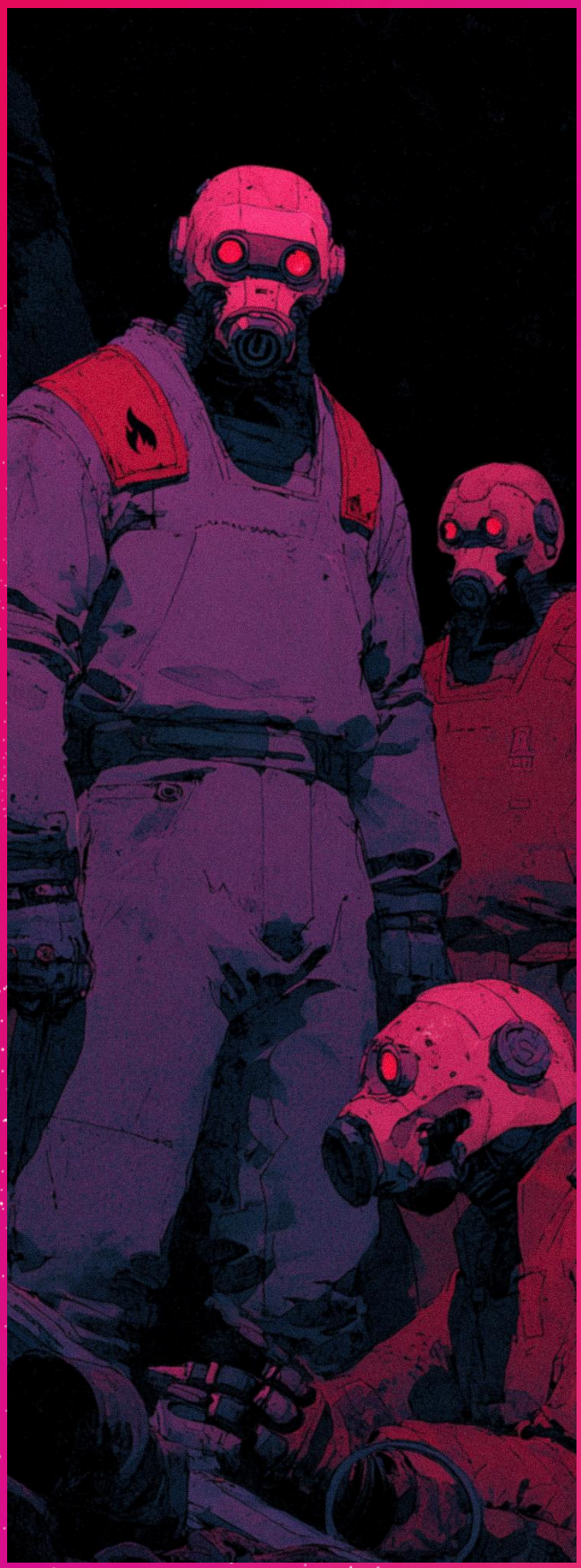
IMMUNEFI AUDIT

Immunefi / MATCHAIN



DATE July 11, 2025

AUDITOR Zealynx, Security Researchers

REPORT BY Immunefi

ABOUT IMMUNEFI

Immunefi is the leading onchain security platform, having directly prevented hacks worth more than $25 billion USD. Immunefi security researchers have earned over $120M USD for responsibly disclosing over 4,000 web2 and web3 vulnerabilities, more than the rest of the industry combined.

Through Magnus, Immunefi delivers a comprehensive suite of best-in-class security services through a single command center to more than 300 projects — including Sky (formerly MakerDAO), Optimism, Polygon, GMX, Reserve, Chainlink, TheGraph, Gnosis Chain, Lido, LayerZero, Arbitrum, StarkNet, EigenLayer, AAVE, ZKsync, Morpho, Ethena, USDT0, Stacks, Babylon, Fuel, Sei, Scroll, XION, Wormhole, Firedancer, Jito, Pyth, Eclipse, PancakeSwap and many more.

Magnus unifies SecOps across the entire onchain lifecycle, combining Immunefi's market leading products and community of elite security researchers with a curated set of the very best security products and technologies provided by top security firms — including Runtime Verification, Dedaub, Fuzzland, Nexus Mutual, Failsafe, OtterSec and others.

Magnus is powered by Immunefi's proprietary vulnerabilities dataset — the largest and most comprehensive in web3, ensuring that security leaders and teams have the best possible tools for identifying and mitigating life threats before they cause catastrophic harm, all while reducing operational overhead and complexity.

Learn how you can benefit too at immunefi.com.

TERMINOLOGY

For the purpose of this assessment, we adopt the following terminology. To classify the severity of our findings, we determine the likelihood and impact (according to the CVSS risk rating methodology).

- **Likelihood** represents the likelihood of a finding to be triggered or exploited in practice
- **Impact** specifies the technical and business-related consequences of a finding
- **Severity** is derived based on the likelihood and the impact

We categorize the findings into four distinct categories, depending on their severity. These severities are derived from the likelihood and the impact using the following table, following a standard risk assessment procedure.

LIKELIHOOD	IMPACT		
	HIGH	MEDIUM	LOW
CRITICAL	Critical	Critical	High
HIGH	High	High	Medium
MEDIUM	Medium	Medium	Low
LOW	Low		
NONE	None		

As seen in the table above, findings that have both a high likelihood and a high impact are classified as critical. Intuitively, such findings are likely to be triggered and cause significant disruption. Overall, the severity correlates with the associated risk. However, every finding's risk should always be closely checked, regardless of severity.

EXECUTIVE SUMMARY

Over the course of 6 days in total, Matchain engaged with Immunefi to review the contracts repository. In this period of time a total of 29 issues were identified.

SUMMARY

Name	Matchain
Repository	https://github.com/matchain/contracts
Audit commit	50690f4ca1cfbaf2a3784eeabe96ee8624e6d6fb
Type of Project	Blockchain
Audit Timeline	June 17th - June 24th
Fix Period	June 24th - July 10th

ISSUES FOUND

Severity	Count	Fixed	Acknowledged
Critical	0	0	0
High	6	5	1
Medium	5	3	2
Low	8	7	1
Insights	4	3	1

CATEGORY BREAKDOWN

Bug	19
Gas Optimization	0
Informational	4

FINDINGS

IMM-HIGH-01

Incorrect Halving Period Reward Calculation Leads to Significant Token Emission Reduction #20

Id	IMM-HIGH-01
Severity	High
Category	Bug
Status	Fixed in https://github.com/matchain/contracts/pull/10

Description

The `rewardDistribution()` function in MATToken contract incorrectly calculates rewards when distribution periods span across halving events. The function uses the current block reward rate (post-halving) for all blocks since the last distribution, including blocks that occurred before the halving when the reward rate should have been higher.

This issue causes the contract to emit significantly fewer tokens than intended by the economic design (approximately 49% fewer rewards when a distribution spans a halving event). As a result, stakers receive fewer rewards than they should, and the timeline to reach the maximum supply cap is substantially extended.

The issue occurs in the reward calculation where the function uses:

```
TypeScript
uint256 rewardAmount = blockReward * blocksPassed;
```

Where `blockReward` is the current block reward (after any halvings) and `blocksPassed` is the total number of blocks since the last distribution

The function correctly detects when a halving has occurred:

```TypeScript
uint256 lastDistributionPeriod = getHalvingPeriod(_lastRewardDistributionBlock);
uint256 currentPeriod = getHalvingPeriod(block.number);
if (currentPeriod > lastDistributionPeriod) {
    currentHalvingPeriod = currentPeriod;
}
```

However, it only updates the currentHalvingPeriod variable without adjusting the reward calculation to account for the different reward rates before and after the halving.

The impact of this issue is:

1. **Reduced Token Emissions**: The contract mints approximately 50% fewer tokens than intended when distributions span a halving event.
2. **Staker Rewards Reduction**: Stakers receive fewer rewards than promised by the protocol design.
3. **Altered Emission Schedule**: The token's maximum supply will be reached much later than intended.
4. **Economic Model Disruption**: The intended token economic model is not functioning as designed.

Proof of Concept

The following test demonstrates the issue by simulating a reward distribution that spans a halving period:

```TypeScript
// SPDX-License-Identifier: UNLICENSED
pragma solidity ^0.8.10;

import "forge-std/Test.sol";
import "../../contracts/MATToken.sol";
import "../../contracts/MatchConfig.sol";
import "../../contracts/PoolOwnership.sol";
import "../../contracts/StakingPool.sol";
import "../../contracts/StakingPoolFactory.sol";
import "@openzeppelin/contracts/proxy/beacon/UpgradeableBeacon.sol";

contract MatTokenIntegrationTest is Test {
    // Contracts
    MATToken public token;
```

```
MatchConfig public config;
StakingPoolFactory public factory;
UpgradeableBeacon public beacon;
StakingPool public implementation;

// Accounts for testing
address deployer = address(1);
address poolOwner1 = address(2);
address poolOwner2 = address(3);
address staker1 = address(4);
address staker2 = address(5);

// Array for tracking created pools
address[] public pools;

function setUp() public {
    // Configure accounts with funds
    vm.deal(deployer, 100 ether);
    vm.deal(poolOwner1, 10 ether);
    vm.deal(poolOwner2, 10 ether);
    vm.deal(staker1, 10 ether);
    vm.deal(staker2, 10 ether);

    vm.startPrank(deployer);

    // Deploy MatchConfig
    config = new MatchConfig();
    config.initialize(deployer);

    // Deploy MATToken
    token = new MATToken();
    token.initialize("Matchain Token", "MAT", deployer, address(config));

    // Deploy StakingPool implementation
    implementation = new StakingPool();

    // Create Beacon for StakingPool
    beacon = new UpgradeableBeacon(address(implementation), deployer);

    // Deploy Factory
    factory = new StakingPoolFactory();
```

```
        factory.initialize(address(config), address(beacon));

        // Configure addresses in MatchConfig
        config.setToken(address(token));
        config.setPoolFactory(address(factory));

        // Distribute initial tokens
        token.transfer(poolOwner1, 5_000_000 * 1e18); // For self-stake
        token.transfer(poolOwner2, 5_000_000 * 1e18);
        token.transfer(staker1, 1_000_000 * 1e18);
        token.transfer(staker2, 1_000_000 * 1e18);

        vm.stopPrank();
    }


function testRewardDistributionAcrossHalving() public {
    // 1. Create a pool to receive rewards
    address pool1 = createPool(poolOwner1, "ipfs://pool1");

    // 2. Stake in the pool
    vm.startPrank(staker1);
    uint256 stakeAmount = 500_000 * 1e18;
    token.approve(pool1, stakeAmount);
    StakingPool(pool1).stake(stakeAmount);
    vm.stopPrank();

    // 3. Record the initial block and block reward in the current period
    uint256 initialBlock = block.number;
    uint256 initialBlockReward = token.getCurrentBlockReward();
    console.log("Initial block reward:", initialBlockReward / 1e18);

    // 4. Calculate how many blocks until the next halving
    uint256 blocksPerHalving = token.BLOCKS_PER_HALVING();
    uint256 deploymentBlock = token.deploymentBlock();
    uint256 currentHalvingPeriod = token.currentHalvingPeriod();
    uint256 nextHalvingBlock = deploymentBlock + (currentHalvingPeriod + 1) *
blocksPerHalving;
    uint256 blocksUntilHalving = nextHalvingBlock - initialBlock;

    console.log("Current halving period:", currentHalvingPeriod);
```

```
console.log("Blocks until next halving:", blocksUntilHalving);

// 5. Perform a distribution before the halving
vm.roll(initialBlock + 1000);
token.rewardDistribution();
uint256 preHalvingBlock = block.number;

// 6. Advance blocks to pass the halving point
vm.roll(nextHalvingBlock + 1000);
uint256 expectedNewHalvingPeriod = currentHalvingPeriod + 1;

// 7. Record balances before the post-halving distribution
uint256 poolBalanceBefore = token.balanceOf(pool1);
uint256 totalEmittedBefore = token.totalEmitted();

// 8. Perform a post-halving distribution
token.rewardDistribution();

// 9. Verify that we are in a new halving period
assertEq(token.currentHalvingPeriod(), expectedNewHalvingPeriod);

// 10. Calculate the actual distributed reward
uint256 poolReward = token.balanceOf(pool1) - poolBalanceBefore;
uint256 totalEmittedDelta = token.totalEmitted() - totalEmittedBefore;

// 11. Calculate the expected reward manually
uint256 blocksBeforeHalving = nextHalvingBlock - preHalvingBlock;
uint256 blocksAfterHalving = block.number - nextHalvingBlock;
uint256 postHalvingBlockReward = initialBlockReward / 2; // Reward is halved

uint256 expectedReward = (blocksBeforeHalving * initialBlockReward) +
                         (blocksAfterHalving * postHalvingBlockReward);

// 12. Compare actual vs expected reward (with some tolerance for distribution)
console.log("Total blocks passed:", block.number - preHalvingBlock);
console.log("Blocks before halving:", blocksBeforeHalving);
console.log("Blocks after halving:", blocksAfterHalving);
console.log("Actual total reward emitted:", totalEmittedDelta / 1e18);
console.log("Expected total reward:", expectedReward / 1e18);

// Verify that the difference between expected and actual is significant
```

```
    bool isRewardUnderemitted = totalEmittedDelta < expectedReward * 90 / 100; // More than
10% difference

    // If this assert fails, there isn't a significant problem
    assertTrue(isRewardUnderemitted, "Reward calculation across halving periods is correct or
close enough");

    // Print the difference
    console.log("Difference (expected - actual):", (expectedReward - totalEmittedDelta) /
1e18);
    console.log("Difference percentage:", (expectedReward - totalEmittedDelta) * 100 /
expectedReward, "%");
}


    function createPool(address owner, string memory uri) internal returns (address) {
        // Approve factory to spend owner's tokens
        vm.startPrank(owner);
        token.approve(address(factory), MIN_POOL_SELFSTAKE);

        // Create pool
        factory.mintPool(
            owner,
            uri,
            MIN_LOCK_PERIOD_DAYS,    // 2 years of lock
            MIN_VESTING_PERIOD_DAYS  // 3 years of vesting
        );
        vm.stopPrank();

        // Get the address of the new pool
        // Count pools manually
        uint256 poolCount = 0;
        bool hasMorePools = true;
        while (hasMorePools) {
            try token._pools(poolCount) returns (address) {
                poolCount++;
            } catch {
                hasMorePools = false;
            }
        }
        address poolAddress = token._pools(poolCount - 1);
```

```
        pools.push(poolAddress);

        return poolAddress;
    }
```

