));

  return out;

}

const compiled = compile(source);

const originalCompiled = compile(original);

const artifact = compiled.contracts['QFlow.sol'].QFlow;

function stripMetadata(hex) { const bytes = parseInt(hex.slice(-4), 16); return
hex.slice(0, -(bytes + 2) * 2); }

assert.equal(stripMetadata(artifact.evm.deployedBytecode.object),
stripMetadata(originalCompiled.contracts['QFlow.sol'].QFlow.evm.deployedBytecode.
object));

results.importRewrite = 'Only import paths changed; executable runtime bytecode
identical after stripping source metadata.';

fs.writeFileSync('audit/compiler-diagnostics.json',
JSON.stringify(compiled.errors || [], null, 2));

const rpc = ganache.provider({ logging: { quiet: true }, wallet: { deterministic:
true, totalAccounts: 6 }, chain: { hardfork: 'merge', time: new
Date('2026-01-01T00:00:00Z') }, miner: { timestampIncrement: 0 } });
```

```
const provider = new ethers.BrowserProvider(rpc, undefined, { cacheTimeout: -1
});

provider.pollingInterval = 10;

let signers, addresses;

const MAX = (1n << 256n) - 1n;

async function tx(p) { return (await p).wait(); }

async function reverts(fn) { await assert.rejects(async () => { const result =
await fn(); if (result && result.wait) await result.wait(); }); }

async function panic11(fn) { try { await fn(); assert.fail('Expected overflow');
} catch (e) { assert.equal(e.data,
'0x4e487b7100000000000000000000000000000000000000000000000000000000000000011'); }
}

async function deploy(supply = 10n ** 24n, owner = addresses[0]) {

  const token = await new ethers.ContractFactory(artifact.abi,
artifact.evm.bytecode.object, signers[0]).deploy(supply, owner);

  await token.waitForDeployment();

  return token;

}

async function advance(seconds) { await rpc.request({ method: 'evm_increaseTime',
params: [seconds] }); await rpc.request({ method: 'evm_mine', params: [] }); }

async function test(name, fn) {

  const snapshot = await rpc.request({ method: 'evm_snapshot', params: [] });

  try { await fn(); results.tests.push({ name, status: 'PASS' });
console.log('PASS ' + name); }

  catch (e) { results.tests.push({ name, status: 'FAIL', error: e.stack }); throw
e; }

  finally { await rpc.request({ method: 'evm_revert', params: [snapshot] }); }
```

```
}

async function main() {

  signers = await Promise.all(Array.from({ length: 5 }, (_, i) =>
provider.getSigner(i)));

  addresses = await Promise.all(signers.map(s => s.getAddress()));

  await test('Constructor: supplied owner receives exact supply; zero owner
rejected', async () => {

    const c = await deploy(12345n, addresses[1]);

    assert.equal(await c.owner(), addresses[1]); assert.equal(await
c.totalSupply(), 12345n); assert.equal(await c.balanceOf(addresses[1]), 12345n);

    await reverts(() => deploy(1n, ethers.ZeroAddress));

  });

  await test('Non-owner cannot configure, fund, transfer ownership, or renounce',
async () => {

    const c = (await deploy()).connect(signers[1]);

    for (const fn of [() => c.setTransferBurnBps(0), () => c.setEpochConfig(100,
1), () => c.fundEpochPool(1), () => c.transferOwnership(addresses[1]), () =>
c.renounceOwnership()]) await reverts(fn);

  });

  await test('Ownership requires pending-owner acceptance; old owner loses
privileges', async () => {

    const c = await deploy(); await tx(c.transferOwnership(addresses[1]));

    await reverts(() => c.connect(signers[2]).acceptOwnership());

    await reverts(() => c.connect(signers[1]).setTransferBurnBps(10));

    await tx(c.connect(signers[1]).acceptOwnership());
```

```
    assert.equal(await c.owner(), addresses[1]); await reverts(() =>
c.setTransferBurnBps(10));

    await tx(c.connect(signers[1]).setTransferBurnBps(10));

