StatusL2 Audit Report

Prepared by Cyfrin

Version 2.0

Lead Auditors

Farouk

Samuraii77

T1MOH

January 5, 2026

Contents

1 About Cyfrin

Cyfrin is a Web3 security company dedicated to bringing industry-leading protection and education to our partners and their projects. Our goal is to create a safe, reliable, and transparent environment for everyone in Web3 and DeFi. Learn more about us at cyfrin.io.

2 Disclaimer

The Cyfrin team makes every effort to find as many vulnerabilities in the code as possible in the given time but holds no responsibility for the findings in this document. A security audit by the team does not endorse the underlying business or product. The audit was time-boxed and the review of the code was solely on the security aspects of the solidity implementation of the contracts.

3 Risk Classification

	Impact: High	**Impact: Medium**	**Impact: Low**
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

4 Protocol Summary

The StatusL2 KARMA protocol is a staking and reputation system designed to reward long term participation through time based accrual mechanics. Users stake tokens into protocol managed vaults and earn non transferable Multiplier Points (MP) proportional to stake size and duration, subject to strict caps to prevent unbounded growth.

Stakes accrue MP linearly over time, with a per stake maximum corresponding to a fixed accrual period (typically four years). Accrued MP contributes to a user's overall KARMA score and is used to determine tiers, reputation, and reward eligibility. The protocol also enforces a global MP cap intended to bound total system influence.

The system is composed of staking vaults deployed via a factory pattern, with core logic implemented in StakeManager, StakeVault, and supporting math libraries. MP balances feed into tiering and reputation modules (Karma, KarmaTiers) and are used by reward distributors to compute user entitlements.

Additional components include an airdrop mechanism based on Merkle proofs and signature delegation, NFT representations of user KARMA state, and access control primitives that define trusted deployment and configuration boundaries.

Overall, the protocol combines capped, time weighted staking with reputation based rewards. Correct enforcement of MP accrual limits and global caps is critical to preserving the intended economic guarantees and system safety.

5 Audit Scope

The audit scope was limited to:

```
// core
status-network-contracts/src/math/MultiplierPointMath.sol
status-network-contracts/src/math/StakeMath.sol
status-network-contracts/src/nft-metadata-generators/BaseNFTMetadataGenerator.sol
status-network-contracts/src/nft-metadata-generators/NFTMetadataGeneratorSVG.sol
status-network-contracts/src/nft-metadata-generators/NFTMetadataGeneratorURL.sol
status-network-contracts/src/rln/RLN.sol
```

```
status-network-contracts/src/rln/PoseidonHasher.sol
status-network-contracts/src/utils/ERC20VotesUpgradeable.sol
status-network-contracts/src/Karma.sol
status-network-contracts/src/KarmaAirdrop.sol
status-network-contracts/src/KarmaNFT.sol
status-network-contracts/src/KarmaTiers.sol
status-network-contracts/src/StakeManager.sol
status-network-contracts/src/SimpleKarmaDistributor.sol
status-network-contracts/src/StakeVault.sol
status-network-contracts/src/TrustedCodehashAccess.sol
status-network-contracts/src/VaultFactory.sol

// deployment scripts
status-network-contracts/script/Base.s.sol
status-network-contracts/script/DeployKarma.s.sol
status-network-contracts/script/DeployKarmaAirdrop.s.sol
status-network-contracts/script/DeployKarmaNFT.s.sol
status-network-contracts/script/DeployKarmaTiers.s.sol
status-network-contracts/script/DeploymentConfig.s.sol
status-network-contracts/script/DeployMetadataGenerator.s.sol
status-network-contracts/script/DeployProtocol.s.sol
status-network-contracts/script/DeploySimpleKarmaDistributor.s.sol
status-network-contracts/script/DeployStakeManager.s.sol
status-network-contracts/script/DeployVaultFactory.s.sol
status-network-contracts/script/RLN.s.sol
status-network-contracts/script/UpgradeKarma.s.sol
status-network-contracts/script/UpgradeStakeManager.s.sol
```

6 Executive Summary

Over the course of 15 days, the Cyfrin team conducted an audit on the StatusL2 smart contracts provided by StatusL2. In this period, a total of 33 issues were found.

StatusL2 KARMA is a staking and reputation system where users deposit tokens into vaults, accrue time weighted Multiplier Points (MP), and redeem rewards via distributor contracts. A separate RLN module provides spam resistance by registering identity commitments and slashing misbehaving members, while governance power is represented through an ERC20Votes style token and optional NFT metadata wrappers. The security model depends on correct global and per vault accounting (MP caps, reward indices, migration state), strict vault registration boundaries, and reliable slashing execution paths even under degraded conditions (pauses, emergency mode, or distributor failures).

Our review focused on staking and MP accrual invariants, vault factory and registration trust assumptions, reward redemption and distributor interactions, migration and lock enforcement, and the RLN slashing flow including its Poseidon hashing boundary. The most severe risks clustered around protocol wide safety invariants and enforceability: a broken global MP cap invariant can trigger arithmetic underflow and permanently DoS core flows, lock semantics could previously be bypassed to extract outsized rewards, and slashing could be made impossible either via gas exhaustion from unbounded vault registration or misconfiguration that breaks reward redemption across distributors. Medium severity issues were concentrated around operational safety and user value retention including paused distributor behavior blocking slashing, airdrop claim griefing via signature frontruns, and migration edge cases that can overwrite accrued rewards. Lower severity items largely related to deployment scripts, tier initialization correctness, RLN registry capacity behavior, and standards compliance in events and cryptographic assumptions.

Overall, the architecture is modular and the core flows are directionally sound, but the system still needs hardening around global invariants, slashing liveness under pause and emergency conditions, and deployment and configuration correctness. Given the severity and breadth of the issues identified, we recommend a follow up audit after fixes are implemented and before deployment to validate the remediations and reduce the risk of regressions.

Summary

Project Name	StatusL2
Repository	status-network-monorepo
Commit	6ab031b9d97d...
Audit Timeline	Dec 1st - Dec 19th, 2025
Methods	Manual Review

Issues Found

Critical Risk	2
High Risk	6
Medium Risk	8
Low Risk	8
Informational	9
Gas Optimizations	0
Total Issues	33

Summary of Findings

[C-1] Any staker can fully avoid slashing by triggering OOG reverts	Resolved
[C-2] User can double claim airdrop	Resolved
[H-1] Malicious actors can force vaults to exit if they wish to get rewards	Resolved
[H-2] Malicious actors can get free rewards if contract gets paused	Resolved
[H-3] Anyone can dodge reveal delays by providing unrelated account	Resolved
[H-4] Any slasher can increase another one's reveal delays	Resolved
[H-5] User can bypass lock and withdraw stake anytime	Resolved
[H-6] Global MP cap invariant can be broken on unstake causing arithmetic underflow and permanent DoS	Resolved
[M-1] `ERC20VotesUpgradeable::getPastTotalSupply` overestimates the value	Acknowledged
[M-2] Users can be overslashed in `Karma.sol`	Resolved
[M-3] Commit-reveal does not sufficiently protect against slash frontrunnings	Resolved
[M-4] `Karma.sol` doesn't initialize signature variables	Resolved
[M-5] Griefer can block user from claiming airdrop	Resolved
[M-6] User can lose accrued rewards during migration	Resolved
[M-7] `DeployProtocol.s.sol` incorrectly allows transfers to `StakeManager.sol`	Resolved

[M-8] Slashing doesn't work if any reward distributor is paused	Resolved
[L-1] Unstaking rounds in favor of users	Resolved
[L-2] Migrations to self are possible	Resolved
[L-3] `totalMPAccrued` accrual should start from first stake	Resolved
[L-4] `totalMaxMP` incorrectly accounts for individual `maxMP`	Resolved
[L-5] New users can't be registered after slashing contrary to documentation	Resolved
[L-6] `KarmaNFT.sol` incorrectly simulates minting event	Resolved
[L-7] `InitializeKarmaTiersScript` doesn't account for decimals	Resolved
[L-8] Potential Divergence Between Off-Chain and On-Chain Poseidon Hashing Causes Unslashable RLN Identities	Resolved
[I-1] Incorrect Natspec in `RLN::setSlashRevealWindowTime`	Resolved
[I-2] Inconsistent naming in `Karma._onlySlasher`	Resolved
[I-3] `KarmaTiers.sol` constructor can be simplified	Resolved
[I-4] Consider removing unused functions in libraries	Acknowledged
[I-5] Consider simplifying formulas in `MultiplierPointMath.sol`	Resolved
[I-6] `DeployProtocol.s.sol` doesn't deploy RLN contracts	Acknowledged
[I-7] `Karma::removeRewardDistributor` burns all virtual Karma	Resolved
[I-8] Emergency mode doesn't save from malicious upgrade of StakeManager.sol	Resolved
[I-9] Slashing won't work after enabling emergency mode	Resolved

7 Findings

7.1 Critical Risk

7.1.1 Any staker can fully avoid slashing by triggering OOG reverts

Description: The finding is similar to *Malicious actors can force vaults to exit if they wish to get rewards*, but actually different (different fix, different impact, similarity is iterating over the vaults to trigger OOG).