Test output confirms the issue:

```
None
Initial block reward: 0
Current halving period: 0
Blocks until next halving: 180000000
Total blocks passed: 180000000
Blocks before halving: 179999000
Blocks after halving: 1000
Actual total reward emitted: 14999999
Expected total reward: 29999916
Difference (expected - actual): 14999916
Difference percentage: 49 %
```

Recommendation

Option 1: Per-Halving-Period Calculation

Modify the reward distribution function to calculate rewards separately for each halving period:

```typescript
function rewardDistribution() external {
    uint256 blocksPassed = block.number - _lastRewardDistributionBlock;
    require(blocksPassed > 0, "No blocks passed since last distribution");

    // ... existing pool activation code ...

    uint256 lastDistributionPeriod = getHalvingPeriod(_lastRewardDistributionBlock);
    uint256 currentPeriod = getHalvingPeriod(block.number);

    uint256 rewardAmount = 0;
```

```
    if (currentPeriod == lastDistributionPeriod) {
        // Simple case: no halving occurred
        uint256 blockReward = getCurrentBlockReward();
        rewardAmount = blockReward * blocksPassed;
    } else {
        // Complex case: halving occurred
        for (uint256 period = lastDistributionPeriod; period <= currentPeriod; period++) {
            uint256 periodStartBlock = deploymentBlock + (period * BLOCKS_PER_HALVING);
            uint256 periodEndBlock = deploymentBlock + ((period + 1) * BLOCKS_PER_HALVING);

            uint256 startBlock = period == lastDistributionPeriod ?
_lastRewardDistributionBlock : periodStartBlock;
            uint256 endBlock = period == currentPeriod ? block.number : periodEndBlock;

            if (startBlock < endBlock) {
                uint256 periodBlocksPassed = endBlock - startBlock;
                uint256 periodBlockReward = INITIAL_BLOCK_REWARD / (2 ** period);
                rewardAmount += periodBlockReward * periodBlocksPassed;
            }
        }
    }

    // Update the current halving period
    if (currentPeriod > lastDistributionPeriod) {
        currentHalvingPeriod = currentPeriod;
    }

    // ... continue with max supply check and distribution ...
}
```

Option 2: Tracking Cumulative Rewards

Alternatively, implement a cumulative reward tracking approach that increments rewards based on blocks and halving periods:

```
TypeScript
// Add this storage variable
uint256 private _cumulativeRewardsAtLastDistribution;
```

```
function rewardDistribution() external {
    uint256 newCumulativeRewards = calculateCumulativeRewards(block.number);
    uint256 rewardAmount = newCumulativeRewards - _cumulativeRewardsAtLastDistribution;

    _cumulativeRewardsAtLastDistribution = newCumulativeRewards;
    _lastRewardDistributionBlock = block.number;

    // ... continue with existing distribution logic ...
}

function calculateCumulativeRewards(uint256 blockNumber) public view returns (uint256) {
    uint256 cumulativeRewards = 0;
    uint256 lastHalvingPeriod = getHalvingPeriod(deploymentBlock);
    uint256 currentHalvingPeriod = getHalvingPeriod(blockNumber);

    for (uint256 period = 0; period <= currentHalvingPeriod; period++) {
        uint256 periodStartBlock = deploymentBlock + (period * BLOCKS_PER_HALVING);
        uint256 periodEndBlock = periodStartBlock + BLOCKS_PER_HALVING;

        uint256 startBlock = period == 0 ? deploymentBlock : periodStartBlock;
        uint256 endBlock = period == currentHalvingPeriod ? blockNumber : periodEndBlock;

        if (startBlock < endBlock) {
            uint256 periodBlocksPassed = endBlock - startBlock;
            uint256 periodBlockReward = INITIAL_BLOCK_REWARD / (2 ** period);
            cumulativeRewards += periodBlockReward * periodBlocksPassed;
        }
    }

    return cumulativeRewards;
}
```

IMM-HIGH-02

Exchange Rate Function Enables Direct Protocol Fee Theft #21

Id	IMM-HIGH-02
Severity	High
Category	Bug
Status	Acknowledged

Description

The `exchangeRate()` function in the `LiquidStakingPool` contract combines both reading and state-modifying operations in a single public function. This critical design flaw enables a direct front-running attack that can systematically deprive the protocol of fee revenue with minimal effort and no capital requirement.

The vulnerable code is in the `exchangeRate()` function:

```
TypeScript
function exchangeRate() public returns (uint256) {
    if (totalSupply() == 0) return 1e18; // Initial 1:1 ratio
    uint256 currentMATBalance = stakingPool.currentStake(address(this));
    // rewards case
    if (currentMATBalance > lastRecordedStake) {
        uint256 newRewards = currentMATBalance - lastRecordedStake;
        uint256 feeAmount = (newRewards * lspFee) / 10000;
        accumulatedFees += feeAmount;
        lastRecordedStake = currentMATBalance;
    }
    // balance decrease case - sanity check
    else if (currentMATBalance < lastRecordedStake) {
        lastRecordedStake = currentMATBalance;
    }
    uint256 effectiveBalance = currentMATBalance > accumulatedFees ?
        currentMATBalance - accumulatedFees : 0;
    uint256 calculatedRate = (effectiveBalance * 1e18) / totalSupply();
    return calculatedRate < 1e18 ? 1e18 : calculatedRate;   // never falls below 1:1
}
```

The core vulnerability stems from the dual nature of the `exchangeRate()` function. While it appears to be a simple getter function that returns the current exchange rate between stMAT and MAT, it actually performs

significant state changes by updating the `lastRecordedStake` variable and calculating protocol fees. Since this function is public, anyone can call it directly without any restrictions or capital requirements.

An attacker can exploit this by:

1. Monitoring the mempool for pending reward distribution transactions
2. Front-running these transactions with a direct call to `exchangeRate()`
3. This updates `lastRecordedStake` to the current balance (before rewards)
4. When rewards are distributed immediately after, the difference between the new balance and `lastRecordedStake` is minimal
5. The protocol calculates fees on this small difference instead of the full reward amount

What makes this attack particularly concerning is its simplicity and low barrier to entry. The attacker doesn't need to stake any tokens, doesn't need any special permissions, and only pays for gas costs. With a simple automated bot monitoring the mempool, this attack could be executed continuously against every reward distribution.

The long-term impact of this vulnerability is severe. By systematically intercepting fee collection, an attacker effectively cuts off the protocol's primary revenue stream.

Concrete Example
Let's walk through a numerical example:

- Current state:
  - `lastRecordedStake` = 1,000,000 MAT (from a week ago)
  - Actual current stake = 1,100,000 MAT (100,000 MAT in uncounted rewards)
  - `lspFee` = 1000 (10%)

- Expected behavior (without attack):
  - When `exchangeRate()` is called, it should calculate:
    - `newRewards` = 1,100,000 - 1,000,000 = 100,000 MAT
    - `feeAmount` = 100,000 × 10% = 10,000 MAT
    - `accumulatedFees` increases by 10,000 MAT
    - `lastRecordedStake` updates to 1,100,000 MAT

- With attack:
  - Attacker front-runs with a minimal transaction calling `exchangeRate()`

- `lastRecordedStake` updates to 1,100,000 MAT (current balance before new rewards)
- Reward distribution adds another 50,000 MAT
- Next legitimate call to `exchangeRate()` calculates:
  - `newRewards` = 1,150,000 - 1,100,000 = 50,000 MAT (instead of 150,000)
  - `feeAmount` = 50,000 × 10% = 5,000 MAT (instead of 15,000)

Protocol loses 10,000 MAT in fee revenue

Recommendation

Separate Read and Write Operations:

- Create a view-only function for getting the exchange rate (the contract already has `getExchangeRate()` but it's not used consistently)
- Create a separate permissioned function for updating fee accounting
- Ensure all user-facing functions use the view function, not the state-modifying one

TypeScript

```
// Make the existing exchangeRate function internal or private
function _updateExchangeRate() internal returns (uint256) {
    // Current implementation of exchangeRate()
    // ...
}

// Use the existing view function for all rate queries
function exchangeRate() public returns (uint256) {
    return getExchangeRate();
}

// Add a permissioned function to update accounting
function updateFeeAccounting() public onlyRole(FEE_MANAGER_ROLE) {
    uint256 currentMATBalance = stakingPool.currentStake(address(this));
    if (currentMATBalance > lastRecordedStake) {
        uint256 newRewards = currentMATBalance - lastRecordedStake;
        uint256 feeAmount = (newRewards * lspFee) / 10000;
        accumulatedFees += feeAmount;
        lastRecordedStake = currentMATBalance;
    } else if (currentMATBalance < lastRecordedStake) {
        lastRecordedStake = currentMATBalance;
    }
```

```
}
```

IMM-HIGH-03

Reentrancy Vulnerability When Transferring Pool Tokens in `PoolOwnership.sol` #23

Id	IMM-HIGH-03
Severity	High
Category	Bug
Status	Fixed in https://github.com/matchain/contracts/pull/7

Description

The `PoolOwnership.sol` contract implements ERC721 transfers for ownership of staking pools. In the following functions:

```
TypeScript
function _safeTransfer(address from, address to, uint256 tokenId, bytes memory data) internal
virtual override {
    super._safeTransfer(from, to, tokenId, data);
    StakingPool(address(uint160(tokenId))).moveStakeToSelfStake(to);
}

function transferFrom(address from, address to, uint256 tokenId) public virtual override {
    super.transferFrom(from, to, tokenId);
    StakingPool(address(uint160(tokenId))).moveStakeToSelfStake(to);
}
```

The contract first calls the standard ERC721 transfer logic via `super._safeTransfer` or `super.transferFrom`. This transfers ownership of the NFT, which represents control of a `StakingPool`, to the address. Only after the transfer is completed, the `moveStakeToSelfStake(to)` function is called on the `StakingPool` contract to update staking balances.

