# Protocol Audit Report

Version 1.0

*Cyfrin.io*

March 23, 2025

# Nudge Protocol Audit Report

Dheeraj Kumar

March 23, 2025

Prepared by: Dheeraj Kumar Lead Auditors:

- Dheeraj Kumar

## Table of Contents

## Protocol Summary

Nudge Protocol is a platform that incentivizes users to reallocate assets within the crypto ecosystem through KPI-driven campaigns. It consists of two main contracts: NudgeFactory, which deploys and manages campaigns, and NudgeCampaign, which handles user participation and reward distribution. The protocol focuses on gas efficiency, programmable incentives, and secure, role-based governance. This audit evaluates its security, gas optimization, and adherence to best practices.

## Disclaimer

Dheeraj Kumar makes all effort to find as many vulnerabilities in the code in the given time period, but holds no responsibilities for the findings provided in this document. A security audit by the team is not an endorsement of the underlying business or product. The audit was time-boxed and the review of the code was solely on the security aspects of the Solidity implementation of the contracts.

## Risk Classification

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

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

## Audit Details

**The findings described in this document corresponds to the following repo:**

https://github.com/code-423n4/2025-03-nudgexyz/tree/main

### Scope

```
./src/
#-- campaign/NudgeCampaign.sol
#-- campaign/NudgeCampaignFactory.sol
#-- campaign/NudgePointsCampaigns.sol
#-- campaign/interfaces/IBaseNudgeCampaign.sol
#-- campaign/interfaces/INudgeCampaign.sol
#-- campaign/interfaces/INudgeCampaignFactory.sol
#-- campaign/interfaces/INudgePointsCampaign.sol
```

### Roles

- NUDGE_ADMIN_ROLE: This role is given to a multisig (Safe) wallet controlled by Nudge.
- DEFAULT_ADMIN_ROLE: Similar to NUDGE_ADMIN_ROLE, initially.
- NUDGE_OPERATOR_ROLE: This role is given to one of our Relayers, submitting transactions programmatically.
- SWAP_CALLER_ROLE: This role is initially given to one of Li.fi's contracts (called Executor).
- CAMPAIGN_ADMIN_ROLE: This role is given to the administrator of a campaign. It is campaign-specific.

## Executive Summary

We spent around 20 hours for this audit.

### Issues found

| Severity | Number of issues found |
|---|---|
| High | 1 |
| Medium | 0 |
| Low | 0 |
| Gas Optimizations | 4 |
| Total number of issues | 5 |

# Findings

## High

### [H-1] Lack of Uniqueness Enforcement for `campaignId`

**Description** The `NudgeCampaign` contract assumes that `campaignId` is a unique identifier for each campaign. However, there is no mechanism in place to enforce this uniqueness. Since the contract is deployed using `CREATE2`, which generates deterministic addresses based on constructor arguments, it is possible to deploy multiple campaigns with the same `campaignId` by simply changing other constructor parameters (e.g., `holdingPeriod`, `startTimestamp`).

```
    /// @param campaignId_ Unique identifier for this campaign
```

This comment suggests that campaignId is intended to be unique, but the contract does not enforce this requirement.

**Impact** Confusion and Misidentification: Multiple campaigns with the same campaignId can exist, leading to confusion for users and front-end applications.

Protocol Integrity: The assumption that campaignId is unique is violated, undermining the integrity of the protocol.

Front-End and Database Challenges: The front-end and database systems would need to rely solely on contract addresses to identify campaigns, which complicates the user experience and data management.

