**Nudge Protocol Audit Audit by SCATERLABs**

---

## 1. Front Running Attack in `handleReallocation`

**Description :**

A front-running attack occurs when an attacker observes a pending transaction and submits a higher gas fee transaction to execute before the original. This can allow them to manipulate token swaps or reward distributions unfairly.

**Severity: Medium**

**Impact**

- Attackers can gain more rewards than intended.
- Users may receive fewer rewards or lose funds.

**Proof of Concept (POC)**

**Vulnerable Code:**

```
function handleReallocation(
    uint256 campaignId_,
    address userAddress,
    address toToken,
    uint256 toAmount,
    bytes memory data
)
    external
    payable
    whenNotPaused
{
    _validateAndActivateCampaignIfReady();
    if (!factory.hasRole(factory.SWAP_CALLER_ROLE(), msg.sender)) {
        revert UnauthorizedSwapCaller();
    }
    if (toToken != targetToken) {
        revert InvalidToTokenReceived(toToken);
    }
    if (campaignId_ != campaignId) {
        revert InvalidCampaignId();
    }

    uint256 amountReceived;
    if (toToken == NATIVE_TOKEN) {
```

```
            amountReceived = msg.value;
        } else {
            IERC20 tokenReceived = IERC20(toToken);
            uint256 balanceBefore = getBalanceOfSelf(toToken);
            SafeERC20.safeTransferFrom(tokenReceived, msg.sender, address(this), toAmount);
            amountReceived = getBalanceOfSelf(toToken) - balanceBefore;
        }

        _transfer(toToken, userAddress, amountReceived);
        totalReallocatedAmount += amountReceived;
        uint256 rewardAmountIncludingFees = getRewardAmountIncludingFees(amountReceived);
        uint256 rewardsAvailable = claimableRewardAmount();
        if (rewardAmountIncludingFees > rewardsAvailable) {
            revert NotEnoughRewardsAvailable();
        }

        (uint256 userRewards, uint256 fees) = calculateUserRewardsAndFees(rewardAmountIncludingF
        pendingRewards += userRewards;
        accumulatedFees += fees;
    }
```

**Exploit Scenario:**

1. User A submits a transaction to reallocate tokens and receive rewards.
2. Attacker observes the transaction in the mempool.
3. Attacker submits a similar transaction with a higher gas fee.
4. Attacker drains the available rewards, leaving User A with nothing.

**Foundry Test (POC):**

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

import "forge-std/Test.sol";
import "src/YourContract.sol";

contract FrontRunningAttackTest is Test {
    YourContract public contractInstance;
    address attacker = address(0x1);
    address victim = address(0x2);

    function setUp() public {
        contractInstance = new YourContract();
    }
```

```
    function testFrontRunning() public {
        vm.startPrank(victim);
        contractInstance.handleReallocation(1, victim, address(0xToken), 100, "");
        vm.stopPrank();

        vm.startPrank(attacker);
        contractInstance.handleReallocation(1, attacker, address(0xToken), 200, "");
        vm.stopPrank();

        uint256 rewards = contractInstance.claimableRewardAmount();
        assertEq(rewards, 0, "Victim received no rewards due to front-running.");
    }
}
```

### Recommended Mitigations

1. Use a commit-reveal scheme: Prevent front-running by making users commit to inputs before
2. Implement gas price caps: Prevent transactions from cutting the queue based on gas price.
3. Batch process transactions: Handle all inputs fairly in a single round.

---

## 2. CREATE2 Address Prediction Attack

### Description

Contracts using `CREATE2` with predictable salts can be precomputed by attackers.
Malicious contracts can be deployed to hijack funds sent to the expected address.

### Severity: High

### Impact

- Attacker receives all initial reward funds.
- Campaign contract deployment fails.
- Total loss of campaign funds.

### Proof of Concept (POC)

### Vulnerable Code:

```
bytes32 salt = keccak256(
    abi.encode(
        holdingPeriodInSeconds,
        targetToken,
        rewardToken,
        rewardPPQ,
```

```
        campaignAdmin,
        startTimestamp,
        FEE_BPS,
        alternativeWithdrawalAddress,
        uuid
    )
);

bytes memory bytecode = abi.encodePacked(type(NudgeCampaign).creationCode, constructorArgs)
campaign = Create2.deploy(0, salt, bytecode);
```

**Exploit Scenario:**

1. Attacker precomputes the deployment address.
2. Attacker deploys malicious contract using the same salt.
3. Funds meant for the legitimate campaign are sent to the malicious contract.
4. Attacker withdraws and self-destructs.

**Foundry Test:**

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.19;
import "forge-std/Test.sol";
import "src/NudgeFactory.sol";
import "src/NudgeCampaign.sol";

contract Create2AttackTest is Test {
    NudgeFactory factory;
    address attacker;

    function setUp() public {
        factory = new NudgeFactory();
        attacker = vm.addr(1);
    }

    function testCreate2Attack() public {
        bytes32 salt = keccak256(abi.encode(3600, address(0x123), address(0x456), 1000, atta
        address predicted = computeCreate2Address(salt, type(NudgeCampaign).creationCode, ad

        vm.prank(attacker);
        new MaliciousContract{salt: salt}();

        vm.expectRevert();
        factory.deployAndFundCampaign(3600, address(0x123), address(0x456), 1000, attacker,
    }
```

```
    function computeCreate2Address(bytes32 salt, bytes memory bytecode, address deployer) in
        return address(uint160(uint256(keccak256(abi.encodePacked(bytes1(0xff), deployer, sa
    }
}

contract MaliciousContract {
    receive() external payable {}
}
```

### Recommended Mitigations

```
1. Use non-predictable salt:
   - Include randomness like blockhash or a nonce.
2. Check if address is already occupied:
   - Use `isContract()` to validate before deployment.

   require(!isContract(predictedAddress), "Address already occupied");


3. Deploy first, fund later:
   - Ensure contract was created successfully before transferring funds.
```

---

## 3. Precision Loss Due to Integer Division

### Description

Integer division in reward calculation can cause truncation, leading to small but cumulative precision losses.

### Vulnerable Code:

```
uint256 finalReward = (rewardAmountIn18Decimals + rewardScalingFactor - 1) / rewardScalingFa
```

### Severity: Medium

### Impact

- Users receive slightly fewer rewards
- Accumulated loss over multiple users
- Inconsistent reward distribution

### Proof of Concept (POC)

```
// SPDX-License-Identifier: MIT
```

```
pragma solidity ^0.8.28;

import "forge-std/Test.sol";

contract PrecisionLossTest is Test {
    function testPrecisionLoss() public {
        uint256 rewardAmountIn18Decimals = 1005;
        uint256 rewardScalingFactor = 100;

        uint256 finalReward = (rewardAmountIn18Decimals + rewardScalingFactor - 1) / rewardS

        assertEq(finalReward, 11, "Precision loss detected"); // Expected 11, may be 10 with
    }
}
```

#### Recommended Mitigations

1. Use higher precision arithmetic:
   - Avoid truncation by multiplying first, dividing later.
2. Use `Math.mulDiv()` from OpenZeppelin:
   - Offers full-precision multiplication followed by division.
3. Use fixed-point math:
   - Store values in 18 decimals and round only when withdrawing.
4. Consider off-chain computation:
   - Move reward logic off-chain and just store outcome on-chain.

---

*Link:* View this audit on GitHub