This ordering introduces a critical vulnerability:

- If the address is a smart contract, it can implement `onERC721Received`.
- Within `onERC721Received`, the contract can perform arbitrary actions on the `StakingPool`, since it is now recognized as the owner of the pool.

- However, the internal staking state has not yet been updated. Specifically, the `stakes[to]` balance has not yet been migrated to `selfStake` via `moveStakeToSelfStake`.

Here's the relevant staking update logic from `StakingPool`:

```
TypeScript
function moveStakeToSelfStake(address staker) public {
    require(msg.sender == address(ownershipNFT), "Only NFT contract can call");
    if (stakes[staker] > 0) {
        selfStake += stakes[staker];
        stakes[staker] = 0;
    }
}
```

Because the transfer happens before calling this function, the recipient has a temporary window where:

- They own the pool NFT.
- Their `stakes[to]` is still intact, and `selfStake` has not yet been increased.

This opens the door to reentrancy attacks or manipulation of the staking logic. The new owner could:

- Call other methods on the `StakingPool` using an outdated balance.
- Trigger a withdraw, delegate, or any operation that reads from `stakes[to]` before it's cleared.
- Exploit this outdated state to double-count, bypass checks, or extract unintended rewards.

Since the NFT transfer includes an external call (`onERC721Received`), and the state update occurs after that call, the contract violates the checks-effects-interactions pattern, a core Solidity security principle to prevent reentrancy and inconsistent state issues.

PoC:

```
TypeScript

// SPDX-License-Identifier: UNLICENSED
pragma solidity ^0.8.10;

import "forge-std/Test.sol";
import "forge-std/console.sol";
import {MATToken, MIN_POOL_SELFSTAKE} from "../../contracts/MATToken.sol";
```

```
import "../../contracts/MatchConfig.sol";
import {StakingPool, MIN_LOCK_PERIOD_DAYS, MIN_VESTING_PERIOD_DAYS} from
"../../contracts/StakingPool.sol";
import "../../contracts/LiquidStakingPool.sol";
import "../../contracts/StakingPoolFactory.sol";
import "../../contracts/PoolOwnership.sol";
import "@openzeppelin/contracts/proxy/beacon/UpgradeableBeacon.sol";
import "@openzeppelin/contracts/proxy/beacon/BeaconProxy.sol";
import "@openzeppelin/contracts/token/ERC721/IERC721Receiver.sol";
import "@openzeppelin/contracts-upgradeable/proxy/utils/Initializable.sol";

contract AttackerContract is IERC721Receiver {
    MATToken public token;
    StakingPool public stakingPool;
    address public owner;
    uint256 public exploitedAmount;
    bool public attackMode;

    constructor(address _token, address _stakingPool) {
        token = MATToken(_token);
        stakingPool = StakingPool(_stakingPool);
        owner = msg.sender;
    }

    function onERC721Received(
        address operator,
        address from,
        uint256 tokenId,
        bytes calldata data
    ) external override returns (bytes4) {
        if (attackMode) {
            exploitedAmount = stakingPool.currentStake(address(this));
            console.log("  [EXPLOIT] Attack detected during NFT transfer");
            console.log("  [EXPLOIT] Vulnerable amount:", exploitedAmount);
        }

        return IERC721Receiver.onERC721Received.selector;
    }

    function enableAttack() external {
        require(msg.sender == owner, "Not owner");
```

```
        attackMode = true;
    }

    function withdrawExploitedFunds() external {
        require(msg.sender == owner, "Not owner");
    }
}

contract PoolOwnershipReentrancyTest is Test {
    // Contracts
    MATToken public token;
    MatchConfig public config;
    StakingPoolFactory public factory;
    UpgradeableBeacon public stakingPoolBeacon;
    UpgradeableBeacon public liquidStakingPoolBeacon;
    StakingPool public stakingPoolImplementation;
    LiquidStakingPool public liquidStakingPoolImplementation;
    StakingPool public stakingPool;
    PoolOwnership public ownershipNFT;
    AttackerContract public attacker;

    address deployer = address(1);
    address poolOwner = address(2);
    address staker1 = address(3);
    address staker2 = address(4);
    address smallStaker = address(5);
    address feeVault = address(6);
    address normalUser = address(7);

    // Constants
    uint256 constant INITIAL_MAT_SUPPLY = 10_000_000 * 1e18;
    uint256 constant LARGE_STAKE = 1_000_000 * 1e18;

    uint256 public poolNFTId;

    function setUp() public {
        vm.deal(deployer, 100 ether);
        vm.deal(poolOwner, 10 ether);
        vm.deal(staker1, 10 ether);
        vm.deal(staker2, 10 ether);
        vm.deal(smallStaker, 10 ether);
```

```
        vm.deal(normalUser, 10 ether);

        config = new MatchConfig();
        vm.prank(deployer);
        config.initialize(deployer);
        token = new MATToken();

        vm.prank(deployer);
        token.initialize("Matchain Token", "MAT", deployer, address(config));

        vm.startPrank(deployer);
        stakingPoolImplementation = new StakingPool();
        liquidStakingPoolImplementation = new LiquidStakingPool();
        stakingPoolBeacon = new UpgradeableBeacon(address(stakingPoolImplementation),
deployer);
        liquidStakingPoolBeacon = new
UpgradeableBeacon(address(liquidStakingPoolImplementation), deployer);
        factory = new StakingPoolFactory();
        factory.initialize(address(config), address(stakingPoolBeacon));

        config.setToken(address(token));
        config.setPoolFactory(address(factory));

        token.transfer(poolOwner, LARGE_STAKE);
        vm.stopPrank();

        vm.recordLogs();

        vm.startPrank(poolOwner);
        token.approve(address(factory), 250_000 * 1e18);
        factory.mintPool(poolOwner, "test-uri", MIN_LOCK_PERIOD_DAYS,
MIN_VESTING_PERIOD_DAYS);
        vm.stopPrank();

        Vm.Log[] memory entries = vm.getRecordedLogs();

        address poolAddress;
        bytes32 poolMintedSelector = keccak256("PoolMinted(address)");

        for (uint i = 0; i < entries.length; i++) {
            if (entries[i].topics[0] == poolMintedSelector) {
```

```
            poolAddress = abi.decode(entries[i].data, (address));
            break;
        }
    }

    require(poolAddress != address(0), "Failed to find pool address in events");

    stakingPool = StakingPool(poolAddress);
    ownershipNFT = stakingPool.ownershipNFT();
    poolNFTId = uint256(uint160(address(stakingPool)));

    vm.startPrank(deployer);
    token.transfer(poolOwner, LARGE_STAKE);
    token.transfer(staker1, LARGE_STAKE);
    token.transfer(staker2, LARGE_STAKE / 2);
    vm.stopPrank();

    vm.startPrank(staker1);
    token.approve(address(stakingPool), LARGE_STAKE / 5);
    stakingPool.stake(LARGE_STAKE / 5);
    vm.stopPrank();

    vm.startPrank(staker2);
    token.approve(address(stakingPool), LARGE_STAKE / 10);
    stakingPool.stake(LARGE_STAKE / 10);
    vm.stopPrank();

    attacker = new AttackerContract(address(token), address(stakingPool));

    vm.startPrank(deployer);
    token.transfer(address(attacker), LARGE_STAKE / 10);
    vm.stopPrank();

    vm.prank(address(attacker));
    token.approve(address(stakingPool), LARGE_STAKE);

    vm.prank(address(attacker));
    stakingPool.stake(LARGE_STAKE / 10);
    vm.stopPrank();
}
```

```
    /**
     * @notice Test to demonstrate the reentrancy vulnerability in PoolOwnership.sol
     * @dev This test shows how a malicious contract can exploit the window
     * between receiving the NFT and updating the staking state
     */
    function testPoolOwnershipReentrancyVulnerability() public {
        // Verify the initial state
        uint256 initialAttackerStake = stakingPool.currentStake(address(attacker));
        uint256 initialSelfStake = stakingPool.currentStake(poolOwner);

        console.log("======= INITIAL STATE =======");
        console.log("NFT Owner:", ownershipNFT.ownerOf(poolNFTId));
        console.log("Attacker stake:", initialAttackerStake);
        console.log("Initial pool self-stake:", initialSelfStake);

        // Verify that the attacker has stake
        assertGt(initialAttackerStake, 0, "Attacker must have initial stake for the test");

        attacker.enableAttack();

        console.log("\n======= EXECUTING NFT TRANSFER =======");
        // The pool owner transfers the NFT to the attacker contract
        vm.startPrank(poolOwner);
        ownershipNFT.safeTransferFrom(poolOwner, address(attacker), poolNFTId);
        vm.stopPrank();

        // Verify that the transfer was completed successfully
        assertEq(ownershipNFT.ownerOf(poolNFTId), address(attacker), "Attacker should be the
new owner of the NFT");

        // Verify the amount that the attacker could "see" during the vulnerability window
        uint256 exploitedAmount = attacker.exploitedAmount();
        uint256 finalSelfStake = stakingPool.currentStake(address(attacker)); // attacker is
now the owner
        uint256 finalAttackerStake = stakingPool.currentStake(address(attacker));

        console.log("\n======= STATE AFTER ATTACK =======");
        console.log("New NFT owner:", ownershipNFT.ownerOf(poolNFTId));
        console.log("Vulnerable amount during attack:", exploitedAmount);
        console.log("Final pool self-stake:", finalSelfStake);
        console.log("Final attacker stake:", finalAttackerStake);
```

```
        vm.assertGt(attacker.exploitedAmount(), 0, "Attacker could access their stake during
the vulnerability window");

        vm.assertEq(
            stakingPool.currentStake(address(attacker)),
            attacker.exploitedAmount(),
            "VULNERABILITY CONFIRMED: Attacker's stake was NOT correctly moved to selfStake
due to reentrancy"
        );
        assertGt(finalSelfStake, initialAttackerStake, "Current stake of the attacker (new
owner) should include the previous self stake");

        console.log("\n======= TEST CONCLUSION =======");
        console.log("Vulnerability confirmed: Attacker could access their stake during the
transfer");
        console.log("In a real attack, this access would allow state manipulation and
possible double spending");
    }
}
```

Logs

TypeScript

```
Logs:
  ======= INITIAL STATE =======
  NFT Owner: 0x0000000000000000000000000000000000000002
  Attacker stake: 100000000000000000000
  Initial pool self-stake: 100000000000000000000000

======= EXECUTING NFT TRANSFER =======
    [EXPLOIT] Attack detected during NFT transfer
    [EXPLOIT] Vulnerable amount: 110000000000000000000000

======= STATE AFTER ATTACK =======
  New NFT owner: 0xF62849F9A0B5Bf2913b396098F7c7019b51A820a
  Vulnerable amount during attack: 110000000000000000000000
  Final pool self-stake: 110000000000000000000000
```

```
  Final attacker stake: 110000000000000000000000

======= TEST CONCLUSION =======
  Vulnerability confirmed: Attacker could access their stake during the transfer
  In a real attack, this access would allow state manipulation and possible double spending
```

Recommendation

Reorder the logic: Always update internal contract state before performing external calls or transferring ownership. In this case, moveStakeToSelfStake(to) should be called before the ERC721 token is transferred.