Upon slashing, we calculate the redeemable rewards for the account to slash by going over all of the account vaults and summing them up:

```
for (uint256 i = 0; i < accountVaults.length; i++) {
    accountTotalRewards += rewardsBalanceOf(accountVaults[i]);
}
```

The issue is that the account can register a ton of vaults using `VaultFactory::createVault()`:

```
function createVault() external returns (StakeVault clone) {
    clone = StakeVault(Clones.clone(vaultImplementation));
    clone.initialize(msg.sender, stakeManager);
    clone.register();
    emit VaultCreated(address(clone), msg.sender);
}
```

This will cause the loop to go OOG, thus it is impossible to slash that account.

Impact: Slashes are made impossible.

Recommended Mitigation: Consider having a sensible limit on the vaults an account can register, i.e. 10.

StatusL2: Fixed in 6697432.

Cyfrin: Verified.

7.1.2 User can double claim airdrop

Description: `KarmaAirdrop.sol` is pausable. That's because there is functionality to update `merkleRoot`: when it's updated, previously unclaimed tokens are added to the new `merkleRoot`. So following scenario is possible:

1) User have not claimed. `merkleRoot` will be updated to include some new people and previously unclaimed users
2) User frontruns upgrade by claiming his part
3) New `merkleRoot` now again contains his airdrop, so user can claim second time

To prevent this scenario, `KarmaAirdrop.sol` inherits `Pausable.sol`, however `KarmaAirdrop::claim` doesn't have `whenNotPaused` modifier

Impact: During `merkleRoot` upgrade, previously unclaimed users can double claim, stealing tokens.

Recommended Mitigation: Add modifier `whenNotPaused` to `KarmaAirdrop::claim`.

StatusL2: Fixed in ebbf84b.

Cyfrin: Verified.

7.2 High Risk

7.2.1 Malicious actors can force vaults to exit if they wish to get rewards

Description: Rewards are redeemed by iterating over all vaults of an account and calculating the accrued rewards:

```
function redeemRewards(address account) external onlyNotEmergencyMode whenNotPaused returns (uint256) {
    // ...
    for (uint256 i = 0; i < accountVaults.length; i++) {
        // ...
    }
    // ...
}
```

There is no way to receive rewards by specifying a singular vault unless you exit the system completely.

The vaults are pushed upon registration in `StakeManager::registerVault`:

```
function registerVault() external onlyNotEmergencyMode whenNotPaused onlyTrustedCodehash {
    address vault = msg.sender;
    address owner = IStakeVault(vault).owner();

    if (vaultOwners[vault] != address(0)) {
        revert StakeManager__VaultAlreadyRegistered();
    }

    vaultOwners[vault] = owner;
    vaults[owner].push(vault);
    emit VaultRegistered(vault, owner);
}
```

The issue is that anyone can deploy their own vaults (bypassing the factory) using the correct vault implementation address to satisfy `onlyTrustedCodehash` modifier. They can specify any specific owner they are targeting. This will cause the `vaults[victim]` array to be extended and with enough entries, iterating over it will be impossible due to gas limits. Thus, there is no way to redeem rewards unless a legitimate vault exits the system. The fact that some users might have staked for 4 years also makes the issue more serious as they will be forced to wait for the whole period to get any rewards.

The issue also makes functions like `updateAccount` and some other view functions unusable.

Impact: Unable to redeem rewards unless you exit out of the system completely.

Recommended Mitigation: Consider adding access control that only allows factory to register vaults.

StatusL2: Fixed in 5e93ecb.

Cyfrin: Verified.

7.2.2 Malicious actors can get free rewards if contract gets paused

Description: The `StakeVault` contract has a leave functionality with a try/catch block:

```
function leave(address _destination) external onlyOwner validDestination(_destination) {
    hasLeft = true;
    try stakeManager.leave() {
        if (lockUntil <= block.timestamp) {
            depositedBalance = 0;
            bool success = STAKING_TOKEN.transfer(_destination, STAKING_TOKEN.balanceOf(address(this)));
            if (!success) {
                revert StakeVault__FailedToLeave();
            }
        }
    } catch {
        if (lockUntil <= block.timestamp) {
            depositedBalance = 0;
            bool success = STAKING_TOKEN.transfer(_destination, STAKING_TOKEN.balanceOf(address(this)));
            if (!success) {
                revert StakeVault__FailedToLeave();
            }
        }
    }
}
```

It aims to handle bad staking managers that revert, causing a DoS. The issue is that if a legitimate revert can happen (or a forced one, by the caller), then the state in the staking manager will be inconsistent as it will still consider the caller staked, even though he actually withdrew his tokens. This will also allow him to get a part of the rewards even though he has not staked at all. The most obvious way where a revert can happen is if the contract is paused.

For example, the following can happen:

1. The staking manager will be paused for any reason, transaction is submitted.
2. Alice, a malicious actor, frontruns the pause and stakes a huge amount with a 0 lock time.
3. Step 1 transaction executes, contract is paused.
4. Alice immediately leaves after that, it reverts and we end up in the catch block where she simply gets her tokens back.
5. She gets rewards for free as she can keep calling `StakeManager::redeemRewards()`.

No way to force reverts has been found yet (e.g. 63/64 rule abuse is very unlikely to work, also no other reverts that can be forced to trigger).

Impact: Users can get free rewards at the expense of others.

Recommended Mitigation: Best way is to remove the try/catch and simply do not do bad upgrades.

StatusL2: Fixed in d813449.

Cyfrin: Verified.

7.2.3 Anyone can dodge reveal delays by providing unrelated account

Description: `RLN` provides a commit-reveal functionality to slashing. Firstly, a commit happens:

```
function slashCommit(address account, bytes32 hash) external onlyRole(SLASHER_ROLE) {
    uint256 lastReveal = lastRevealStartTime[account];
    uint256 revealStartTime;

    if (lastReveal == 0 || lastReveal + slashRevealWindowTime < block.timestamp) {
        revealStartTime = block.timestamp;
    } else {
        revealStartTime = lastReveal + slashRevealWindowTime;
    }

    slashCommitments[account][hash] = revealStartTime;
    lastRevealStartTime[account] = revealStartTime;
}
```

The hash is based on the private key of the private key and reward recipient address. Thus, a slasher might provide `(bob, hash(PK, slasherAddress))` as the inputs. Then, as he is the first to commit, that would store the earliest reveal start time and would be able to commit first:

```
function slashReveal(
    address account,
    bytes32 privateKey,
    address rewardRecipient
)
    external
    onlyRole(SLASHER_ROLE)
{
    /// forge-lint: disable-next-line(asm-keccak256)
    bytes32 hash = keccak256(abi.encodePacked(privateKey, rewardRecipient));
    uint256 revealStartTime = slashCommitments[account][hash];

    if (revealStartTime == 0) {
        revert RLN__InvalidCommitment();
    }

    if (block.timestamp < revealStartTime) {
        revert RLN__RevealWindowNotStarted();
    }

    delete slashCommitments[account][hash];
    slash(privateKey, rewardRecipient);
}
```

The issue is that the `account` provided actually serves no real purpose. Another slasher can simply provide `(randomAddress, hash(PK, otherSlasherAddress))`. This will get a 0 delay as this account has not been used, thus the reveal will be possible immediately. `slashCommitments` would be based on the `randomAddress, hash` keys, then during the reveal, he simply has to provide `(randomAddress, PK, otherSlasherAddress)` and the account is only used for accessing and deleting that same `slashCommitments` mapping. Thus, it is not enforced at all that the account actually corresponds to the slashed address.

Impact: Any slasher can get the rewards for himself as he is dodging the delay completely.

Recommended Mitigation: Enforce the account provided to actually be the slashed account. The easiest way is to do it upon the reveal when the PK is known and validate it against the user in `members[poseidonHash(privateKey)]`.

StatusL2: Fixed in 62021fc.

Cyfrin: Verified.

7.2.4 Any slasher can increase another one's reveal delays

Description: Commits in `RLN` are made as follows:

```
function slashCommit(address account, bytes32 hash) external onlyRole(SLASHER_ROLE) {
    uint256 lastReveal = lastRevealStartTime[account];
    uint256 revealStartTime;

    if (lastReveal == 0 || lastReveal + slashRevealWindowTime < block.timestamp) {
        revealStartTime = block.timestamp;
    } else {
        revealStartTime = lastReveal + slashRevealWindowTime;
    }

    slashCommitments[account][hash] = revealStartTime;
    lastRevealStartTime[account] = revealStartTime;
}
```

The hash is based on a PK and a reward recipient address. Thus, it seems like an assumption is made that this hash would only ever be used from the legitimate address as he is the one who benefits from having his own reward recipient address. However, any other slasher can simply spam the function a lot using someone else's hash and cause the reveal time for that slash to be very far in the future, thus disallow that slasher from getting the rewards he is entitled to. Then, he can be the one who actually gets these rewards.

Impact: Any slasher can significantly increase another one's reveal delays

Recommended Mitigation: Consider adding another key to `slashCommitments` which holds the address who commits (`msg.sender` during `slashCommit` call).

StatusL2: Fixed in f15f5e9.

Cyfrin: Verified.