  });

  await test('Configuration bounds and zero funding rejected', async () => {

    const c = await deploy();

    await reverts(() => c.setTransferBurnBps(1001)); await reverts(() =>
c.setEpochConfig(10001, 1));

    await reverts(() => c.setEpochConfig(1, 0)); await reverts(() =>
c.fundEpochPool(0));

    await tx(c.setTransferBurnBps(1000)); await tx(c.setEpochConfig(10000, 1));

  });

  await test('Taxed transfer conserves balances plus burn; funding bypasses tax
exactly', async () => {

    const c = await deploy(100000n); await tx(c.setTransferBurnBps(1000));

    await tx(c.transfer(addresses[1], 10000));

    assert.equal(await c.balanceOf(addresses[0]), 90000n); assert.equal(await
c.balanceOf(addresses[1]), 9000n); assert.equal(await c.totalSupply(), 99000n);

    await tx(c.fundEpochPool(10000)); assert.equal(await c.epochPool(), 10000n);
assert.equal(await c.balanceOf(c.target), 10000n); assert.equal(await
c.totalSupply(), 99000n);

  });

  await test('transferFrom spends gross allowance; infinite allowance and absent
approval behave correctly', async () => {

    const c = await deploy(100000n); await tx(c.setTransferBurnBps(1000)); await
tx(c.approve(addresses[1], 10000));
```

```
    await tx(c.connect(signers[1]).transferFrom(addresses[0], addresses[2],
10000));

    assert.equal(await c.allowance(addresses[0], addresses[1]), 0n);
assert.equal(await c.balanceOf(addresses[2]), 9000n);

    await reverts(() => c.connect(signers[1]).transferFrom(addresses[0],
addresses[2], 1));

    await tx(c.approve(addresses[1], MAX)); await
tx(c.connect(signers[1]).transferFrom(addresses[0], addresses[2], 100));

    assert.equal(await c.allowance(addresses[0], addresses[1]), MAX);

  });

  await test('Failed on-chain transfers atomically restore burn, balances,
supply, and allowance', async () => {

    const c = await deploy(100n); await tx(c.setTransferBurnBps(1000)); await
tx(c.approve(addresses[1], 1000));

    await reverts(() => c.connect(signers[1]).transferFrom(addresses[0],
addresses[2], 101, { gasLimit: 300000 }));

    await reverts(() => c.transfer(ethers.ZeroAddress, 50, { gasLimit: 300000
}));

    assert.equal(await c.totalSupply(), 100n); assert.equal(await
c.balanceOf(addresses[0]), 100n); assert.equal(await c.allowance(addresses[0],
addresses[1]), 1000n);

  });

  await test('Self-transfers only destroy the fee; zero transfers preserve
supply', async () => {

    const c = await deploy(1000n); await tx(c.setTransferBurnBps(1000)); await
tx(c.transfer(addresses[0], 100));

    assert.equal(await c.balanceOf(addresses[0]), 990n); assert.equal(await
c.totalSupply(), 990n);
```

```
    await tx(c.transfer(addresses[1], 0)); assert.equal(await c.totalSupply(),
990n);

  });

  await test('Epoch gates: disabled, early, empty; permissionless execution burns
pool only', async () => {

    const c = await deploy(100000n); await reverts(() => c.executeEpoch());

    await tx(c.setEpochConfig(1000, 100)); await tx(c.fundEpochPool(10000));

    await reverts(() => c.connect(signers[2]).executeEpoch()); const before =
await c.nextEpochAt();

    await advance(100); await tx(c.connect(signers[2]).executeEpoch());

    assert.equal(await c.epochPool(), 9000n); assert.equal(await
c.balanceOf(c.target), 9000n); assert.equal(await c.totalSupply(), 99000n);

    assert.equal(await c.balanceOf(addresses[0]), 90000n); assert.equal(await
c.balanceOf(addresses[2]), 0n); assert.equal(await c.nextEpochAt(), before +
100n);

    await tx(c.setEpochConfig(10000, 1)); await advance(1); await
tx(c.executeEpoch());

    assert.equal(await c.epochPool(), 0n); await advance(1); await reverts(() =>
c.executeEpoch());