TypeScript

```
function _safeTransfer(address from, address to, uint256 tokenId, bytes memory data) internal
virtual override {
    StakingPool(address(uint160(tokenId))).moveStakeToSelfStake(to);
    super._safeTransfer(from, to, tokenId, data);
}
```

IMM-HIGH-04

Reentrancy via Ownership Transfer Before Stake State Update in `StakingPool` #24

Id	IMM-HIGH-04
Severity	High
Category	Bug
Status	Fixed in `Resolved in` https://github.com/matchain/contracts/pull/6

Description

In `StakingPool.sol`, the `_transferOwnership` function is overridden to support ownership handoff via ERC721 transfer of the `ownershipNFT` token. However, the current implementation introduces a serious reentrancy vulnerability due to the incorrect ordering of operations.

Here is the function in question:

```
TypeScript
function _transferOwnership(address newOwner) internal virtual override {
    address oldOwner = owner();
    if (oldOwner == newOwner) return;

    ownershipNFT.transferFrom(oldOwner, newOwner, uint160(address(this)));

    if (stakes[newOwner] > 0) {
        selfStake += stakes[newOwner];
        stakes[newOwner] = 0;
    }

    emit OwnershipTransferred(oldOwner, newOwner);
}
```

The issue arises from the call to `ownershipNFT.transferFrom(...)`, which occurs before updating the internal staking state via:

```
TypeScript
if (stakes[newOwner] > 0) {
    selfStake += stakes[newOwner];
    stakes[newOwner] = 0;
}
```

This ordering violates the checks-effects-interactions pattern because transferFrom may trigger arbitrary logic, particularly the onERC721Received hook, on the newOwner if it is a smart contract. At this point in execution:

- The `newOwner` already owns the pool NFT.
- The `stakes[newOwner]` balance is not yet migrated to `selfStake`.
- The `owner()` has not been officially updated yet either (depending on how ownership is managed), potentially introducing inconsistencies.

Recommendation

- Reorder the logic to update staking state before transferring the NFT:

```
TypeScript
function _transferOwnership(address newOwner) internal virtual override {
    address oldOwner = owner();
    if (oldOwner == newOwner) return;

    // Migrate stake to selfStake before transferring NFT
    if (stakes[newOwner] > 0) {
        selfStake += stakes[newOwner];
        stakes[newOwner] = 0;
    }

    ownershipNFT.transferFrom(oldOwner, newOwner, uint160(address(this)));

    emit OwnershipTransferred(oldOwner, newOwner);
}
```

- Add nonReentrant modifier

IMM-HIGH-05

Incorrect Reward Rate Selection in Cross-Halving Distributions Causes Token Underemission #31

Id	IMM-HIGH-05
Severity	High
Category	Bug
Status	Fixed in https://github.com/matchain/contracts/pull/10/commits/7a68ac78c947dfb7ab4b8e32d0a9167e4419dfb4

Description

In the `rewardDistribution()` function of the `MATToken` contract, the reward rate is incorrectly used when distribution crosses a halving boundary.
This issue was introduced by the following pull request: https://github.com/matchain/contracts/pull/10, which was initially intended to address the IMM-HIGH-1 finding.

Specifically, the error is in these lines:

```
TypeScript
uint256 rewardAmount = getBlockReward(nextStep) * (nextStep - _lastRewardDistributionBlock);
```

And inside the while loop:

```
TypeScript
rewardAmount += getBlockReward(nextStep) * (nextStep - oldNextStep);
```

The problem is as follows:

- In `getBlockReward(nextStep)`, the reward rate of the next halving period is applied to all blocks in the current segment.
- `nextStep` is the end of the segment, which in many cases already belongs to the next halving period

with a reduced rate.
- It should use the rate from the beginning of the segment (`_lastRewardDistributionBlock` or `oldNextStep`).
- Since the reward rate decreases in each halving period, using the rate from the next period results in a lower emission than it should be.

The reward rate is constant within a single halving period but decreases by half between different halving periods. The problem occurs because the code uses the reward rate from the end of each segment (`nextStep`) for the entire segment calculation.

When `nextStep` is the halving block or later, `getBlockReward(nextStep)` returns the reduced post-halving rate. This reduced rate is then incorrectly applied to all blocks in the segment, including those that occurred before the halving and should have received the higher rate.

For example, when `_lastRewardDistributionBlock` is before the halving boundary and `nextStep` is the halving block itself, the entire segment (including all pre-halving blocks) gets calculated using the lower post-halving rate. This is mathematically incorrect and results in significant underemission of tokens.

Proof of Concept

TypeScript
```
function test_halvingBug() public {
        // 1. Create a pool to receive rewards
        address pool1 = createPool(poolOwner1, "ipfs://pool1");

        // 2. Make stake in the pool to activate it
        vm.startPrank(staker1);
        uint256 stakeAmount = 500_000 * 1e18;
        token.approve(pool1, stakeAmount);
        StakingPool(pool1).stake(stakeAmount);
        vm.stopPrank();

        // 4. Calculate the block of the first halving
        uint256 blocksPerHalving = token.BLOCKS_PER_HALVING();
        uint256 firstHalvingBlock = token.deploymentBlock() + blocksPerHalving;

        console.log("\n=== DIRECT DEMOSTRATION OF HALVING BUG ===\n");
        console.log("Block deployment:", token.deploymentBlock());
```

```
        console.log("Block of first halving:", firstHalvingBlock);

        // 5. CASE 1: Distribution that DOES NOT cross a halving limit
        // Advance 100 blocks before the halving
        uint256 startBlock = firstHalvingBlock - 200;
        uint256 endBlock = firstHalvingBlock - 100;
        uint256 blockCount = endBlock - startBlock;

        // Make distribution in startBlock
        vm.roll(startBlock);
        token.rewardDistribution();

        // Advance to endBlock and register state before distribution
        vm.roll(endBlock);
        uint256 initialTotalEmitted = token.totalEmitted();

        // Execute distribution and measure emitted tokens
        token.rewardDistribution();
        uint256 emissionNormal = token.totalEmitted() - initialTotalEmitted;

        // Calculate emission rate per block (without crossing halving)
        uint256 emissionRateNormal = emissionNormal / blockCount;

        console.log("\nCASE 1: Distribution without crossing halving");
        console.log("Blocks processed:", blockCount);
        console.log("Tokens emitted:", emissionNormal / 1e18);
        console.log("Emission rate per block (wei):", emissionRateNormal);
        console.log("Emission rate per block (scaled):", emissionRateNormal * 1000 / 1e18, "*
10^-3");

        // 6. CASE 2: Distribution that crosses a halving limit
        startBlock = firstHalvingBlock - 50; // 50 blocks before the halving
        endBlock = firstHalvingBlock + 50;   // 50 blocks after the halving
        blockCount = endBlock - startBlock;   // Same number of blocks as case 1

        // Make distribution in startBlock
        vm.roll(startBlock);
        token.rewardDistribution();

        // Advance to endBlock and register state before distribution
        vm.roll(endBlock);
```

```
        initialTotalEmitted = token.totalEmitted();

        // Execute distribution and measure emitted tokens
        token.rewardDistribution();
        uint256 emissionCrossHalving = token.totalEmitted() - initialTotalEmitted;

        // Calculate emission rate per block (crossing halving)
        uint256 emissionRateCrossHalving = emissionCrossHalving / blockCount;

        console.log("\nCASE 2: Distribution crossing halving");
        console.log("Blocks processed:", blockCount);
        console.log("Tokens emitted:", emissionCrossHalving / 1e18);
        console.log("Emission rate per block (wei):", emissionRateCrossHalving);
        console.log("Emission rate per block (scaled):", emissionRateCrossHalving * 1000 /
1e18, "* 10^-3");

        // 7. CASE 3: Distribution after halving
        startBlock = firstHalvingBlock + 100;
        endBlock = firstHalvingBlock + 200;
        blockCount = endBlock - startBlock;

        // Make distribution in startBlock
        vm.roll(startBlock);
        token.rewardDistribution();

        // Advance to endBlock and register state before distribution
        vm.roll(endBlock);
        initialTotalEmitted = token.totalEmitted();

        // Execute distribution and measure emitted tokens
        token.rewardDistribution();
        uint256 emissionAfterHalving = token.totalEmitted() - initialTotalEmitted;

        // Calculate emission rate per block (after halving)
        uint256 emissionRateAfterHalving = emissionAfterHalving / blockCount;

        console.log("\nCASE 3: Distribution after halving");
        console.log("Blocks processed:", blockCount);
        console.log("Tokens emitted:", emissionAfterHalving / 1e18);
        console.log("Emission rate per block (wei):", emissionRateAfterHalving);
        console.log("Emission rate per block (scaled):", emissionRateAfterHalving * 1000 /
```

```
1e18, "* 10^-3");

        // 8. Compare results and demonstrate the bug
        console.log("\nCOMPARATION OF EMISSION RATES:");
        console.log("Rate before halving (wei):", emissionRateNormal);
        console.log("Rate crossing halving (wei):", emissionRateCrossHalving);
        console.log("Rate after halving (wei):", emissionRateAfterHalving);

        console.log("Rate before halving (scaled):", emissionRateNormal * 1000 / 1e18, "*
10^-3");
        console.log("Rate crossing halving (scaled):", emissionRateCrossHalving * 1000 /
1e18, "* 10^-3");
        console.log("Rate after halving (scaled):", emissionRateAfterHalving * 1000 / 1e18,
"* 10^-3");

        // 9. Calculate the expected rate for case 2 (average of cases 1 and 3)
        uint256 expectedAvgRate = (emissionRateNormal + emissionRateAfterHalving) / 2;
        uint256 expectedEmission = expectedAvgRate * blockCount;

        console.log("\nANALYSIS OF THE BUG:");
        console.log("Expected average rate (wei):", expectedAvgRate);
        console.log("Expected average rate (scaled):", expectedAvgRate * 1000 / 1e18, "*
10^-3");
        console.log("Expected emission crossing halving:", expectedEmission / 1e18);
        console.log("Actual emission crossing halving:", emissionCrossHalving / 1e18);

        // 10. Calculate the difference and percentage of underemission
        uint256 emissionDiff = expectedEmission - emissionCrossHalving;
        uint256 percentUnderemission = emissionDiff * 100 / expectedEmission;

        console.log("Difference of emission:", emissionDiff / 1e18);
        console.log("Percentage of underemission:", percentUnderemission, "%");

        // 11. Verifications
        // The rate crossing halving should be less than the average of the rates before and
after
        assertTrue(emissionRateCrossHalving < expectedAvgRate, "The rate crossing halving
should be less than the average");

        // The rate crossing halving should be closer to the rate after the halving than to
the rate before the halving
```

```
        uint256 diffToBeforeRate = emissionRateNormal - emissionRateCrossHalving;
        uint256 diffToAfterRate = emissionRateCrossHalving - emissionRateAfterHalving;
        assertTrue(diffToBeforeRate > diffToAfterRate, "The rate crossing halving is closer
to the rate after the halving");

        // There should be a significant underemission
        assertTrue(percentUnderemission > 20, "There should be a significant underemission
(>20%)");

        console.log("\nCONCLUSION: The bug causes an underemission of approximately",
percentUnderemission, "%");
    }
```

TypeScript

```
  Block deployment: 1
  Block of first halving: 180000001

CASE 1: Distribution without crossing halving
  Blocks processed: 100
  Tokens emitted: 16
  Emission rate per block (wei): 166666666666666666
  Emission rate per block (scaled): 166 * 10^-3

CASE 2: Distribution crossing halving
  Blocks processed: 100
  Tokens emitted: 8
  Emission rate per block (wei): 83333333333333333
  Emission rate per block (scaled): 83 * 10^-3

CASE 3: Distribution after halving
  Blocks processed: 100
  Tokens emitted: 8
  Emission rate per block (wei): 83333333333333333
  Emission rate per block (scaled): 83 * 10^-3

COMPARATION OF EMISSION RATES:
  Rate before halving (wei): 166666666666666666
  Rate crossing halving (wei): 83333333333333333
```

```
  Rate after halving (wei): 83333333333333333
  Rate before halving (scaled): 166 * 10^-3
  Rate crossing halving (scaled): 83 * 10^-3
  Rate after halving (scaled): 83 * 10^-3

ANALYSIS OF THE BUG:
  Expected average rate (wei): 124999999999999999
  Expected average rate (scaled): 124 * 10^-3
  Expected emission crossing halving: 12
  Actual emission crossing halving: 8
  Difference of emission: 4
  Percentage of underemission: 33 %

CONCLUSION: The bug causes an underemission of approximately 33 %
```

Case 1 (before halving):

- Blocks processed: 100 blocks before halving
- Tokens emitted: 16 tokens
- Emission rate: 0.166 tokens per block

Case 2 (crossing halving):

- Blocks processed: 100 blocks (50 before + 50 after halving)
- Tokens emitted: 8 tokens
- Emission rate: 0.083 tokens per block

Case 3 (after halving):

- Blocks processed: 100 blocks after halving
- Tokens emitted: 8 tokens
- Emission rate: 0.083 tokens per block

Analysis of results

- The emission rate of Case 2 is identical to Case 3, despite Case 2 including 50 blocks before halving that should have a higher rate.
- The expected rate for Case 2 should be an average between the rates of Case 1 and Case 3:
    - Expected rate = (0.166 + 0.083) / 2 = 0.124 tokens per block

    - Expected emission = 0.124 * 100 = 12.4 tokens (rounded to 12)
- The actual emission (8 tokens) is significantly less than the expected emission (12 tokens), representing a 33% underemission.