7.2.5 User can bypass lock and withdraw stake anytime

Description: Attack scenario is following:

1) User staked with lock, let's say 1000 tokens for 4 years
2) User registers new vault
3) User calls `StakeVault::leave` on locked vault. It sets `vault.stakedBalance = 0` and resets other values:

```
    function leave() external whenNotPaused onlyTrustedCodehash {
        _updateGlobalState();
        _updateVault(msg.sender, false);

        VaultData storage vault = vaultData[msg.sender];

        if (vault.stakedBalance > 0) {
            // calling `_unstake` to update accounting accordingly
@>          _unstake(vault.stakedBalance, vault);
        }

        uint256 rewardsToRedeem = vault.rewardsAccrued;
        // reset accrued rewards
        totalRewardsAccrued -= rewardsToRedeem;
        vault.rewardsAccrued = 0;
        vault.rewardIndex = 0;
        // further cleanup that isn't done in `_unstake`
        vault.lastMPUpdateTime = 0;

        bool success = REWARD_TOKEN.transfer(IStakeVault(msg.sender).owner(), rewardsToRedeem);
        if (!success) {
            revert StakeManager__RewardTransferFailed();
        }
        emit VaultLeft(msg.sender);
    }
```

4) Migrate from new vault to locked. Checks will succeed. Finally it will reset `lockUntil` and `depositedBalance` to 0 in locked vault, which allows to withdraw immediately:

```
    function migrateToVault(address migrateTo)
        external
        onlyNotEmergencyMode
        whenNotPaused
        onlyTrustedCodehash
        onlyRegisteredVault
    {
@>      if (vaultOwners[migrateTo] == address(0)) {
            revert StakeManager__InvalidVault();
        }

@>      if (vaultData[migrateTo].stakedBalance > 0) {
            revert StakeManager__MigrationTargetHasFunds();
        }

        ...

        IStakeVault.MigrationData memory migrationData = IStakeVault.MigrationData({
            lockUntil: IStakeVault(msg.sender).lockUntil(),
            depositedBalance: IStakeVault(msg.sender).depositedBalance()
        });

@>      IStakeVault(migrateTo).migrateFromVault(migrationData);

        delete vaultData[msg.sender];

        emit VaultMigrated(msg.sender, migrateTo);
    }

    function migrateFromVault(MigrationData calldata data) external {
        if (msg.sender != address(stakeManager)) {
            revert StakeVault__NotAuthorized();
        }
@>      lockUntil = data.lockUntil;
        depositedBalance = data.depositedBalance;
    }
```

Impact: Stake can be withdrawn anytime, therefore user can stake at max lock period without any downside. When user stakes without lock, he has `2X` shares earning rewards, however with this trick he has `6X` shares earning rewards - so user earns up to 3x more rewards, basically stealing reward from other users.

Proof of Concept: Insert this into `status-network-contracts/test/BypassLockViaMigration.t.sol`, execute with `forge test --match-test test_a -vv`:

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.26;

import { StakeVault } from "../../src/StakeVault.sol";
import { StakeManagerTest } from "./StakeManagerBase.t.sol";
import { UpgradeStakeManagerScript } from "../../script/UpgradeStakeManager.s.sol";
import { console } from "forge-std/console.sol";

contract BypassLockViaMigrationTest is StakeManagerTest {
    function setUp() public virtual override {
        super.setUp();
    }

    function test_a() public {
        uint256 stakeAmount = 1000e18;
        uint256 lockPeriod = 4 * 365 days; // 4 years

        StakeVault lockedVault = StakeVault(vaults[alice]);

        vm.startPrank(alice);
        stakingToken.approve(address(lockedVault), stakeAmount);
        lockedVault.stake(stakeAmount, lockPeriod);
        vm.stopPrank();

        vm.prank(alice);
        StakeVault emptyVault = vaultFactory.createVault();

        vm.prank(alice);
        lockedVault.leave(alice);

        vm.prank(alice);
        emptyVault.migrateToVault(address(lockedVault));

        uint256 balanceBefore = stakingToken.balanceOf(alice);
        vm.prank(alice);
        lockedVault.withdraw(stakingToken, stakeAmount, alice);
        console.log("tokens withdrawn %e", stakingToken.balanceOf(alice) - balanceBefore);
    }
}
```

Recommended Mitigation: Possible mitigation is to disallow migrating to vault if `vault.hasLeft == true`.

StatusL2: Fixed in fb6c8ef.

Cyfrin: Verified.

7.2.6 Global MP cap invariant can be broken on unstake causing arithmetic underflow and permanent DoS

Description: The protocol assumes the invariant `totalMPAccrued <= totalMaxMP` always holds. This invariant is relied upon in the global MP accrual logic `_totalMP()`, which caps newly accrued MP using an unchecked subtraction:

```
accruedMP = totalMaxMP - totalMPAccrued;
```

However, this invariant can be violated during normal protocol operation due to asymmetric updates in `_unstake()`.

When a vault unstakes, both its accrued MP (`vault.mpAccrued`) and its MP cap (`vault.maxMP`) are reduced proportionally to the unstaked amount:

```
_deltaMpTotal = vault.mpAccrued * amount / vault.stakedBalance;
_deltaMpMax   = vault.maxMP     * amount / vault.stakedBalance;
```

These reductions are then propagated to global aggregates:

```
totalMPAccrued -= _deltaMpTotal;
totalMaxMP     -= _deltaMpMax;
```

If a vault has a large MP cap but has not yet fully accrued up to that cap (a common case for recently staked or locked vaults), then `_deltaMpMax > _deltaMpTotal`. If the system is globally saturated at the time of unstake (`totalMPAccrued == totalMaxMP`), this asymmetric reduction results in:

```
totalMPAccrued_new > totalMaxMP_new
```

Once this state is reached, any subsequent call to `_updateGlobalState()` after time has advanced will revert. Specifically, `_totalMP()` attempts to clamp accrual using:

```
if (totalMPAccrued + accruedMP > totalMaxMP) {
    accruedMP = totalMaxMP - totalMPAccrued;
}
```

When `totalMPAccrued > totalMaxMP`, the subtraction underflows and reverts in Solidity 0.8.x. Because `_updateGlobalState()` is invoked by most user-facing flows (`stake`, `unstake`, `lock`, `redeemRewards`, `updateVault`, etc.), this leads to a protocol-level denial of service where staking and reward interactions become permanently unusable until state is corrected.

No malicious behavior is required. A normal sequence of staking, time-based MP accrual reaching the global cap, followed by unstaking from a not-yet-saturated vault is sufficient to trigger the issue.

Impact: After the invariant `totalMPAccrued > totalMaxMP` is reached, all operations that call `_updateGlobalState()` revert once time advances. This results in a permanent denial of service affecting staking, unstaking, vault updates, and reward distribution.

Proof of Concept:

```
// SPDX-License-Identifier: MIT
pragma solidity 0.8.26;

import "forge-std/Test.sol";
import "forge-std/StdError.sol";

import { StakeManager } from "../src/StakeManager.sol";
import { StakeVault } from "../src/StakeVault.sol";
import { VaultFactory } from "../src/VaultFactory.sol";
import { MockToken } from "./mocks/MockToken.sol";

import { ERC1967Proxy } from "@openzeppelin/contracts/proxy/ERC1967/ERC1967Proxy.sol";
import { Clones } from "@openzeppelin/contracts/proxy/Clones.sol";

contract StakeManagerDoSTest is Test {
    StakeManager public stakeManager;
    StakeVault public stakeVaultImpl;
    VaultFactory public vaultFactory;

    MockToken public snt;
    MockToken public karma;

    address public admin = address(0x1);
    address public userA = address(0x2);
    address public userB = address(0x3);

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

        snt = new MockToken("SNT", "SNT");
        karma = new MockToken("KARMA", "KARMA");

        StakeManager impl = new StakeManager();
        ERC1967Proxy proxy = new ERC1967Proxy(
            address(impl),
            abi.encodeWithSelector(StakeManager.initialize.selector, admin, address(snt),
            ↪  address(karma))
        );
        stakeManager = StakeManager(address(proxy));

        stakeVaultImpl = new StakeVault(snt);
        vaultFactory = new VaultFactory(admin, address(stakeManager), address(stakeVaultImpl));

        // Whitelist codehash for cloned vaults
        address manualClone = Clones.clone(address(stakeVaultImpl));
        stakeManager.setTrustedCodehash(manualClone.codehash, true);

        vm.stopPrank();
    }

    function _warpAndUpdateGlobal(uint256 secondsForward) internal {
        vm.warp(block.timestamp + secondsForward);
        stakeManager.updateGlobalState();
    }

    function test_totalMPAccrued_can_exceed_totalMaxMP_causing_updateGlobalState_DoS() public {
        // 1) User B stakes first and becomes fully saturated (hits their MP cap)
        vm.startPrank(userB);
        StakeVault vaultB = vaultFactory.createVault();

        snt.mint(userB, 1000 ether);
        snt.approve(address(vaultB), 1000 ether);

        vaultB.stake(100 ether, 90 days);
        vm.stopPrank();

        // Warp far enough for B to saturate, then materialize B's MP into vault + globals
        vm.warp(block.timestamp + 4 * 365 days + 1 days);
        stakeManager.updateVault(address(vaultB));

        // 2) User A stakes later (so A has significant remaining MP gap while global accrual can still
        ↪   run)
        vm.startPrank(userA);
        StakeVault vaultA = vaultFactory.createVault();

        snt.mint(userA, 1000 ether);
        snt.approve(address(vaultA), 1000 ether);

        vaultA.stake(100 ether, 90 days);
        vm.stopPrank();

        // 3) Advance time until the global MP reaches the global cap (totalMPAccrued == totalMaxMP)
        // We do this in chunks to avoid relying on one magic timestamp.
        // Important: updateGlobalState() moves lastMPUpdatedTime forward each time.
        for (uint256 i = 0; i < 30; i++) {
            _warpAndUpdateGlobal(60 days);