  });

  await test('Observation: a direct token donation is not credited to epochPool
or burnable through it', async () => {

    const c = await deploy(10000n); await tx(c.setTransferBurnBps(1000)); await
tx(c.transfer(c.target, 1000));

    await tx(c.fundEpochPool(1000)); await tx(c.setEpochConfig(10000, 1)); await
advance(1); await tx(c.executeEpoch());

    assert.equal(await c.epochPool(), 0n); assert.equal(await
c.balanceOf(c.target), 900n);
```

```
    await reverts(() => c.connect(signers[1]).transferFrom(c.target,
addresses[1], 1));

    results.evidence.directDonation = { grossDonation: '1000', transferBurn:
'100', strandedBalance: '900', remainingEpochPool: '0' };

  });

  await test('Observation: overdue epochs burn newly funded tokens in one
attacker transaction', async () => {

    const c = await deploy(10000n); await tx(c.setEpochConfig(5000, 100));

    const initialDeadline = await c.nextEpochAt(); await advance(1000); await
tx(c.fundEpochPool(1024));

    const a = compiled.contracts['audit/EpochCaller.sol'].EpochCaller;

    const caller = await new ethers.ContractFactory(a.abi, a.evm.bytecode.object,
signers[2]).deploy(); await caller.waitForDeployment();

    const receipt = await tx(caller.batch(c.target, 10));

    assert.equal(await c.epochPool(), 1n); assert.equal(await
c.balanceOf(c.target), 1n); assert.equal(await c.totalSupply(), 8977n);

    assert.equal(await c.nextEpochAt(), initialDeadline + 1000n);
assert.equal(await c.balanceOf(caller.target), 0n);

    results.evidence.overdue = { newlyFunded: '1024', epochsInOneTransaction: 10,
burned: '1023', remaining: '1', attackerProceeds: '0', transactionHash:
receipt.hash };

  });

  await test('Observation: fractional-unit burn rounding leaves dust and stalls
the deadline', async () => {

    const c = await deploy(10000n); await tx(c.setEpochConfig(1, 1)); await
tx(c.fundEpochPool(9999)); await advance(1);
```

```
    const next = await c.nextEpochAt(); await reverts(() => c.executeEpoch());
assert.equal(await c.nextEpochAt(), next); assert.equal(await c.epochPool(),
9999n);

    await tx(c.fundEpochPool(1)); await tx(c.executeEpoch()); assert.equal(await
c.epochPool(), 9999n);

  });

  await test('Observation: checked multiplication overflows only at extreme token
amounts', async () => {

    const c = await deploy(MAX); await tx(c.setTransferBurnBps(1000));

    const threshold = MAX / 1000n + 1n; await panic11(() =>
c.transfer.staticCall(addresses[1], threshold));

    await tx(c.transfer(addresses[1], MAX / 1000n));

    await tx(c.setEpochConfig(10000, 1)); const pool = MAX / 10000n + 1n; await
tx(c.fundEpochPool(pool)); await advance(1);

    await panic11(() => c.executeEpoch.staticCall());

    results.evidence.overflow = { transferFirstOverflowAt1000Bps:
String(threshold), epochFirstOverflowAt10000Bps: String(pool), panic: '0x11' };

  });

  await test('Observation: splitting sub-fee base-unit transfers avoids only
dust-sized burns', async () => {

    const c = await deploy(100n); await tx(c.setTransferBurnBps(1000));

    for (let i = 0; i < 10; i++) await tx(c.transfer(addresses[1], 9));

    assert.equal(await c.balanceOf(addresses[1]), 90n); assert.equal(await
c.totalSupply(), 100n);

  });

  await test('Deterministic stateful model: 180 mixed operations preserve supply,
pool, and exact balances', async () => {
```