Recommendation

Modify the `rewardDistribution()` function to use the reward rate from the beginning of each segment:

```
TypeScript
// For the first segment
uint256 rewardAmount = getBlockReward(_lastRewardDistributionBlock) * (nextStep -
_lastRewardDistributionBlock);

// For subsequent segments in the while loop
rewardAmount += getBlockReward(oldNextStep) * (nextStep - oldNextStep);
```

IMM-HIGH-06

Unrestricted Pool Creation Leads to Reward Starvation Beyond First 21 Pools #8

Id	IMM-HIGH-06
Severity	High
Category	Bug
Status	Fixed in https://github.com/matchain/contracts/pull/17

Description

The `registerNewPool()` function in the `MATToken` contract allows the `StakingPoolFactory` to register new staking pools without enforcing any cap or upper bound on the number of pools:

```
TypeScript
function registerNewPool(address pool, address payer) public {
    require(msg.sender == MatchConfig(config).poolFactory(), "Only pool factory");
    _transfer(payer, pool, MIN_POOL_SELFSTAKE);
    _pools.push(pool);
}
```

While this function does correctly restrict access to only the pool factory, it does not restrict the number of pools that can be added to the `_pools` array. As a result, users can continually deploy and register new pools via `mintPool()` in the `StakingPoolFactory` contract, leading to an unbounded growth in the pool list.

However, in the `rewardDistribution()` function, rewards are only distributed to the first 21 active pools:

```
TypeScript
StakingPool[MAX_POOL_NUMBER] memory activePools;
...
if (poolIndex == MAX_POOL_NUMBER) break;
```

Here, `MAX_POOL_NUMBER` is a hard-coded constant (uint256 public constant `MAX_POOL_NUMBER = 21`;), and only the first 21 active pools in `_pools` are selected for reward distribution. Any pool registered beyond this limit,

regardless of its stake or activity, will never receive staking rewards, leading to:

- Reward starvation for later pools.
- Unfair economic advantage for early pool creators.
- Potential for griefing or DoS-like behavior where actors flood the system with dummy pools to prevent others from entering the top 21.

This contradicts principles of fairness and decentralization, and severely limits protocol scalability.

Recommendation

Several options or design adjustments are available to address this issue. It is recommended to rethink the current reward distribution model to ensure that all active pools receive rewards fairly, not just the first 21.

- Avoid relying on a fixed-size array and static cap (`MAX_POOL_NUMBER`) in the reward distribution logic.
- The current push-based pattern, where the token contract actively mints and distributes to pools, is simple but does not scale well and is prone to fairness issues.

IMM-MED-01

`getCurrentBlockReward()` May Return Outdated Reward Due to Stale `currentHalvingPeriod` #16

Id	IMM-MED-01
Severity	Medium
Category	Bug
Status	Fixed in https://github.com/matchain/contracts/pull/9

Description

The `MattToken` contract exposes a public variable `currentHalvingPeriod`, which is used to determine the current halving stage of the token emission schedule. This variable is used in the public function `getCurrentBlockReward()` to calculate the reward per block:

```
TypeScript
function getCurrentBlockReward() public view returns (uint256) {
    if (totalEmitted >= MAX_SUPPLY) return 0;
    uint256 baseReward = REWARD_PER_THREE_BLOCKS / 3;
    uint256 reward = baseReward >> currentHalvingPeriod; // @audit this can be outdated if
getHalvingPeriod() is not called before
    uint256 remainingSupply = MAX_SUPPLY - totalEmitted;
    return reward > remainingSupply ? remainingSupply : reward;
}
```

However, this function does not check whether the halving period has changed based on the number of blocks passed since the last update. The function `getHalvingPeriod(uint256 blockNumber)` does perform this logic:

```
TypeScript
function getHalvingPeriod(uint256 blockNumber) public view returns (uint256) {
    return (blockNumber - deploymentBlock) / BLOCKS_PER_HALVING;
```

```
}
```

As a result, `getCurrentBlockReward()` may return a reward based on an outdated halving period, especially when the `currentHalvingPeriod` variable hasn't been updated through another internal call.

While this does not pose a direct security risk in the current implementation (since any call that uses `getCurrentBlockReward()` for actual minting will have updated `currentHalvingPeriod` beforehand), it does lead to inaccurate reward data being returned to any external readers or integrators that rely on the public `getCurrentBlockReward()` or the raw `currentHalvingPeriod` variable.

Recommendation

1. Make `currentHalvingPeriod` internal so external consumers are forced to call `getHalvingPeriod()` to obtain an up-to-date halving period.
2. Modify `getCurrentBlockReward()` to first calculate the up-to-date halving period dynamically by calling `getHalvingPeriod()` internally, e.g.:

```typescript
function getCurrentBlockReward() public view returns (uint256) {
    if (totalEmitted >= MAX_SUPPLY) return 0;
    uint256 currentHalvingPeriod= getHalvingPeriod(block.number);
    uint256 baseReward = REWARD_PER_THREE_BLOCKS / 3;
    uint256 reward = baseReward >> currentHalvingPeriod;
    uint256 remainingSupply = MAX_SUPPLY - totalEmitted;
    return reward > remainingSupply ? remainingSupply : reward;
}
```

IMM-MED-02

Integer Division Truncation in `LiquidStakingPool` Leads to Loss of User Funds #17

Id	IMM-MED-02
Severity	Medium
Category	Bug
Status	Fixed in https://github.com/matchain/contracts/pull/12

Description

The `LiquidStakingPool` contract contains an integer division truncation vulnerability in its staking mechanism. When users stake small amounts of MAT tokens while the exchange rate is high, the calculation of stMAT tokens can result in zero tokens due to solidity's integer division truncation. This causes users to lose their staked MAT tokens without receiving any stMAT tokens in return.

The issue occurs in the `stake` function when calculating how many stMAT tokens to mint:

```
TypeScript
uint256 stMATAmount = (amount * 1e18) / exchangeRate();
```

If `exchangeRate()` is significantly higher than `amount`, the division will result in a value less than 1, which gets truncated to 0 in integer division. The contract will then proceed to take the user's MAT tokens but mint 0 stMAT tokens to them.

As the protocol matures and accumulates rewards, the exchange rate naturally increases over time, making this vulnerability more likely to affect users with smaller stake amounts.

Proof of Concept

We created a test that demonstrates this vulnerability:

1. We established a high exchange rate (450:1) by simulating accumulated rewards
2. A user attempted to stake 400 wei of MAT tokens
3. The calculation (400 * 1e18) / 450e18 = 0 due to integer truncation
4. The user's MAT tokens were transferred, but they received 0 stMAT tokens