            // Sanity: accrued should never exceed max in the "healthy" path
            assertLe(stakeManager.totalMPAccrued(), stakeManager.totalMaxMP(), "accrued should not
            ↪   exceed max before unstake");

            if (stakeManager.totalMPAccrued() == stakeManager.totalMaxMP()) {
                break;
            }
        }

        // Ensure we actually reached the cap; otherwise the repro won't work.
        assertEq(
            stakeManager.totalMPAccrued(),
            stakeManager.totalMaxMP(),
            "global should be capped before triggering the divergence via unstake"
        );

        // 4) Now unstake A entirely.
        // If A is not fully saturated locally, we expect:
        // - totalMaxMP decreases by deltaMax (big)
        // - totalMPAccrued decreases by deltaTotal (smaller)
        // Starting from totalMPAccrued == totalMaxMP, this can produce totalMPAccrued > totalMaxMP.
        vm.startPrank(userA);
        vaultA.unstake(100 ether);
        vm.stopPrank();

        // Prove the invariant is now broken: accrued exceeds max (this is the dangerous state)
        assertGt(
            stakeManager.totalMPAccrued(),
            stakeManager.totalMaxMP(),
            "post-unstake accrued should exceed max to trigger the underflow DoS"
        );

        // 5) Once time advances, _totalMP() will enter the cap branch and attempt:
        // accruedMP = totalMaxMP - totalMPAccrued; which underflows and reverts in Solidity 0.8.x.
        vm.warp(block.timestamp + 1);

        vm.expectRevert(stdError.arithmeticError);
        stakeManager.updateGlobalState();
    }
}
```

- Output:

```
Ran 1 test for test/StakeManagerDoS.t.sol:StakeManagerDoSTest
[PASS] test_totalMPAccrued_can_exceed_totalMaxMP_causing_updateGlobalState_DoS() (gas: 1094540)
Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 4.34ms (1.15ms CPU time)

Ran 1 test suite in 147.18ms (4.34ms CPU time): 1 tests passed, 0 failed, 0 skipped (1 total tests)
```

Recommended Mitigation: Defensively enforce the global MP cap invariant inside `_totalMP()` by handling the case where `totalMPAccrued >= totalMaxMP` before performing any subtraction. A minimal fix is to clamp early:

```
if (totalMPAccrued >= totalMaxMP) {
    return totalMaxMP;
}
```

Then compute remaining capacity safely:

```
uint256 remaining = totalMaxMP - totalMPAccrued;
if (accruedMP > remaining) {
    accruedMP = remaining;
}
```

StatusL2: Fixed in 56a7b64.

Cyfrin: Verified.

7.3 Medium Risk

7.3.1 `ERC20VotesUpgradeable::getPastTotalSupply` overestimates the value

Description: Karma token is used for voting. `ERC20VotesUpgradeable` keeps track of checkpoints: 1) balance of user at certain block, 2) `totalSupply` at certain block. Already minted but not yet distributed tokens are excluded from accounting in both `balanceOf` and `totalSupply` in `Karma.sol`:

```
    function balanceOf(address account) public view override returns (uint256) {
        if (rewardDistributors.contains(account)) {
            return 0;
        }
        return _balanceOf(account);
    }

    function totalSupply() public view override returns (uint256) {
        uint256 externalSupply = 0;
        uint256 totalDistributorBalance = 0;

        for (uint256 i = 0; i < rewardDistributors.length(); i++) {
            IRewardDistributor distributor = IRewardDistributor(rewardDistributors.at(i));
            externalSupply += distributor.totalRewardsSupply();
            totalDistributorBalance += super.balanceOf(address(distributor));
        }

        if (externalSupply > totalDistributorBalance) {
            externalSupply = totalDistributorBalance;
        }

        // subtract the distributor balances to avoid double counting
        return super.totalSupply() - totalDistributorBalance + externalSupply;
    }
```

However internal checkpoint accounting in `ERC20VotesUpgradeable.sol` doesn't use updated `Karma::totalSupply`, it simply increases by amount of `mint` and `burn`:

```
    function _mint(address account, uint256 amount) internal virtual override {
        super._mint(account, amount);
        require(totalSupply() <= _maxSupply(), "ERC20Votes: total supply risks overflowing votes");

        _writeCheckpoint(_totalSupplyCheckpoints, _add, amount);
    }

    /**
     * @dev Snapshots the totalSupply after it has been decreased.
     */
    function _burn(address account, uint256 amount) internal virtual override {
        super._burn(account, amount);

        _writeCheckpoint(_totalSupplyCheckpoints, _subtract, amount);
    }
```

Impact: Total supply is overestimated by not yet distributed tokens. Usually total supply is used for quorum during voting, in this case quorum value is incorrect

Recommended Mitigation: TBD. Likely, mint and burn functions should not update `totalSupply` if that's reward distributor, rather `totalSupply` should be updated when rewards are distributed.

StatusL2: Acknowledged, as the team are not planning to rely on 'getPastTotalSupply'.

7.3.2 Users can be overslashed in `Karma.sol`

Description: `Karma::_calculateSlashAmount` increases slashed amount to `MIN_SLASH_AMOUNT = 1e18` or even full balance:

```
    function _calculateSlashAmount(uint256 balance) internal view returns (uint256) {
        uint256 amountToSlash = Math.mulDiv(balance, slashPercentage, MAX_SLASH_PERCENTAGE);
        if (amountToSlash < MIN_SLASH_AMOUNT) {
            if (balance < MIN_SLASH_AMOUNT) {
                // Not enough balance for minimum slash, slash entire balance
                amountToSlash = balance;
            } else {
                amountToSlash = MIN_SLASH_AMOUNT;
            }
        }
        return amountToSlash;
    }
```

Problem is that such rounding is applied multiple times in the same action

```
    function _slash(address account, address rewardRecipient) internal virtual returns (uint256) {
        ...

        // first, calculate the total amount to slash from the actual reward tokens
@>      uint256 totalAmountToSlash = _calculateSlashAmount(super.balanceOf(account));

        for (uint256 i = 0; i < rewardDistributors.length(); i++) {
            address distributor = rewardDistributors.at(i);
            uint256 currentDistributorAccountBalance =
            ↪  IRewardDistributor(distributor).rewardsBalanceOfAccount(account);

            // then, calculate the amount to slash from each reward distributor
@>          totalAmountToSlash += _calculateSlashAmount(currentDistributorAccountBalance);

            // turn virtual Karma into real Karma for slashing
            IRewardDistributor(distributor).redeemRewards(account);
        }

        ...

        // Burn the entire slashed amount from the account
        _burn(account, totalAmountToSlash);

        ...
    }
```

Suppose following scenario:

1) Current balance is `0.9e18`
2) Virtual balances in RewardDistributors are `0.8e18` and `0.7e18`
3) Finally it will slash full `2.4e18` because every amount is rounded up. Actually it should slash only 50%, i.e. `1.2e18`

Impact: Users are overslashed on low balance

Recommended Mitigation: Apply min amount calculation only once on final amount in `Karma::_slash`.

StatusL2: Fixed in cc00600.

Cyfrin: Verified.

7.3.3 Commit-reveal does not sufficiently protect against slash frontrunnings

Description: A commit-reveal scheme in `RLN` to battle the case where a slasher frontruns another by using the provided private key, but setting up his own reward recipient address. It works in the following 2-step way:

1. Commit a hash.
2. Reveal the private key and rewards address corresponding to that hash. In step 2, the actual slash happens.

However, this protection is insufficient as instead of the usual case where the attacker would frontrun the slash directly, he can simply frontrun the reveal by seeing the private key then.

Impact: Frontrunning protection is insufficient.

Recommended Mitigation: Make `slash()` internal so only commit-reveal way of slashing is available. The other option is to only make `slash()` available to call when no commit slash has been made.

StatusL2: Fixed in 0644175.

Cyfrin: Verified.

7.3.4 `Karma.sol` doesn't initialize signature variables

Description: Karma.sol only calls following initializers:

```
contract Karma is Initializable, ERC20VotesUpgradeable, UUPSUpgradeable, AccessControlUpgradeable {

    function initialize(address _owner) public initializer {
        ...
        __ERC20_init(NAME, SYMBOL);
        __ERC20Votes_init();
        __UUPSUpgradeable_init();
        __AccessControl_init();

        ...
    }
```

However `ERC20VotesUpgradeable` inherits `ERC20PermitUpgradeable`:

```
abstract contract ERC20VotesUpgradeable is Initializable, IVotesUpgradeable, ERC20PermitUpgradeable {
```

```
    /**
     * @dev Initializes the {EIP712} domain separator using the `name` parameter, and setting `version`
     ↪   to `"1"`.
     *
     * It's a good idea to use the same `name` that is defined as the ERC20 token name.
     */
    function __ERC20Permit_init(string memory name) internal onlyInitializing {
        __EIP712_init_unchained(name, "1");
    }
```

Impact: It means `Karma.sol` uses `name = ""` and `version = 0` for EIP712 signature, which makes it incompatible with digital wallets. I.e. function `ERC20VotesUpgradeable::delegateBySig` can't be used.

Recommended Mitigation: Initialize ERC20Permit:

```
    function initialize(address _owner) public initializer {
        ...
        __ERC20_init(NAME, SYMBOL);
+       __ERC20Permit_init(NAME);
        __ERC20Votes_init();
        __UUPSUpgradeable_init();
        __AccessControl_init();

        ...
    }
```

StatusL2: Fixed in 7d447fd.

Cyfrin: Verified.

7.3.5 Griefer can block user from claiming airdrop

Description: Function `KarmaAirdrop::claim` claims airdrop and delegates voting power via a call to `Erc20Votes::delegateBySig`:

```
    function claim(
        uint256 index,
        address account,
        uint256 amount,
        bytes32[] calldata merkleProof,
        uint256 nonce,
        uint256 expiry,
        uint8 v,
        bytes32 r,
        bytes32 s
    )
        external
    {
        if (merkleRoot == bytes32(0)) {
            revert KarmaAirdrop__MerkleRootNotSet();
        }
        if (isClaimed(index)) {
            revert KarmaAirdrop__AlreadyClaimed();
        }

        // Verify the merkle proof.
        /// forge-lint: disable-next-line(asm-keccak256)
        bytes32 node = keccak256(abi.encodePacked(index, account, amount));
        if (!MerkleProof.verify(merkleProof, merkleRoot, node)) {
            revert KarmaAirdrop__InvalidProof();
        }

        // Mark it claimed and send the token.
        _setClaimed(index);
        if (!IERC20(TOKEN).transfer(account, amount)) {
            revert KarmaAirdrop__TransferFailed();
        }

        // If the account has no karma balance before this claim, delegate to the default delegatee
        if (IERC20(TOKEN).balanceOf(account) == amount) {
@>          IVotes(TOKEN).delegateBySig(DEFAULT_DELEGATEE, nonce, expiry, v, r, s);
        }
    }
```

There is known vulnerability with EIP2612 permit: signature can be observed in mempool and executed separately, making original call revert. Suppose following example:

1) User sends transaction to mempool, there is no way to not use signature during first claim
2) Griefer observes mempool, notices signature
3) Griefer frontruns by executing `delegateBySig` separately
4) Now claim transaction is executed. Nonce is already used, so signature is invalid and therefore transaction reverts:

```
    function delegateBySig(
        address delegatee,
        uint256 nonce,
        uint256 expiry,
        uint8 v,
        bytes32 r,
        bytes32 s
    ) public virtual {
        if (block.timestamp > expiry) {
            revert VotesExpiredSignature(expiry);
        }
        address signer = ECDSA.recover(
            _hashTypedDataV4(keccak256(abi.encode(DELEGATION_TYPEHASH, delegatee, nonce, expiry))),
            v,
            r,
            s
        );
@>      _useCheckedNonce(signer, nonce);
        _delegate(signer, delegatee);
    }

    function _useCheckedNonce(address owner, uint256 nonce) internal virtual {
        uint256 current = _useNonce(owner);
        if (nonce != current) {
@>          revert InvalidAccountNonce(owner, current);
        }
    }
```

Impact: User can be blocked from claiming airdrop.

Recommended Mitigation: Wrap call to `Erc20VotesUpgradeable::delegateBySig` into try-catch and in catch ensure there is already sufficient allowance.

StatusL2: Fixed in f9b97ab.

Cyfrin: Verified.

7.3.6 User can lose accrued rewards during migration

Description: User can migrate his staking position to new vault. However it will overwrite his `rewardsAccrued` in that new vault:

```
    function migrateToVault(address migrateTo)
        external
        onlyNotEmergencyMode
        whenNotPaused
        onlyTrustedCodehash
        onlyRegisteredVault
    {
        if (vaultOwners[migrateTo] == address(0)) {
            revert StakeManager__InvalidVault();
        }

        if (vaultData[migrateTo].stakedBalance > 0) {
            revert StakeManager__MigrationTargetHasFunds();
        }

        _updateGlobalState();
        _updateVault(msg.sender, false);

        VaultData storage oldVault = vaultData[msg.sender];
        VaultData storage newVault = vaultData[migrateTo];

        // migrate vault data to new vault
        newVault.stakedBalance = oldVault.stakedBalance;
        newVault.rewardIndex = oldVault.rewardIndex;
        newVault.mpAccrued = oldVault.mpAccrued;
        newVault.maxMP = oldVault.maxMP;
        newVault.lastMPUpdateTime = oldVault.lastMPUpdateTime;
@>      newVault.rewardsAccrued = oldVault.rewardsAccrued;

        IStakeVault.MigrationData memory migrationData = IStakeVault.MigrationData({
            lockUntil: IStakeVault(msg.sender).lockUntil(),
            depositedBalance: IStakeVault(msg.sender).depositedBalance()
        });

        IStakeVault(migrateTo).migrateFromVault(migrationData);

        delete vaultData[msg.sender];

        emit VaultMigrated(msg.sender, migrateTo);
    }
```

Consider following example:

1) User has staking in old vault
2) New vault version is released
3) User creates stake in new vault
4) Unstakes from new vault
5) Migrates stake from old vault to the new one
6) User loses unclaimed rewards in that new vault, which were earned between steps 3 and 4.

Impact: User can lose accrued rewards during migration

Recommended Mitigation: Ensure user migrates to empty vault:

```
    function migrateToVault(address migrateTo)
        external
        onlyNotEmergencyMode
        whenNotPaused
        onlyTrustedCodehash
        onlyRegisteredVault
    {
        if (vaultOwners[migrateTo] == address(0)) {
            revert StakeManager__InvalidVault();
        }

-       if (vaultData[migrateTo].stakedBalance > 0) {
+       if (vaultData[migrateTo].stakedBalance > 0 || vaultData[migrateTo].rewardsAccrued > 0) {
            revert StakeManager__MigrationTargetHasFunds();
        }

        ...
    }
```

StatusL2: Fixed in e038ece.

Cyfrin: Verified.

7.3.7 `DeployProtocol.s.sol` incorrectly allows transfers to `StakeManager.sol`

Description: `DeployProtocol.s.sol` is used to automatically deploy and configure protocol. Contracts `StakeManager.sol` and `SimpleKarmaDistributor.sol` should be allowed to transfer Karma token. However it provides `stakeManager` both times:

```
        // whitelist reward distributors for transferring Karma tokens
        karma.setAllowedToTransfer(address(stakeManager), true);
        console.log("Whitelisted reward distributor (StakeManager)", address(stakeManager), "for
        ↪   transfer");
@>      karma.setAllowedToTransfer(address(stakeManager), true);
        console.log(
            "Whitelisted reward distributor (SimpleKarmaDistributor)", address(simpleKarmaDistributor),
            ↪   "for transfer"
        );
```

Impact: `SimpleKarmaDistributor::redeemRewards` will revert. It means that slashing won't work because `Karma::_slash` iterates over all reward distributors:

```
    function _slash(address account, address rewardRecipient) internal virtual returns (uint256) {
        uint256 currentBalance = _balanceOf(account);
        if (currentBalance == 0) {
            revert Karma__CannotSlashZeroBalance();
        }

        // first, calculate the total amount to slash from the actual reward tokens
        uint256 totalAmountToSlash = _calculateSlashAmount(super.balanceOf(account));

        for (uint256 i = 0; i < rewardDistributors.length(); i++) {
            address distributor = rewardDistributors.at(i);
            uint256 currentDistributorAccountBalance =
            ↪   IRewardDistributor(distributor).rewardsBalanceOfAccount(account);

            // then, calculate the amount to slash from each reward distributor
            totalAmountToSlash += _calculateSlashAmount(currentDistributorAccountBalance);

            // turn virtual Karma into real Karma for slashing
@>          IRewardDistributor(distributor).redeemRewards(account);
        }
```

Recommended Mitigation:

```
        // whitelist reward distributors for transferring Karma tokens
        karma.setAllowedToTransfer(address(stakeManager), true);
        console.log("Whitelisted reward distributor (StakeManager)", address(stakeManager), "for
        ↪   transfer");
-       karma.setAllowedToTransfer(address(stakeManager), true);
+       karma.setAllowedToTransfer(address(simpleKarmaDistributor), true);
        console.log(
            "Whitelisted reward distributor (SimpleKarmaDistributor)", address(simpleKarmaDistributor),
            ↪   "for transfer"
        );
```

StatusL2: Fixed in 025f790.

Cyfrin: Verified.

7.3.8 Slashing doesn't work if any reward distributor is paused

Description: `Karma::_slash` iterates over all reward distributors and claims rewards by calling `IRewardDistributor::redeemRewards`:

```
    function _slash(address account, address rewardRecipient) internal virtual returns (uint256) {
        ...

        for (uint256 i = 0; i < rewardDistributors.length(); i++) {
            address distributor = rewardDistributors.at(i);
            uint256 currentDistributorAccountBalance =
            ↪  IRewardDistributor(distributor).rewardsBalanceOfAccount(account);

            // then, calculate the amount to slash from each reward distributor
            totalAmountToSlash += _calculateSlashAmount(currentDistributorAccountBalance);

            // turn virtual Karma into real Karma for slashing
@>          IRewardDistributor(distributor).redeemRewards(account);
        }

        ...
    }
```

Problem is that `StakeManager::redeemRewards` is pausable:

```
    function redeemRewards(address account) external onlyNotEmergencyMode whenNotPaused returns
    ↪  (uint256) {
```

So users can't be slashed if any reward distributor is paused.

Impact: Slashing doesn't work if any reward distributor is paused

Recommended Mitigation: Consider adding function `isPaused() returns (bool)` to `interface IRewardDistributor` and skip paused during slashing.

StatusL2: Fixed in 99b73b0.

Cyfrin: Verified.

7.4 Low Risk

7.4.1 Unstaking rounds in favor of users

Description: Unstaking via `StakeManager` computes the delta in the MP values based on the % unstaked by the user out of the total staked by him, using `mulDiv` which rounds down. Then, these values are used for subtracting. As higher MP values benefit the user, then subtracting less benefits him, thus the roundings were in favor of him.

Impact: Wrong roundings favor the user. The impact is not significant, but better to fix it.

Recommended Mitigation: Round up instead.

StatusL2: Fixed in ff85b3a.

Cyfrin: Verified.

7.4.2 Migrations to self are possible

Description: Migrations to self are possible (if staked balance == 0). This causes weird state transitions like completely deleting the position due to the `delete vaultData[msg.sender];`. While no serious impact has been found yet (except self-harm, user mistake scenarios), it is advised to disallow that.

Impact: Unintended state transitions.

Recommended Mitigation:

```
function migrateToVault(address migrateTo) external onlyOwner onlyNotLeft {
+   require(migrateTo != address(this));
    // ...
}
```

StatusL2: Fixed in 08ada30.

Cyfrin: Verified.

7.4.3 `totalMPAccrued` accrual should start from first stake

Description: The StakeManager contract initializes `lastMPUpdatedTime` to `block.timestamp` during deployment. This timestamp is used to calculate MP accrual periods in the `StakeManager::_totalMP` function. However, when the first user stakes, `lastMPUpdatedTime` is not updated to the current timestamp due to a logic flaw in `StakeManager::_updateGlobalMP`.

During first stake it doesn't update `lastMPUpdatedTime`, because this line lies under condition `if (newTotalMPAccrued > totalMPAccrued)` which is false because before first stake MP are 0:

```
    function stake(
        uint256 amount,
        uint256 lockPeriod,
        uint256 currentLockUntil
    )
        external
        onlyNotEmergencyMode
        whenNotPaused
        onlyTrustedCodehash
        onlyRegisteredVault
        returns (uint256 newLockUntil)
    {
        if (amount == 0) {
            revert StakeManager__AmountCannotBeZero();
        }

@>      _updateGlobalState();
        _updateVault(msg.sender, true);
        ...
    }

    function _updateGlobalState() internal virtual {
@>      _updateGlobalMP();
        _updateRewardIndex();
    }

    function _updateGlobalMP() internal {
        uint256 newTotalMPAccrued = _totalMP();
@>      if (newTotalMPAccrued > totalMPAccrued) {
            totalMPAccrued = newTotalMPAccrued;
            lastMPUpdatedTime = block.timestamp;
        }
    }
```

Suppose following scenario:

1) Contract is deployed and initialized
2) After 0.5 year there is first stake
3) 1 year passes
4) `totalMPAccrued` will contain extra amount equal to `firstStakeAmount * 0.5 year`

Impact: Turns out variable `totalMPAccrued` is not used on-chain, so there is no meaningful impact.

Proof of Concept: Paste into `status-network-contracts/test/stake-manager/MPAccrualAfterDelayTest.t.sol`, execute with `forge test --match-test test_MPAccrualIncludesTimeBeforeFirstStake -vvv`

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.26;

import { StakeManagerTest } from "./StakeManagerBase.t.sol";
import { MultiplierPointMath } from "../../src/math/MultiplierPointMath.sol";

contract MPAccrualAfterDelayTest is StakeManagerTest {
    function setUp() public virtual override {
        super.setUp();
    }

    /**
     *
     * Scenario:
     * 1. Contract is initialized at T0, lastMPUpdatedTime = T0
     * 2. 1 year passes (no one stakes)
     * 3. At T1 (T0 + 1 year) Alice makes the first stake
     * 4. During first stake, lastMPUpdatedTime does NOT update to T1 (remains T0)
     *    Reason: totalMaxMP == 0, so _totalMP() returns totalMPAccrued (0)
     *            and condition (newTotalMPAccrued > totalMPAccrued) is false
     * 5. Another 1 year passes
     * 6. At T2 (T0 + 2 years) update is called
     * 7. MP accrues for period (T2 - T0) = 2 years instead of (T2 - T1) = 1 year
     */
    function test_CRITICAL_MPAccrualIncludesTimeBeforeFirstStake() public {
        uint256 stakeAmount = 1000e18;
        uint256 YEAR = 365 days;

        uint256 initTime = block.timestamp;

        vm.warp(block.timestamp + YEAR);
        uint256 firstStakeTime = block.timestamp;

        _stake(alice, stakeAmount, 0);
        // lastMPUpdatedTime did not update to first stake time
        assertEq(
            streamer.lastMPUpdatedTime(),
            initTime
        );

        uint256 expectedInitialMP = stakeAmount;

        vm.warp(block.timestamp + YEAR);
        uint256 updateTime = block.timestamp;

        _updateVault(alice);

        // Calculate expected MP
        uint256 mpFor1Year = MultiplierPointMath._accrueMP(stakeAmount, YEAR);
        uint256 expectedTotalMP = expectedInitialMP + mpFor1Year;

        // Calculate actual MP
        uint256 mpFor2Years = MultiplierPointMath._accrueMP(stakeAmount, updateTime - initTime);
        uint256 wrongTotalMP = expectedInitialMP + mpFor2Years;

        // MP accrued for 2 years instead of 1 year
        assertEq(
            streamer.totalMPAccrued(),
            wrongTotalMP,
            "BUG: MP accrued for 2 years (since init) instead of 1 year (since first stake)"
        );

        emit log_named_uint("Expected MP after 1 year", expectedTotalMP);
        emit log_named_uint("Actual MP (wrong)", wrongTotalMP);
        emit log_named_uint("Overcount (MP for extra year)", wrongTotalMP - expectedTotalMP);
        emit log_named_uint("Time since init (2 years)", updateTime - initTime);
        emit log_named_uint("Time since first stake (1 year)", updateTime - firstStakeTime);
    }
}
```

Recommended Mitigation: Always update `lastMPUpdatedTime`:

```
    function _updateGlobalMP() internal {
        uint256 newTotalMPAccrued = _totalMP();
+       lastMPUpdatedTime = block.timestamp;
        if (newTotalMPAccrued > totalMPAccrued) {
            totalMPAccrued = newTotalMPAccrued;
-           lastMPUpdatedTime = block.timestamp;
        }
    }
```

Or use same pattern as in `_updateVault`, i.e. send flag `bool forceMPUpdate`:

```
    function _updateVault(address vaultAddress, bool forceMPUpdate) internal virtual {
        ...
@>      if (accruedMP > 0 || forceMPUpdate) {
            vault.mpAccrued += accruedMP;
@>          vault.lastMPUpdateTime = block.timestamp;
            totalMPStaked += accruedMP;
        }
    }
```

StatusL2: Fixed in 56a7b64.

Cyfrin: Verified.

7.4.4 `totalMaxMP` incorrectly accounts for individual `maxMP`

Description: Each stake has a limit of MP to accrue, accrual must stop after 4 years. For each stake it calculates `maxMP` and simply stops accrual after that value, and it is correct:

```
    function _calculateAccrual(
        uint256 _balance,
        uint256 _currentTotalMP,
        uint256 _currentMaxMP,
        uint256 _lastAccrualTime,
        uint256 _processTime
    )
        internal
        pure
        returns (uint256 _deltaMpTotal)
    {
        uint256 dt = _processTime - _lastAccrualTime;
@>      if (_currentTotalMP < _currentMaxMP) {
@>          _deltaMpTotal = Math.min(_accrueMP(_balance, dt), _currentMaxMP - _currentTotalMP);
        }
    }
```

However such approach is incorrect for `totalMaxMP` value:

```
    function _totalMP() internal view returns (uint256) {
        if (totalMaxMP == 0) {
            return totalMPAccrued;
        }

        uint256 currentTime = block.timestamp;
        uint256 timeDiff = currentTime - lastMPUpdatedTime;
        if (timeDiff == 0) {
            return totalMPAccrued;
        }

@>      uint256 accruedMP = _accrueMP(totalStaked, timeDiff);
@>      if (totalMPAccrued + accruedMP > totalMaxMP) {
@>          accruedMP = totalMaxMP - totalMPAccrued;
        }

        uint256 newTotalMPAccrued = totalMPAccrued + accruedMP;

        return newTotalMPAccrued;
    }
```

That's because total value doesn't know how much from `totalStaked` has already approached the limit, so it uses full amount for accrual. Suppose following scenario:

1) There are 2 stakes of 1000 tokens (suppose both with 4 years lock), i.e. `totalMaxMP = 1000 * 9 * 2 = 18000`. The only difference is that one of them has accrued max MP because lives for 4 years.
2) It means that actually MP grows only based on 1000 staked.
3) But `StakeManager::_totalMP` will accrue MP using `totalStaked = 2000`. And this line won't save, because capacity is more than enough `9000 + 5000 > 18000 ---> false` :

```
        if (totalMPAccrued + accruedMP > totalMaxMP) {
            accruedMP = totalMaxMP - totalMPAccrued;
        }
```

Impact: `StakeManager::totalMPAccrued` and `StakeManager::totalMP` inflate real values.

Recommended Mitigation: Fix is not trivial.

StatusL2: Fixed in 56a7b64.