```
    const c = await deploy(); const participants = [...addresses.slice(0, 4),
c.target];

    const balances = [10n ** 24n, 0n, 0n, 0n, 0n]; let supply = balances[0], pool
= 0n, bps = 0n, epochBps = 0n, next = 0n, duration = 0n;

    let seed = 0x51f10a; const random = n => { seed ^= seed << 13; seed ^= seed
>>> 17; seed ^= seed << 5; return (seed >>> 0) % n; };

    let now = BigInt((await rpc.request({ method: 'eth_getBlockByNumber', params:
['latest', false] })).timestamp);

    for (let i = 0; i < 180; i++) {

      const op = random(6);

      if (op === 0 || op === 1) {

        const from = random(4), to = random(5), amount = balances[from] === 0n ?
0n : balances[from] * BigInt(random(10001)) / 10000n;

        const burn = amount * bps / 10000n;

        await tx(c.connect(signers[from]).transfer(participants[to], amount));
balances[from] -= amount; balances[to] += amount - burn; supply -= burn;

      } else if (op === 2) {

        bps = BigInt(random(1001)); await tx(c.setTransferBurnBps(bps));

      } else if (op === 3) {

        const amount = balances[0] * BigInt(random(1001)) / 10000n;

        if (amount > 0n) { await tx(c.fundEpochPool(amount)); balances[0] -=
amount; balances[4] += amount; pool += amount; }

      } else if (op === 4) {

        epochBps = BigInt(random(10001)); duration = BigInt(random(100) + 1);
await tx(c.setEpochConfig(epochBps, duration)); next = now + duration;

      } else {
```

```
        const delta = random(200); await advance(delta); now += BigInt(delta);

        const burn = pool * epochBps / 10000n;

        if (epochBps > 0n && duration > 0n && now >= next && burn > 0n) { await
tx(c.connect(signers[3]).executeEpoch()); pool -= burn; balances[4] -= burn;
supply -= burn; next += duration; }

        else await reverts(() => c.executeEpoch.staticCall());

      }

      assert.equal(await c.totalSupply(), supply, 'supply step ' + i);
assert.equal(await c.epochPool(), pool, 'pool step ' + i);

      assert.equal(await c.nextEpochAt(), next, 'deadline step ' + i);

      const actual = await Promise.all(participants.map(a => c.balanceOf(a)));

      assert.deepEqual(actual, balances, 'balances step ' + i);
assert.equal(balances.reduce((a, b) => a + b, 0n), supply); assert.ok(pool <=
balances[4]);

    }

    results.evidence.statefulModel = { operations: 180, seed: results.seed,
invariants: ['exact per-account balances', 'total supply equals sum of balances',
'epochPool <= contract token balance', 'epoch deadline matches model'] };

  });

}

main().then(() => { results.status = 'PASS'; }).catch(e => { results.status =
'FAIL'; console.error(e); process.exitCode = 1; }).finally(async () => {

  fs.writeFileSync('audit/test-results.json', JSON.stringify(results, null, 2));

  await rpc.disconnect();

});
```

6.0 Summary of the audit [PASS ✅]

The QFlow security audit passed with no confirmed Critical, High, or Medium vulnerabilities. The previously detected issue during testing overdue-epoch behavior was confirmed as intentional protocol design.

Holder-protection review found:

1. No hidden or post-deployment minting capability.
2. No blacklist, whitelist, pause, or wallet-freezing mechanism.
3. No function allowing the owner to seize, transfer, confiscate, or burn tokens from arbitrary holders.
4. No hidden balance modification or allowance bypass.
5. No upgradeability, proxy, delegatecall, external-call, or reentrancy mechanism.
6. Owner privileges are limited to configuring burn rates and epoch settings, funding the epoch pool from the owner's own balance, and standard two-step ownership management.
7. Transfer burns are capped at 10% and apply according to the public configuration.

Vulnerability Level	Total	Pending	Not Apply	Acknowledged	Partially Resolved	Resolved
High						
Medium						
Low	1			1		
Informational						