The test output clearly shows:

```
TypeScript
Initial exchange rate: 1
...
New exchange rate: 450
Expected stMAT tokens for 400 wei at current rate: 0
Will this result in zero tokens due to truncation? Yes
Small staker MAT balance before: 2000
Small staker stMAT balance before: 0
Small staker MAT balance after: 1600
Small staker stMAT balance after: 0
```

```
TypeScript

// SPDX-License-Identifier: UNLICENSED
pragma solidity ^0.8.10;

import "forge-std/Test.sol";
import "forge-std/console.sol";
import "../../contracts/MATToken.sol";
import "../../contracts/MatchConfig.sol";
import "../../contracts/StakingPool.sol";
import "../../contracts/LiquidStakingPool.sol";
import "../../contracts/StakingPoolFactory.sol";
import "@openzeppelin/contracts/proxy/beacon/UpgradeableBeacon.sol";
import "@openzeppelin/contracts/proxy/beacon/BeaconProxy.sol";
import "@openzeppelin/contracts-upgradeable/proxy/utils/Initializable.sol";


contract LiquidStakingPoolTest is Test {
    // Contracts
    MATToken public token;
    MatchConfig public config;
    StakingPoolFactory public factory;
    UpgradeableBeacon public stakingPoolBeacon;
    UpgradeableBeacon public liquidStakingPoolBeacon;
    StakingPool public stakingPoolImplementation;
    LiquidStakingPool public liquidStakingPoolImplementation;
    StakingPool public stakingPool;
    LiquidStakingPool public liquidStakingPool;
```

```
// Accounts for testing
address deployer = address(1);
address poolOwner = address(2);
address staker1 = address(3);
address staker2 = address(4);
address smallStaker = address(5);
address feeVault = address(6);
address attacker = address(7);

// Constants
uint256 constant INITIAL_MAT_SUPPLY = 10_000_000 * 1e18;
uint256 constant LARGE_STAKE = 1_000_000 * 1e18;
uint256 constant INITIAL_SELF_STAKE = 250_000 * 1e18;
uint256 constant MIN_POOL_SELFSTAKE = 250_000 * 1e18;
uint256 constant MIN_LOCK_PERIOD_DAYS = 730; // 2 years
uint256 constant MIN_VESTING_PERIOD_DAYS = 1095; // 3 years

    // Flash unstake fee (5%)
uint256 constant FLASH_UNSTAKE_FEE = 500; // 5% fee
uint256 constant FLASH_UNSTAKE_AMOUNT = 100_000 * 1e18; // 100,000 MAT

function setUp() public {
    // Configure accounts with funds
    vm.deal(deployer, 100 ether);
    vm.deal(poolOwner, 10 ether);
    vm.deal(staker1, 10 ether);
    vm.deal(staker2, 10 ether);
    vm.deal(smallStaker, 1 ether);
    vm.deal(attacker, 10 ether);

    vm.startPrank(deployer);

    // Deploy MatchConfig
    config = new MatchConfig();
    config.initialize(deployer);

    // Deploy MATToken
    token = new MATToken();
    token.initialize("Matchain Token", "MAT", deployer, address(config));
```

```
        // Deploy StakingPool implementation
        stakingPoolImplementation = new StakingPool();

        // Create Beacon for StakingPool
        stakingPoolBeacon = new UpgradeableBeacon(address(stakingPoolImplementation),
deployer);

        // Deploy LiquidStakingPool implementation
        liquidStakingPoolImplementation = new LiquidStakingPool();

        // Create Beacon for LiquidStakingPool
        liquidStakingPoolBeacon = new
UpgradeableBeacon(address(liquidStakingPoolImplementation), deployer);

        // Deploy Factory
        factory = new StakingPoolFactory();
        factory.initialize(address(config), address(stakingPoolBeacon));

        // Configure addresses in MatchConfig
        config.setToken(address(token));
        config.setPoolFactory(address(factory));

        // Distribute initial tokens
        token.transfer(poolOwner, INITIAL_MAT_SUPPLY); // For self-stake
        token.transfer(staker1, LARGE_STAKE);
        token.transfer(staker2, LARGE_STAKE);
        token.transfer(smallStaker, 1000); // Small amount for testing

        vm.stopPrank();

        // Create a staking pool
        address poolAddress = createPool(poolOwner);
        stakingPool = StakingPool(poolAddress);

        // Deploy LiquidStakingPool via BeaconProxy
        vm.startPrank(deployer);

        // Create a BeaconProxy for LiquidStakingPool
        bytes memory initData = abi.encodeWithSelector(
            LiquidStakingPool.initialize.selector,
            address(token),
```

```
            address(stakingPool),
            deployer,
            feeVault
        );

        address liquidProxy = address(new BeaconProxy(
            address(liquidStakingPoolBeacon),
            initData
        ));

        liquidStakingPool = LiquidStakingPool(liquidProxy);
        vm.stopPrank();
    }

function testSmallStakeWithHighExchangeRate() public {
        // 1. First set up a high exchange rate by adding some stake and rewards
        vm.startPrank(staker1);
        token.approve(address(liquidStakingPool), LARGE_STAKE);
        liquidStakingPool.stake(LARGE_STAKE);
        vm.stopPrank();

        // Get initial exchange rate (should be 1:1)
        uint256 initialExchangeRate = liquidStakingPool.exchangeRate();
        console.log("Initial exchange rate:", initialExchangeRate / 1e18);
        assertEq(initialExchangeRate, 1e18, "Initial exchange rate should be 1:1");

        // Give smallStaker tokens
        vm.startPrank(deployer);
        token.transfer(smallStaker, 1000);
        vm.stopPrank();

        // 2. Simulate rewards to increase exchange rate significantly
        // StakingPool.stakes mapping is at slot 2
        // LiquidStakingPool.lastRecordedStake is at slot 5

        uint256 totalSupply = liquidStakingPool.totalSupply();
        console.log("Total supply of stMAT:", totalSupply);

        // Target a very high exchange rate (500:1) to demonstrate the truncation issue
        uint256 targetExchangeRate = 500e18;
```

```
        // Calculate target stake to create that exchange rate:
        // exchangeRate = currentStake / totalSupply
        // So currentStake = exchangeRate * totalSupply
        uint256 targetStakedAmount = (targetExchangeRate * totalSupply) / 1e18;
        console.log("Target staked amount to create high exchange rate:",
targetStakedAmount);

        vm.startPrank(deployer);

        // 1. First set the actual stake in StakingPool mapping
        // The slot for the mapping is computed with: keccak256(abi.encode(key,
uint256(slot)))
        bytes32 stakesSlot = keccak256(abi.encode(address(liquidStakingPool), uint256(2)));
        vm.store(
            address(stakingPool),
            stakesSlot,
            bytes32(targetStakedAmount)
        );

        // 2. Update totalStakedAmount in StakingPool (slot 5)
        vm.store(
            address(stakingPool),
            bytes32(uint256(5)),
            bytes32(targetStakedAmount)
        );

        // 3. Leave lastRecordedStake in LiquidStakingPool unchanged
        vm.stopPrank();

        // Force update of the exchange rate by calling it
        uint256 newExchangeRate = liquidStakingPool.exchangeRate();

        // Verify the new exchange rate is high enough
        console.log("New exchange rate:", newExchangeRate / 1e18);
        assertGe(newExchangeRate, 100e18, "Exchange rate should be high");

        // Calculate what happens with a small stake amount at this exchange rate
        uint256 smallAmount = 400; // wei
        uint256 expectedStMAT = (smallAmount * 1e18) / newExchangeRate;
        console.log("Expected stMAT tokens for 400 wei at current rate:", expectedStMAT);
        console.log("Will this result in zero tokens due to truncation?", expectedStMAT == 0
```

```
? "Yes" : "No");
        assertEq(expectedStMAT, 0, "Small amount should result in zero stMAT tokens due to
truncation");


        // 3. Now attempt to stake a small amount that will result in 0 stMAT due to
truncation
        vm.startPrank(smallStaker);
        token.approve(address(liquidStakingPool), 400);

        // Record balances before staking
        uint256 matBalanceBefore = token.balanceOf(smallStaker);
        uint256 stMatBalanceBefore = liquidStakingPool.balanceOf(smallStaker);

        console.log("Small staker MAT balance before:", matBalanceBefore);
        console.log("Small staker stMAT balance before:", stMatBalanceBefore);

        // Stake a small amount
        // For example, if exchange rate is 500e18, then staking 400 wei would result in:
        // stMATAmount = (400 * 1e18) / 500e18 = 0 due to integer division truncation
        liquidStakingPool.stake(400);

        // 4. Verify that MAT was taken but no stMAT was received
        uint256 matBalanceAfter = token.balanceOf(smallStaker);
        uint256 stMatBalanceAfter = liquidStakingPool.balanceOf(smallStaker);

        console.log("Small staker MAT balance after:", matBalanceAfter);
        console.log("Small staker stMAT balance after:", stMatBalanceAfter);

        // Assertions
        assertEq(matBalanceAfter, matBalanceBefore - 400, "MAT should be deducted");
        assertEq(stMatBalanceAfter, 0, "No stMAT should be received due to truncation");
    }

    function createPool(address owner) internal returns (address) {
        // Approve factory to spend owner's tokens
        vm.startPrank(owner);
        token.approve(address(factory), MIN_POOL_SELFSTAKE);

        // Create pool
        factory.mintPool(
```

```
            owner,
            "ipfs://testpool",
            MIN_LOCK_PERIOD_DAYS,    // 2 years of lock
            MIN_VESTING_PERIOD_DAYS  // 3 years of vesting
        );
        vm.stopPrank();

        // Get the address of the new pool
        // Count pools manually
        uint256 poolCount = 0;
        bool hasMorePools = true;
        while (hasMorePools) {
            try token._pools(poolCount) returns (address) {
                poolCount++;
            } catch {
                hasMorePools = false;
            }
        }
        address poolAddress = token._pools(poolCount - 1);

        return poolAddress;
    }
```

Recommendation

We recommend implementing a minimum stake amount check in the `stake` function:

TypeScript

```
function stake(uint256 amount) external {
    if (amount == 0) revert InsufficientAmount();

    uint256 currentRate = exchangeRate();
    uint256 stMATAmount = (amount * 1e18) / currentRate;

    // Add this check to prevent truncation loss
    if (stMATAmount == 0) revert StakeTooSmall(amount, currentRate);

    bool success = matToken.transferFrom(msg.sender, address(this), amount);
    if (!success) revert TransferFailed();
```

```
    stakingPool.stake(amount);
    _mint(msg.sender, stMATAmount);

    emit Staked(msg.sender, amount, stMATAmount);
}
```

IMM-MED-03

Unclaimable Dust May Accumulate and Lock Tokens #18

Id	IMM-MED-03
Severity	Medium
Category	Bug
Status	Acknowledged

Description

In the `claimDust()` function, the following condition is enforced:

```
TypeScript
require(totalEmitted + dust <= MAX_SUPPLY, "Max supply exceeded");
```

If this function is not called periodically, the dust variable may grow over time due to repeated accumulation within the `rewardDistribution()` function:

```
TypeScript
if (dustAmount > 0) {
    dust += dustAmount;
}
```

This dust is the leftover remainder from reward allocation rounding, and while it is stored for later minting, it is only claimable by the owner via `claimDust()`. If dust grows large enough such that `totalEmitted + dust > MAX_SUPPLY`, the required condition in `claimDust()` will revert permanently, locking the accumulated dust and preventing its recovery, effectively causing a denial-of-service on this portion of the supply.

This creates a scenario where tokens that should have been minted and distributed remain forever unclaimable due to an avoidable overflow condition.

Recommendation

There are several options:

1. Ensure dust is never allowed to grow beyond `MAX_SUPPLY` - totalEmitted during accumulation.

2. Automatic Dust Distribution: Instead of relying on manual `claimDust()`, integrate dust distribution directly into `rewardDistribution()` by redistributing leftovers in the next cycle. This avoids centralization of dust collection and prevents accumulation beyond reclaimable limits.

IMM-MED-04

Incomplete fee accounting during pool migration leads to protocol revenue loss #19

Id	IMM-MED-04
Severity	Medium
Category	Bug
Status	Fixed in https://github.com/matchain/contracts/pull/11

Description

The `LiquidStakingPool` contract fails to properly account for rewards generated during the unbonding period of a pool migration. When a migration is initiated via `changeStakingPool()`, the contract unstakes all funds from the current pool and creates a pending migration. After the unbonding period, `completeMigration()` stakes the available funds in the new pool and resets `lastRecordedStake` to the current stake amount.

However, the contract does not apply fee calculations to rewards that accumulate during the unbonding period. This creates a gap in fee accounting where rewards generated during this period (typically 2 weeks) completely bypass the protocol's fee system.

The issue occurs because:

1. `changeStakingPool()` calls `exchangeRate()` to update fee accounting before unstaking
2. `completeMigration()` resets `lastRecordedStake` to the current stake in the new pool
3. No fee calculation is performed on rewards accrued during the unbonding period

For a protocol with significant TVL, this can result in substantial fee revenue loss with each migration. For example, with $10M TVL and 5% APY, approximately $1,923 in fee revenue would be lost during a single 2-week migration period (assuming 10% fee rate).

Recommendation

Track Unbonding Rewards: Modify the `completeMigration()` function to track and account for rewards generated during the unbonding period:

TypeScript

```
function completeMigration() external onlyOwner {
    // Existing checks...

    stakingPool.safeClaimUnstakes();

    // Calculate rewards generated during unbonding
    uint256 currentBalance = matToken.balanceOf(address(this));
    uint256 reservedBalance = flashPool + totalPendingClaims;
    uint256 availableBalance = currentBalance > reservedBalance ?
                            currentBalance - reservedBalance : 0;

    // If we received more than the original unstaked amount, the difference is rewards
    if (availableBalance > pendingMigration.amount) {
        uint256 unbondingRewards = availableBalance - pendingMigration.amount;
        uint256 feeAmount = (unbondingRewards * lspFee) / 10000;
        accumulatedFees += feeAmount;
    }

    // Continue with existing migration logic...
}
```

IMM-MED-05

Integer Division Rounding in Reward Distribution Enables Malicious Reward Redirection and Protocol Sabotage #26

Id	IMM-MED-05
Severity	Medium
Category	Bug
Status	Acknowledged

Description

The MatChain protocol's reward distribution mechanism contains a critical vulnerability where integer division rounding errors can be exploited to divert rewards from legitimate stakers to the dust accumulator. A malicious actor can exploit this vulnerability by frequently calling the `rewardDistribution()` function with small block intervals, causing nearly all rewards to be redirected as dust that only the contract owner can claim.

The vulnerability stems from the integer division operations in the reward distribution calculations. When rewards are distributed in small amounts (per block), the integer division rounds down to zero for individual pool rewards, causing all rewards to accumulate as dust rather than being distributed to stakers.

This creates two significant issues:

Legitimate stakers receive significantly fewer rewards than they should
The dust accumulator, which is only accessible to the contract owner, collects an unfair proportion of rewards
In the worst-case scenario demonstrated by our PoC, a malicious actor can cause 100% of rewards to be diverted to dust by calling the distribution function every block.