**Proof of Concepts** The following test demonstrates how two campaigns with the same campaignId can be deployed by changing other constructor parameters:

```
        function test_DeployCampaignsWithSameID() public {
        // Deploy first campaign
        address campaign1 = factory.deployCampaign(
            holdingPeriod,
            address(targetToken),
            address(rewardToken),
            REWARD_PPQ,
            campaignAdmin,
            startTimestamp,
            withdrawalAddress,
            uuidOne
        );

        // Deploy second campaign
        address campaign2 = factory.deployCampaign(
            holdingPeriod * 2,
```

```
            address(targetToken),
            address(rewardToken),
            REWARD_PPQ,
            campaignAdmin,
            startTimestamp,
            withdrawalAddress,
            uuidOne
        );

        console.log("Address Campaign 1: ", campaign1);
        console.log("Address Campaign 2: ", campaign2);
    }
```

**Recommended mitigation** To enforce the uniqueness of campaignId and prevent duplicate deployments, implement the following changes:

- Maintain a mapping or set in the NudgeFactory contract to track used campaignIds.
- Revert if a duplicate campaignId is provided during deployment.

```
+   mapping(uint256 => bool) public usedCampaignIds;

    function deployCampaign(uint256 campaignId, ...) external {
+       require(!usedCampaignIds[campaignId], "Campaign ID already used");
+       usedCampaignIds[campaignId] = true;
        // Deploy campaign
        ..
    }
```

## Gas

### [G-1] `NudgeCampaign::feeBps` should be made immutable.

**Description** The storage variable `NudgeCampaign::feeBps` is initialized in the constructor and never modified afterward. However, it is not marked as immutable. Immutable variables are stored in the contract's bytecode rather than in storage, which saves gas by reducing storage reads and writes.

```
    uint16 public feeBps;
```

**Impact** Using a storage variable for a value that never changes after deployment incurs unnecessary gas costs for reading and writing.

**Recommended mitigation** To optimize gas usage, the `feeBps` variable must be marked immutable.

```
-   uint16 public feeBps;
+   uint16 immutable feeBps;
```

### [G-2] Storage variables must be packed efficiently.

**Description** In the `NudgeCampaign.sol` contract, storage variables are not packed efficiently. Specifically, `bool` variables (which occupy 1 byte each) are declared separately, resulting in wasted storage space. Each storage slot consists of 32 bytes, and small variables like bool can be packed together to save gas.

**Impact** Each unused byte in a storage slot still incurs gas costs. Wasting storage space increases deployment and transaction costs.

**Recommended mitigation** To optimize storage usage, the bool variables should be packed together into a single storage slot.

```
    // Fee parameter in basis points (1000 = 10%)
    uint16 public feeBps;
    bool public isCampaignActive;
+   bool private _manuallyDeactivated;

    // Unique identifier for this campaign
    uint256 public immutable campaignId;

    ...

    uint256 public distributedRewards;
    // Track whether campaign was manually deactivated
-   bool private _manuallyDeactivated;
```

### [G-3] Store array length in a local variable for loops

**Description** At multiple instances in the contracts `NudgeCampaign` and `NudgeCampaignFactory`, the code reads the length of an array directly from the function parameter within a for loop. This results in repeated storage or memory reads, which are gas-inefficient. These functions are `claimRewards()` and `invalidateParticipants()` in `NudgeCampaign` and `pauseCampaigns()`, `unpauseCampaigns()` and `collectFeesFromCampaigns()` in `NudgeCampaignFactory`.

```
    function claimRewards(uint256[] calldata pIDs) external
       whenNotPaused {
        ...

        for (uint256 i = 0; i < pIDs.length; i++) {

            ...
        }
    }
```

**Impact** Gas Inefficiency: Reading array.length in each iteration of a loop consumes extra gas, as it involves repeated memory reads.

**Recommended mitigation** To optimize gas usage, the functions should:

1. Store the array length in a local variable before the loop.
2. Use the local variable in the loop condition.

```
    function claimRewards(uint256[] calldata pIDs) external
        whenNotPaused {
        ...
+       uint256 totalpIDs = pIDs.length;
-       for (uint256 i = 0; i < pIDs.length; i++) {
+       for (uint256 i = 0; i < totalpIDs; i++) {
            ...
        }
    }
```

### [G-4] Repeated storage writes in `NudgeCampaign::invalidateParticipations()` and `NudgeCampaign::claimRewards()`

**Description** The functions `NudgeCampaign::invalidateParticipations()` and `NudgeCampaign::claimRewards()` repeatedly write to storage variables (`pendingRewards` and `distributedRewards`) within a for loop. Specifically: - `invalidateParticipations()` updates `pendingRewards` in each iteration. - `claimRewards()` updates both `pendingRewards` and `distributedRewards` in each iteration.