Cyfrin: Verified.

7.4.5 New users can't be registered after slashing contrary to documentation

Description: `RLN.sol` has variable `SET_SIZE`, which defines maximum number of registered users. During slashing, users are deleted from registry:

```
    function slash(bytes32 privateKey, address rewardRecipient) private onlyRole(SLASHER_ROLE) {
        // Hash the private key using Poseidon to get identityCommitment
        uint256 identityCommitment = poseidonHasher.hash(uint256(privateKey));

        User memory member = members[identityCommitment];
        if (member.userAddress == address(0)) {
            revert RLN__MemberNotFound();
        }
        karma.slash(member.userAddress, rewardRecipient);
@>      delete members[identityCommitment];
    }
```

According to documentation `https://github.com/status-im/status-network-monorepo/blob/develop/status-network-contracts/docs/rln.md#registry-capacity`:

> Once the registry reaches capacity, new registrations are rejected until space is available. When accounts are slashed, their identity commitments are removed from the registry, freeing up space for new registrations.

However slashing accounts doesn't free up space for new registrations:

```
    function register(uint256 identityCommitment, address user) external onlyRole(REGISTER_ROLE) {
@>      uint256 index = identityCommitmentIndex;
@>      if (index >= SET_SIZE) {
            revert RLN__SetIsFull();
        }
        if (members[identityCommitment].userAddress != address(0)) {
            revert RLN__IdCommitmentAlreadyRegistered();
        }

        /// forge-lint: disable-next-line(named-struct-fields)
        members[identityCommitment] = User(user, index);
        emit MemberRegistered(identityCommitment, index);

        unchecked {
@>          identityCommitmentIndex = index + 1;
        }
    }
```

Impact: New users can't be registered after slashing contrary to documentation.

Recommended Mitigation: Add missing feature or update documentation.

StatusL2: Fixed in 2c73d4d.

Cyfrin: Verified.

7.4.6 `KarmaNFT.sol` incorrectly simulates minting event

Description: It emits Transfer event to simulate minting:

```
    /**
     * @notice Emits transfer event to simulate minting an NFT to the caller's address.
     */
    function mint() external {
        emit Transfer(msg.sender, msg.sender, uint256(uint160(msg.sender)));
    }
```

EIP-721 specifies that when NFT is created, it uses `from = address(0)` https://eips.ethereum.org/EIPS/eip-721#specification:

> /// @dev This emits when ownership of any NFT changes by any mechanism. /// This event emits when NFTs are created (`from` == 0) and destroyed /// (`to` == 0). Exception: during contract creation, any number of NFTs /// may be created and assigned without emitting Transfer. At the time of /// any transfer, the approved address for that NFT (if any) is reset to none. event Transfer(address indexed _from, address indexed _to, uint256 indexed _tokenId);

Impact: Incorrect event is emitted.

Recommended Mitigation:

```
    function mint() external {
-       emit Transfer(msg.sender, msg.sender, uint256(uint160(msg.sender)));
+       emit Transfer(address(0), msg.sender, uint256(uint160(msg.sender)));
    }

```

StatusL2: Fixed in 72cd30d.

Cyfrin: Verified.

7.4.7 `InitializeKarmaTiersScript` doesn't account for decimals

Description: Script uses raw token amounts as you can see. However Karma token has 18 decimals, therefore all users will have "legendary" tier.

```
contract InitializeKarmaTiersScript is Script {
    function run() external {
        ...

        // Tier 0: 0 karma = 0 tx (no gasless for users without karma)
        tiers[0] = KarmaTiers.Tier({ minKarma: 0, maxKarma: 1, name: "entry", txPerEpoch: 2 });
        tiers[1] = KarmaTiers.Tier({ minKarma: 2, maxKarma: 49, name: "newbie", txPerEpoch: 6 });
        tiers[2] = KarmaTiers.Tier({ minKarma: 50, maxKarma: 499, name: "basic", txPerEpoch: 16 });
        tiers[3] = KarmaTiers.Tier({ minKarma: 500, maxKarma: 4999, name: "active", txPerEpoch: 96 });
        tiers[4] = KarmaTiers.Tier({ minKarma: 5000, maxKarma: 19_999, name: "regular", txPerEpoch: 480
        ↪  });
        tiers[5] = KarmaTiers.Tier({ minKarma: 20_000, maxKarma: 99_999, name: "power", txPerEpoch: 960
        ↪  });
        tiers[6] = KarmaTiers.Tier({ minKarma: 100_000, maxKarma: 499_999, name: "pro", txPerEpoch:
        ↪  10_080 });
        tiers[7] =
            KarmaTiers.Tier({ minKarma: 500_000, maxKarma: 4_999_999, name: "high-throughput",
            ↪  txPerEpoch: 108_000 });
        tiers[8] = KarmaTiers.Tier({ minKarma: 5_000_000, maxKarma: 9_999_999, name: "s-tier",
        ↪  txPerEpoch: 240_000 });
        tiers[9] = KarmaTiers.Tier({
            minKarma: 10_000_000, maxKarma: type(uint256).max, name: "legendary", txPerEpoch: 480_000
        });

        ...
    }
}
```

Impact: `KarmaTiers.sol` is initialized with incorrect amounts.

Recommended Mitigation: Add e18 notation to token amounts.

StatusL2: Fixed in fa6a44e.

Cyfrin: Verified.

7.4.8 Potential Divergence Between Off-Chain and On-Chain Poseidon Hashing Causes Unslashable RLN Identities

Description: The `PoseidonHasher` contract initializes the Poseidon state using a raw EVM addition:

```
let s1 := add(input, C1)
```

The Yul `add` instruction wraps modulo `2^256`, not modulo the BN254 field prime `Q`. As a result, the effective input absorbed into the Poseidon permutation becomes:

```
s1_effective = ((input + C1) mod 2^256) mod Q
```

However, canonical Poseidon implementations used in RLN circuits and off-chain tooling interpret inputs as field elements:

```
s1_canonical = ( (input mod Q) + C1 ) mod Q = (input + C1) mod Q
```

These two differ whenever `input + C1  2^256`. This only occurs for inputs in the range:

```
input  [2^256  C1, 2^256)
```

If off-chain code uses canonical field arithmetic (as typical Poseidon libraries do) while the on-chain hasher uses this wraparound behavior, the two commitments diverge.

In the StatusL2 RLN implementation, this inconsistency affects slashing:

- A user registers with an identity commitment computed off-chain.
- During `slash`, the contract recomputes the commitment on-chain using this hasher.
- If the two computations do not match, `slash` reverts with `RLN__MemberNotFound()`, making that identity unslashable.

This only becomes reachable if RLN private keys are treated as arbitrary `bytes32` rather than constrained to the BN254 scalar field (`< Q`). If the protocol correctly samples keys in `Fr`, the overflow path is unreachable.

Impact: If the wider system allows arbitrary 32-byte private keys, approximately:

```
C1 / 2^256  4.7 × 10  (0.0047%, ~1 in 21,000 keys)
```

will fall into the problematic range.

For these identities:

- The off-chain commitment (computed with standard Poseidon over `Fr`) the on-chain commitment computed by `PoseidonHasher`.

Consequences for the RLN module:

- Such members can register normally.
- When misbehaving, `slash(privateKey)` **fails**, making them **effectively unslashable**.
- This weakens the RLN spam-resistance guarantees.
- An attacker could deliberately search for such a key if key generation is not constrained to `< Q`.

If private keys are correctly restricted to the BN254 field (`< Q`), then `(input + C1) < 2Q < 2^256` holds, preventing wraparound entirely; the issue becomes theoretical only.

Proof of Concept:

```
// SPDX-License-Identifier: MIT
pragma solidity 0.8.26;

import { Test } from "forge-std/Test.sol";
import { PoseidonHasher } from "../src/rln/PoseidonHasher.sol";

contract PoseidonHasherTest is Test {
    PoseidonHasher public hasher;

    function setUp() public {
        hasher = new PoseidonHasher();
    }

    function test_PoCHashMaxUint() public {
        uint256 input = type(uint256).max;
        // Expected hash from circomlibjs (which handles field arithmetic correctly)
        uint256 expected =
        ↪  11254588113248280256028662529799552354366536761492627237202955510067774853962;
        uint256 result = hasher.hash(input);
        assertNotEq(result, expected, "Hash mismatch for max uint256 (Overflow issue)");
    }
}
```

- Output:

```
Ran 1 test for test/PoseidonHasherAudit.t.sol:PoseidonHasherTest
[PASS] test_PoCHashMaxUint() (gas: 19607)
Traces:
  [19607] PoseidonHasherTest::test_PoCHashMaxUint()
     [11098] PoseidonHasher::hash(1157920892373161954235709850086879078532699846656405640394575840079131
    ↪  129639935 [1.157e77]) [staticcall]
        [Return] 3366645945435192953002076803303112651887535928162668198103357554665518664470
    ↪  [3.366e75]
     [0] VM::assertNotEq(3366645945435192953002076803303112651887535928162668198103357554665518664470
    ↪  [3.366e75], 11254588113248280256028662529799552354366536761492627237202955510067774853962
    ↪  [1.125e76], "Hash mismatch for max uint256 (Overflow issue)") [staticcall]
       ← [Return]
     ← [Stop]

Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 456.63µs (161.25µs CPU time)
```

Recommended Mitigation: One of the following approaches should be adopted (depending on intended protocol design):

1. **If the intended domain is Fr (recommended for RLN):**
    - Enforce `privateKey < Q` in the off-chain SDK and/or circuits.
    - Document that PoseidonHasher expects field elements, not arbitrary uint256 values.
    - This guarantees `(input + C1) < 2^256`, eliminating wraparound altogether.
2. **If arbitrary uint256 inputs must be supported:** Add explicit field reduction before absorbing the input:

    ```
      input := mulmod(input, 1, Q)     // input = input mod Q
      let s1 := add(input, C1)
    ```

    This ensures on-chain hashing matches canonical Poseidon semantics.