PoC:

```
TypeScript
// SPDX-License-Identifier: UNLICENSED
```

```
pragma solidity ^0.8.10;

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

/// @title Reward Distribution Malicious Exploitation PoC
/// @notice This test demonstrates how a malicious actor can exploit the reward distribution
///         mechanism to redirect rewards from legitimate stakers to the dust accumulator
///         that only the owner can claim.
contract RewardDistributionAttackTest is Test {
    // Constants for testing
    uint256 constant TOTAL_POOLS = 3;
    uint256 constant BLOCK_REWARD = 3; // Small reward to demonstrate the exploit
    uint256 constant ATTACK_FREQUENCY = 10; // Number of blocks to simulate in attack
    uint256 constant NORMAL_FREQUENCY = 1000; // Normal expected distribution frequency

    // Pool stake amounts
    uint256 constant POOL1_STAKE = 1_000_000 * 10**18; // 1M tokens
    uint256 constant POOL2_STAKE = 2_000_000 * 10**18; // 2M tokens
    uint256 constant POOL3_STAKE = 3_000_000 * 10**18; // 3M tokens

    // Test variables
    uint256 normalDust;
    uint256 attackDust;
    uint256[3] normalPoolRewards;
    uint256[3] attackPoolRewards;
    uint256 totalNormalPoolRewards;
    uint256 totalAttackPoolRewards;

    function setUp() public {
        // No setup needed for this test
    }

    /// @notice Test to demonstrate how a malicious actor can exploit the reward distribution
    function testMaliciousExploitation() public {
        // Scenario 1: Normal operation - distribution every 1000 blocks
        uint256 normalTotalReward = BLOCK_REWARD * NORMAL_FREQUENCY;

        // Get pool stakes in an array for easier processing
        uint256[3] memory poolStakes = [POOL1_STAKE, POOL2_STAKE, POOL3_STAKE];
```

```
        // Run normal distribution simulation
        (normalPoolRewards, normalDust) =
            simulateSingleDistribution(normalTotalReward, poolStakes);

        // Calculate total rewards for normal distribution
        totalNormalPoolRewards =
            normalPoolRewards[0] +
            normalPoolRewards[1] +
            normalPoolRewards[2];

        // Scenario 2: Attack scenario - malicious actor calls distribution every block for
10 blocks
        // This simulates an attack where someone calls the function very frequently
        (attackPoolRewards, attackDust) =
            simulateAttack(BLOCK_REWARD, ATTACK_FREQUENCY, poolStakes);

        // Calculate total rewards during attack
        totalAttackPoolRewards =
            attackPoolRewards[0] +
            attackPoolRewards[1] +
            attackPoolRewards[2];

        // Log results
        logResults();

        // Assert that attack creates more dust proportionally
        assertGt(
            attackDust * NORMAL_FREQUENCY,
            normalDust * ATTACK_FREQUENCY,
            "Attack should create proportionally more dust due to rounding"
        );

        // Assert that pools receive proportionally less rewards during attack
        assertLt(
            totalAttackPoolRewards * NORMAL_FREQUENCY,
            totalNormalPoolRewards * ATTACK_FREQUENCY,
            "Pools should receive proportionally less rewards during attack"
        );
    }

    /// @notice Simulates a single reward distribution after multiple blocks
```

```
/// @param totalReward Total reward amount to distribute.
/// @param poolStakes Array of stake amounts for each pool.
/// @return poolRewards Array of rewards distributed to each pool.
/// @return dust Amount of dust accumulated.
function simulateSingleDistribution(
    uint256 totalReward,
    uint256[3] memory poolStakes
) internal pure returns (uint256[3] memory poolRewards, uint256 dust) {
    uint256 totalDistributed = 0;
    uint256 totalStaked = poolStakes[0] + poolStakes[1] + poolStakes[2];

    // Calculate owner rewards (equal split among pool owners)
    for (uint256 i = 0; i < TOTAL_POOLS; i++) {
        uint256 ownerAmount = totalReward / (TOTAL_POOLS * 2);
        poolRewards[i] += ownerAmount;
        totalDistributed += ownerAmount;
    }

    // Calculate pool rewards (proportional to stake)
    for (uint256 i = 0; i < TOTAL_POOLS; i++) {
        uint256 poolAmount = ((totalReward / 2) * poolStakes[i]) / totalStaked;
        poolRewards[i] += poolAmount;
        totalDistributed += poolAmount;
    }

    // Calculate dust (undistributed amount due to rounding)
    dust = totalReward > totalDistributed ? totalReward - totalDistributed : 0;

    return (poolRewards, dust);
}

/// @notice Simulates an attack where distribution is called frequently.
/// @param blockReward Reward amount per block.
/// @param attackBlocks Number of blocks in the attack.
/// @param poolStakes Array of stake amounts for each pool.
/// @return poolRewards Array of rewards distributed to each pool.
/// @return totalDust Total amount of dust accumulated.
function simulateAttack(
    uint256 blockReward,
    uint256 attackBlocks,
    uint256[3] memory poolStakes
```

```
) internal pure returns (uint256[3] memory poolRewards, uint256 totalDust) {
    uint256 totalStaked = poolStakes[0] + poolStakes[1] + poolStakes[2];

    // Simulate distribution for each block individually (malicious frequent calls)
    for (uint256 b = 0; b < attackBlocks; b++) {
        uint256 distributedThisBlock = 0;

        // Calculate owner rewards for this block
        for (uint256 i = 0; i < TOTAL_POOLS; i++) {
            uint256 ownerAmount = blockReward / (TOTAL_POOLS * 2);
            poolRewards[i] += ownerAmount;
            distributedThisBlock += ownerAmount;
        }

        // Calculate pool rewards for this block
        for (uint256 i = 0; i < TOTAL_POOLS; i++) {
            uint256 poolAmount = ((blockReward / 2) * poolStakes[i]) / totalStaked;
            poolRewards[i] += poolAmount;
            distributedThisBlock += poolAmount;
        }

        // Calculate dust for this block
        uint256 dustThisBlock = blockReward > distributedThisBlock ?
            blockReward - distributedThisBlock : 0;

        totalDust += dustThisBlock;
    }

    return (poolRewards, totalDust);
}

/// @notice Logs the test results.
function logResults() internal view {
    console.log("=== Normal Distribution Results (Every 1000 blocks) ===");
    console.log("Blocks passed:", NORMAL_FREQUENCY);
    console.log("Total reward amount:", BLOCK_REWARD * NORMAL_FREQUENCY);
    console.log("Pool 1 reward (1M stake):", normalPoolRewards[0]);
    console.log("Pool 2 reward (2M stake):", normalPoolRewards[1]);
    console.log("Pool 3 reward (3M stake):", normalPoolRewards[2]);
    console.log("Total pool rewards:", totalNormalPoolRewards);
    console.log("Dust accumulated:", normalDust);
```

```
        console.log("");

        console.log("=== Attack Scenario Results (Every block for 10 blocks) ===");
        console.log("Blocks passed:", ATTACK_FREQUENCY);
        console.log("Total reward amount:", BLOCK_REWARD * ATTACK_FREQUENCY);
        console.log("Pool 1 reward (1M stake):", attackPoolRewards[0]);
        console.log("Pool 2 reward (2M stake):", attackPoolRewards[1]);
        console.log("Pool 3 reward (3M stake):", attackPoolRewards[2]);
        console.log("Total pool rewards:", totalAttackPoolRewards);
        console.log("Dust accumulated:", attackDust);
        console.log("");

        console.log("=== Comparison (Normalized per block) ===");
        console.log("Dust per block in normal scenario:", normalDust / NORMAL_FREQUENCY);
        console.log("Dust per block in attack scenario:", attackDust / ATTACK_FREQUENCY);

        // Calculate efficiency (rewards that reach pools vs total rewards)
        console.log("Normal distribution efficiency: %",
            totalNormalPoolRewards * 100 / (BLOCK_REWARD * NORMAL_FREQUENCY));
        console.log("Attack scenario efficiency: %",
            totalAttackPoolRewards * 100 / (BLOCK_REWARD * ATTACK_FREQUENCY));

        // Calculate dust percentage
        console.log("Normal dust percentage: %",
            normalDust * 100 / (BLOCK_REWARD * NORMAL_FREQUENCY));
        console.log("Attack dust percentage: %",
            attackDust * 100 / (BLOCK_REWARD * ATTACK_FREQUENCY));

        // Calculate attack impact
        console.log("");
        console.log("=== Attack Impact ===");
        console.log("Increase in dust percentage: %",
            (attackDust * 100 / (BLOCK_REWARD * ATTACK_FREQUENCY)) -
            (normalDust * 100 / (BLOCK_REWARD * NORMAL_FREQUENCY)));

        // Calculate per-pool impact (normalized to same reward amount)
        console.log("");
        console.log("=== Per-Pool Impact (normalized) ===");

        // Calculate what pools would receive in attack scenario if same total rewards
        uint256 normalizedAttackPool1 = attackPoolRewards[0] * NORMAL_FREQUENCY /
```

```
ATTACK_FREQUENCY;
        uint256 normalizedAttackPool2 = attackPoolRewards[1] * NORMAL_FREQUENCY /
ATTACK_FREQUENCY;
        uint256 normalizedAttackPool3 = attackPoolRewards[2] * NORMAL_FREQUENCY /
ATTACK_FREQUENCY;

        console.log("Pool 1 reward loss: %",
            normalPoolRewards[0] > normalizedAttackPool1 ?
            (normalPoolRewards[0] - normalizedAttackPool1) * 100 / normalPoolRewards[0] : 0);
        console.log("Pool 2 reward loss: %",
            normalPoolRewards[1] > normalizedAttackPool2 ?
            (normalPoolRewards[1] - normalizedAttackPool2) * 100 / normalPoolRewards[1] : 0);
        console.log("Pool 3 reward loss: %",
            normalPoolRewards[2] > normalizedAttackPool3 ?
            (normalPoolRewards[2] - normalizedAttackPool3) * 100 / normalPoolRewards[2] : 0);
    }
}
```

PoC explanation:
This test demonstrates the vulnerability by comparing two scenarios:

1. ***Normal Operation***: Rewards distributed every 1000 blocks (3000 total rewards)
2. ***Attack Scenario***: Rewards distributed every block for 10 blocks (30 total rewards)

The test results show that in the normal operation scenario, rewards are distributed correctly to all pools with 0% dust. However, in the attack scenario, 100% of rewards go to dust and 0% reach the staking pools.

The key functions in the PoC are:

- `simulateSingleDistribution()`: Simulates a single reward distribution after multiple blocks (normal case)
- `simulateAttack()`: Simulates frequent reward distributions, one block at a time (attack case)

The test output shows:

- Normal distribution efficiency: 100%
- Attack scenario efficiency: 0%
- Pool reward loss: 100% for all pools

```
TypeScript
Ran 1 test for test/foundry/RewardDistributionAttack.t.sol:RewardDistributionAttackTest
[PASS] testMaliciousExploitation() (gas: 177013)
Logs:
  === Normal Distribution Results (Every 1000 blocks) ===
  Blocks passed: 1000
  Total reward amount: 3000
  Pool 1 reward (1M stake): 750
  Pool 2 reward (2M stake): 1000
  Pool 3 reward (3M stake): 1250
  Total pool rewards: 3000
  Dust accumulated: 0

  === Attack Scenario Results (Every block for 10 blocks) ===
  Blocks passed: 10
  Total reward amount: 30
  Pool 1 reward (1M stake): 0
  Pool 2 reward (2M stake): 0
  Pool 3 reward (3M stake): 0
  Total pool rewards: 0
  Dust accumulated: 30

  === Comparison (Normalized per block) ===
  Dust per block in normal scenario: 0
  Dust per block in attack scenario: 3
  Normal distribution efficiency: % 100
  Attack scenario efficiency: % 0
  Normal dust percentage: % 0
  Attack dust percentage: % 100

  === Attack Impact ===
  Increase in dust percentage: % 100

  === Per-Pool Impact (normalized) ===
  Pool 1 reward loss: % 100
  Pool 2 reward loss: % 100
  Pool 3 reward loss: % 100
```

This proves that a malicious actor can completely subvert the reward distribution mechanism by calling it frequently with small block intervals, causing all rewards to be redirected to dust.