This results in unnecessary gas costs, as each storage write consumes additional gas.

```
    function invalidateParticipations(uint256[] calldata pIDs) external
        onlyNudgeOperator {
        for (uint256 i = 0; i < pIDs.length; i++) {
            Participation storage participation = participations[pIDs[i
                ]];

            if (participation.status != ParticipationStatus.
                PARTICIPATING) {
                continue;
            }

            participation.status = ParticipationStatus.INVALIDATED;
            pendingRewards -= participation.rewardAmount;
        }
        emit ParticipationInvalidated(pIDs);

    }

```

```
16      function claimRewards(uint256[] calldata pIDs) external
            whenNotPaused {
17          if (pIDs.length == 0) {
18              revert EmptyParticipationsArray();
19          }
20
21          uint256 availableBalance = getBalanceOfSelf(rewardToken);
22
23          for (uint256 i = 0; i < pIDs.length; i++) {
24              Participation storage participation = participations[pIDs[i
                    ]];
25              ...
26              uint256 userRewards = participation.rewardAmount;
27              ...
28
29              // Update contract state
30              pendingRewards -= userRewards;
31              distributedRewards += userRewards;
32
33              ...
34
35              emit NudgeRewardClaimed(pIDs[i], participation.userAddress,
                    userRewards);
36          }
37      }
```

**Impact** Each storage write operation costs 100 gas units. Repeated writes within a loop significantly increase gas consumption, especially for large arrays.

**Recommended mitigation** To optimize gas usage, the functions should:

1. Use local variables to accumulate changes during the loop.
2. Perform storage writes only once after the loop completes.

The following changes must be made to the `invalidateParticipants()` and `claimRewards()` functions.

```
1
2       function invalidateParticipations(uint256[] calldata pIDs) external
            onlyNudgeOperator {
3   +       uint256 pendingRewardsToSubtract;
4           for (uint256 i = 0; i < pIDs.length; i++) {
5               Participation storage participation = participations[pIDs[i
                    ]];
6
7               ...
8               participation.status = ParticipationStatus.INVALIDATED;
9   -           pendingRewards -= participation.rewardAmount;
10  +           pendingRewardsToSubtract += participation.rewardAmount;
11          }
```

```
12 +        pendingRewards -= pendingRewardsToSubtract;
13          emit ParticipationInvalidated(pIDs);
14
15      }
16
17      function claimRewards(uint256[] calldata pIDs) external
            whenNotPaused {
18          if (pIDs.length == 0) {
19              revert EmptyParticipationsArray();
20          }
21
22          uint256 availableBalance = getBalanceOfSelf(rewardToken);
23
24 +        uint256 pendingRewardsToSubtract;
25 +        uint256 distributedRewardsToAdd;
26
27          for (uint256 i = 0; i < pIDs.length; i++) {
28              Participation storage participation = participations[pIDs[i
                    ]];
29              ...
30              uint256 userRewards = participation.rewardAmount;
31              ...
32
33              // Update contract state
34 -            pendingRewards -= userRewards;
35 -            distributedRewards += userRewards;
36 +            pendingRewardsToSubtract += userRewards;
37 +            distributedRewardsToAdd += userRewards;
38              ...
39
40              emit NudgeRewardClaimed(pIDs[i], participation.userAddress,
                    userRewards);
41          }
42
43 +        pendingRewards -= pendingRewardsToSubtract;
44 +        distributedRewards += distributedRewardsToAdd;
45      }
```