StatusL2: Fixed in 4ead416.

Cyfrin: Verifed.

7.5 Informational

7.5.1 Incorrect Natspec in `RLN::setSlashRevealWindowTime`

Description: Comment mentions max value of `365 days`, however actual code bounds by `1 days`:

```
    /// @dev Sets the slash reveal window time.
    /// @param _slashRevealWindowTime: new reveal window time in seconds.
@>  /// @notice The window time must be at least 1 second and no more than 365 days.
    ///         A non-zero value is required to ensure the queuing mechanism functions correctly.
    ///         An excessively large value could lock commitments indefinitely.
    function setSlashRevealWindowTime(uint256 _slashRevealWindowTime) external
    ↪   onlyRole(DEFAULT_ADMIN_ROLE) {
@>      if (_slashRevealWindowTime == 0 || _slashRevealWindowTime > 1 days) {
            revert RLN__InvalidSlashRevealWindowTime(_slashRevealWindowTime);
        }

        slashRevealWindowTime = _slashRevealWindowTime;
    }
```

Recommended Mitigation: Update Natspec

StatusL2: Fixed in 1f8414d.

Cyfrin: Verified.

7.5.2 Inconsistent naming in `Karma._onlySlasher`

Description: `_onlySlasher` allows Admin to perform an operation same as `_onlyAdminOrOperator`, however you don't include Admin to naming:

```
    function _onlyAdminOrOperator(address sender) internal view {
        if (!hasRole(DEFAULT_ADMIN_ROLE, sender) && !hasRole(OPERATOR_ROLE, sender)) {
            revert Karma__Unauthorized();
        }
    }

    //@audit INFO. Inconsistent naming
    function _onlySlasher(address sender) internal view {
        if (!hasRole(DEFAULT_ADMIN_ROLE, sender) && !hasRole(SLASHER_ROLE, sender)) {
            revert Karma__Unauthorized();
        }
    }
```

Recommended Mitigation:

```
-   modifier onlySlasher() {
+   modifier onlyAdminOrSlasher() {
-       _onlySlasher(msg.sender);
+       _onlyAdminOrSlasher(msg.sender);
        _;
    }

-   function _onlySlasher(address sender) internal view {
+   function _onlyAdminOrSlasher(address sender) internal view {
        if (!hasRole(DEFAULT_ADMIN_ROLE, sender) && !hasRole(SLASHER_ROLE, sender)) {
            revert Karma__Unauthorized();
        }
    }
```

StatusL2: Fixed in 030efdd.

Cyfrin: Verified.

7.5.3 `KarmaTiers.sol` constructor can be simplified

Description: It sets owner:

```
    constructor() {
        transferOwnership(msg.sender);
    }
```

However owner is already set in `Ownable.sol`:

```
    constructor() {
        _transferOwnership(_msgSender());
    }
```

Same logic exists in other 2 constructors in `BaseNFTMetadataGenerator.sol` and `KarmaNFT.sol`

Recommended Mitigation: Remove `transferOwnership(msg.sender)` from constructor in `KarmaTiers.sol`, `KarmaNFT.sol`, `BaseNFTMetadataGenerator.sol`.

StatusL2: Fixed in 606e3d1.

Cyfrin: Verified.

7.5.4 Consider removing unused functions in libraries

Description: Functions `StakeMath::_estimateLockTime`, `MultiplierPointMath::_lockTimeAvailable`, `MultiplierPointMath::_timeToAccrueMP`, `MultiplierPointMath::_retrieveBonusMP`, `MultiplierPointMath::_retrieveAccruedMP` are internal in libraries and never used in protocol.

Recommended Mitigation: Consider removing them.

StatusL2: Acknowledged, the team stated that they will be implementing views functions for the UI.

7.5.5 Consider simplifying formulas in `MultiplierPointMath.sol`

Description: For some reason `MultiplierPointMath.sol` uses 100 in formulas instead of simplifying to this:

```
-   uint256 public constant MP_APY = 100;

    function _accrueMP(uint256 _balance, uint256 _deltaTime) internal pure returns (uint256 accruedMP) {
-       return Math.mulDiv(_balance, _deltaTime * MP_APY, YEAR * 100);
+       return Math.mulDiv(_balance, _deltaTime, YEAR);
    }
```

Expected `_balance` variable has 1e18 precision, so multiplying with 100 doesn't change anything. Removing it from calculations will make code easier to read.

Recommended Mitigation: Consider simplifying formulas in `MultiplierPointMath.sol`.

StatusL2: Fixed in 16c5b3c.

Cyfrin: Verified.

7.5.6 `DeployProtocol.s.sol` doesn't deploy RLN contracts

Description: `DeployProtocol.s.sol` is supposed to automatically deploy all contracts and configure them. However it forgets to invoke script `RLN.s.sol` which deploys contracts `RLN.sol` and `PoseidonHasher.sol`

Recommended Mitigation: Invoke the script `RLN.s.sol` and add contract `RLN.sol` as Slasher to Karma.

StatusL2: Acknowledged, the team stated that the contracts can be deployed separately.

7.5.7 `Karma::removeRewardDistributor` burns all virtual Karma

Description: Admin can remove reward distributor from `Karma.sol`, this function burns full balance:

```
    function _removeRewardDistributor(address distributor) internal virtual {
        if (!rewardDistributors.contains(distributor)) {
            revert Karma__UnknownDistributor();
        }
@>      _burn(distributor, super.balanceOf(distributor));
        rewardDistributors.remove(distributor);
    }
```

Problem is that not all tokens belong to reward distributor, at least part of them is unclaimed rewards belonging to users. Such tokens can be claimed via `redeemRewards` for example in `SimpleKarmaDistributor.sol`:

```
    function redeemRewards(address account) external returns (uint256) {
@>      uint256 amount = balances[account];
        if (amount == 0) {
            return 0;
        }

        balances[account] = 0;
        mintedSupply -= amount;

@>      karmaToken.safeTransfer(account, amount);

        return amount;
    }
```

Recommended Mitigation: `Karma::removeRewardDistributor` is supposed to be used during emergency. In that case distributor can report incorrect virtual Karma, so it's unreliable to leave those virtual karma on balance. That's why it's burned currently.

Consider documenting this behaviour. Additionally if removing burns virtual Karma belonging to users, you should reimburse it to users via `KarmaAirdrop.sol` or direct mint.

StatusL2: Fixed in 19557eb.

Cyfrin: Verified.

7.5.8 Emergency mode doesn't save from malicious upgrade of StakeManager.sol

Description: Documentation describes that one of scenarios when emergency mode will be used is malicious upgrade of `StakingManager.sol` https://github.com/status-im/status-network-monorepo/blob/develop/status-network-contracts/docs/staking-reward-distributor/emergency-mode.md#when-emergency-mode-is-used

> If the StakeManager is upgraded to a malicious or broken implementation, guardians can enable emergency mode to allow users to exit before interacting with the compromised contract.

It's implemented as following:

```
    function emergencyExit(address _destination) external onlyOwner validDestination(_destination) {
        depositedBalance = 0;
        try stakeManager.emergencyModeEnabled() returns (bool enabled) {
            if (!enabled) {
                revert StakeVault__NotAllowedToExit();
            }
            bool success = STAKING_TOKEN.transfer(_destination, STAKING_TOKEN.balanceOf(address(this)));
            if (!success) {
                revert StakeVault__FailedToExit();
            }
        } catch {
            bool success = STAKING_TOKEN.transfer(_destination, STAKING_TOKEN.balanceOf(address(this)));
            if (!success) {
                revert StakeVault__FailedToExit();
            }
        }
    }
```

However if attacker can upgrade `StakeManager.sol`, then he can for example remove function `emergencyModeEnabled` so that try-catch doesn't work, or make `emergencyModeEnabled` return `false`. Yes there is no ability to steal staked tokens, but he still can brick them.

Impact: In case of malicious upgrade of `StakeManager.sol`, attacker can block users from withdrawing staked tokens.

Recommended Mitigation: It's not immediately obvious how to save tokens from bricking in case of `StakeManager.sol` owner compromise in current design. At least update docs to document it.

StatusL2: Fixed in 7401475.

Cyfrin: Verified.

7.5.9 Slashing won't work after enabling emergency mode

Description: Misbehaving users must be slashed via `RLN.sol`, it calls `Karma::slash`. This function iterates over reward distributors and calls `redeemRewards` which is expected to not revert. However `StakeManager::redeemRewards` reverts in emergency mode:

```
    function redeemRewards(address account) external onlyNotEmergencyMode whenNotPaused returns
    ↪   (uint256) {
```

Recommended Mitigation: You should remove reward distributor after enabling it's emergency mode. Then retrieve rewards snapshot and mint lost Karma balances directly to users.

StatusL2: Fixed in a5d51d5.

Cyfrin: Verified.