Recommendation

To mitigate this vulnerability, we recommend implementing the following changes:

1. **Enforce Minimum Distribution Intervals**:
    a. Implement a minimum number of blocks (e.g., 100) that must pass between reward distributions
    b. Store the last distribution block and revert if the minimum interval hasn't passed

2. **Use Fixed-Point Arithmetic**:
    a. Replace integer division with fixed-point arithmetic to minimize rounding errors
    b. Store "remainder" values for each pool and include them in the next distribution

IMM-LOW-01

Missing call to `_disableInitializers` in several upgradeable contracts #1

Id	IMM-LOW-01
Severity	LOW
Category	Bug
Status	Acknowledged

Description

Several upgradeable contracts `MATToken`, `MatchConfig`, `StakingPoolFactory` and `StakingPool` do not implement the `_disableInitializers()` function in their constructors, while others `LiquidStakingPool.sol` and `FeeDistributionVault.sol` do implement it correctly.

This inconsistency creates a security vulnerability where the implementation contracts of `MATToken` and `StakingPool` could potentially be initialized directly by an attacker.

When an upgradeable contract doesn't call `_disableInitializers()` in its constructor, the implementation contract itself remains initializable.

If an attacker initializes the implementation contract directly (not through the proxy), they could potentially gain control over it.

This is a known security issue with upgradeable contracts that OpenZeppelin explicitly warns about in their documentation.

The proper pattern, as seen in `LiquidStakingPool.sol` and `FeeDistributionVault.sol`, is:

```
TypeScript
/// @custom:oz-upgrades-unsafe-allow constructor
constructor() {
    _disableInitializers();
}
```

Recommendation

Add a constructor with `_disableInitializers()` to all upgradeable contracts in the protocol, specifically to `MATToken` and `StakingPool`:

```
TypeScript
/// @custom:oz-upgrades-unsafe-allow constructor
constructor() {
    _disableInitializers();
}
```

IMM-LOW-02

Potential Denial of Service Risk in `FeeDistributionVault.sol` Due to Unbounded Loops #4

Id	IMM-LOW-02
Severity	LOW
Category	Bug
Status	Fixed in https://github.com/matchain/contracts/pull/18

Description

The `FeeDistributionVault.sol` smart contract uses for loops across multiple functions to iterate over the list of beneficiaries. While this approach is straightforward, it introduces a potential Denial of Service (DoS) vulnerability if the beneficiaries array becomes too large.

In such a case, executing these loops could consume more gas than the block limit allows, causing the transaction to revert. As a result, critical contract functionality could become inaccessible until a smaller set of beneficiaries is manually set.

Recommendation

There are two options:

1. Introduce a function to remove beneficiaries individually, to avoid having to reset the entire set from scratch when adjustments are needed.
2. Consider implementing a pull-based pattern, where beneficiaries manually claim their share of tokens, rather than the contract pushing tokens to all beneficiaries in a single call. This would eliminate the risk of gas exhaustion in a single transaction and make the system more scalable.

IMM-LOW-03

Missing stakers Counter Decrement in Ownership Transfer Leads to Inaccurate Staker Accounting #9

Id	IMM-LOW-03
Severity	LOW
Category	Bug
Status	Fixed in https://github.com/matchain/contracts/pull/16

Description

When ownership of a `StakingPool` is transferred to an address that already has a stake in the pool, the contract correctly consolidates the new owner's regular stake into the `selfStake` variable and zeros out their entry in the `stakes` mapping. However, the contract fails to decrement the `stakers` counter, which tracks the number of unique addresses with active stakes.

This occurs in the `_transferOwnership` function:

```
TypeScript
function _transferOwnership(address newOwner) internal virtual override {
    address oldOwner = owner();
    if (oldOwner == newOwner) return;
    ownershipNFT.transferFrom(oldOwner, newOwner, uint160(address(this)));
    if (stakes[newOwner] > 0) {
        selfStake += stakes[newOwner];
        stakes[newOwner] = 0;
        // Missing: stakers--;
    }
    emit OwnershipTransferred(oldOwner, newOwner);
}
```

While the comment in the contract indicates that the `stakers` variable is "for information only," maintaining accurate protocol metrics is important for transparency and governance decisions. Additionally, if the protocol ever begins to rely on this counter for any functionality, the inaccuracy could lead to unexpected behavior.

Recommendation

Decrement the `stakers` counter when consolidating a new owner's stake:

```typescript
if (stakes[newOwner] > 0) {
    selfStake += stakes[newOwner];
    stakes[newOwner] = 0;
    stakers--; // Add this line
}
```

IMM-LOW-04

`moveStakeToSelfStake` Does Not Decrease stakers Count #12

Id	IMM-LOW-04
Severity	LOW
Category	Bug
Status	Fixed in https://github.com/matchain/contracts/pull/16

Description

When a user transfers their `PoolOwnership` NFT by calling `transferFrom()`, the NFT contract internally invokes:

```
TypeScript
function moveStakeToSelfStake(address staker) public {
    require(msg.sender == address(ownershipNFT), "Only NFT contract can call");
    if (stakes[staker] > 0) {
        selfStake += stakes[staker];
        stakes[staker] = 0;
    }
}
```

While the function correctly transfers the user's stake into selfStake and clears the user's individual stake record, it does not decrement the stakers counter, which is used to track the number of unique addresses with a non-zero stake.

As a result, if a user holding stake transfers their ownership NFT, their stake is zeroed out, but the system still counts them as an active staker.

- The stakers variable becomes inaccurate, reflecting a higher count than the actual number of unique stakers.
- This can affect analytics, on-chain or off-chain reward calculations, frontend representations, and any logic relying on the true count of active stakers.

Recommendation

Update the moveStakeToSelfStake function to ensure the stakers count is decremented when a non-zero stake is moved:

```TypeScript
function moveStakeToSelfStake(address staker) public {
    require(msg.sender == address(ownershipNFT), "Only NFT contract can call");
    if (stakes[staker] > 0) {
        selfStake += stakes[staker];
        stakes[staker] = 0;
        stakers--; // ✅ Adjust stakers count accurately
    }
}
```

IMM-LOW-05

Comparison in Unbonding Period Verification Forces Users to Wait Extra Block Before Claiming Tokens #13

Id	IMM-LOW-05
Severity	LOW
Category	Bug
Status	Fixed in https://github.com/matchain/contracts/pull/15

Description

The `LiquidStakingPool` contract implements an incorrect condition check in the `claimUnstake()` function when verifying if unbonding periods have been satisfied. The function uses strict comparison operators (`>`) instead of inclusive comparison operators (`>=`), which extends the actual unbonding period by one block beyond what is intended.

```
TypeScript
uint256 blocksSinceRequest = block.number - request.requestBlock;

bool lspUnbondingComplete = (blocksSinceRequest > UNBONDING_PERIOD);
bool poolUnbondingComplete = (blocksSinceRequest > poolUnbondingPeriod);
```

When a user requests an unstake at block N with an unbonding period of X blocks:

- At block N+X (when the period should complete), the condition `X > X` evaluates to false
- Users can only claim at block N+X+1, extending the unbonding period by one block

This implementation error has the following impact:

1. Users who attempt to claim their tokens exactly at the end of the unbonding period will have their transactions revert with `UnbondingNotComplete()`, even though the period has technically elapsed.
2. The actual unbonding duration is extended by one block beyond what is specified by `UNBONDING_PERIOD` and `poolUnbondingPeriod` constants.

The issue affects all unstake claims in the protocol and represents a deviation from the intended functionality.

Recommendation

The comparison operators should be changed to inclusive comparisons to accurately reflect the intended unbonding period:

```
TypeScript
bool lspUnbondingComplete = (blocksSinceRequest >= UNBONDING_PERIOD);
bool poolUnbondingComplete = (blocksSinceRequest >= poolUnbondingPeriod);
```

IMM-LOW-06

Inconsistent Share Calculation in `calculateShareByAddress` for the Last Beneficiary by public view function #14

Id	IMM-LOW-06
Severity	LOW
Category	Bug
Status	Fixed in https://github.com/matchain/contracts/pull/14

Description

The function `calculateShareByAddress` in `FeeDistributionVault.sol` calculates the share of a given beneficiary based on their weight relative to `totalWeight`:

```
TypeScript
(totalAmount * beneficiary.weight) / totalWeight;
```

However, this logic diverges from the actual distribution logic implemented in the distribute function, specifically for the last beneficiary.

In distribute, the remaining balance (`remainingAmount = balance - distributedAmount`) is transferred to the last beneficiary to avoid dust (small leftover tokens caused by rounding errors):

```
TypeScript
uint256 remainingAmount = balance - distributedAmount;
matToken.safeTransfer(beneficiaries[beneficiaries.length - 1].addr, remainingAmount);
```

As a result, when `calculateShareByAddress` is called for the last beneficiary, it may return a value that does not match what will actually be transferred during distribution.

This discrepancy can mislead frontends, indexers, or other off-chain systems relying on `calculateShareByAddress` to predict actual payout amounts. Specifically, the last beneficiary might see a computed value that underestimates or overestimates their real share.

Recommendation

Update `calculateShareByAddress` to mirror the behavior of distributing for the last beneficiary. This could involve:

- Summing the calculated shares for all previous beneficiaries.
- Returning `totalAmount - sum(previousShares)` for the last beneficiary.

This change will ensure consistency between the view logic and the actual on-chain distribution, eliminating confusion or mismatches in expected payouts.

IMM-LOW-07

Incorrect Event Emitted in `setPoolFactory` Function #15

Id	IMM-LOW-07
Severity	LOW
Category	Bug
Status	Fixed in https://github.com/matchain/contracts/pull/13

Description

The `setPoolFactory` function in the `MatchConfig` contract incorrectly emits the `TokenChanged` event instead of the intended `PoolFactoryChanged` event.

This is a logical bug that could lead to inaccurate off-chain tracking of critical state changes.

```
TypeScript
function setPoolFactory(address _poolFactory) public onlyOwner {
    if (_poolFactory == poolFactory)
        return;
    address oldValue = poolFactory;
    poolFactory = _poolFactory;
    emit TokenChanged(oldValue, poolFactory); // <-- Incorrect event
}
```

- **Expected Behavior**: The function should emit the `PoolFactoryChanged` event to reflect the change in the `poolFactory` address.
- **Actual Behavior**: The `TokenChanged` event is emitted with the old and new `poolFactory` values, which is misleading and incorrect.

While this does not affect on-chain logic, it breaks event-based tracking for external systems such as frontends, subgraphs, indexers, or monitoring tools.

It may also cause confusion during audits or debugging.

Recommendation

Replace the emitted event with the correct one:

```
TypeScript
emit PoolFactoryChanged(oldValue, poolFactory);
```

IMM-LOW-08

Use of `blockhash(block.number)` Returns Zero in `generateSalt()` #30

Id	IMM-LOW-08
Severity	LOW
Category	Bug
Status	Fixed in https://github.com/matchain/contracts/pull/3

Description

The `generateSalt()` function is intended to produce a unique salt for use with CREATE2 by combining a blockhash, the caller's address, and a nonce:

```
TypeScript
function generateSalt() internal returns (bytes32) {
    saltNonce++;
    return keccak256(abi.encodePacked(blockhash(block.number), msg.sender, saltNonce));
}
```

However, `blockhash(block.number)` always returns zero per the EVM specification. The `blockhash()` opcode only returns a valid value for the previous 256 blocks, excluding the current block.

As a result, this line: `blockhash(block.number)`

Recommendation

Update the code to use `blockhash(block.number - 1)`, which is valid:

```
TypeScript
function generateSalt() internal returns (bytes32) {
    saltNonce++;
    return keccak256(abi.encodePacked(blockhash(block.number - 1), msg.sender, saltNonce));
}
```

IMM-INSIGHT-01

Typo on file name `FeeDistibutionVault` #2

Id	IMM-INSIGHT-01
Severity	INSIGHT
Category	Informational
Status	Fixed in https://github.com/matchain/contracts/pull/19

Description

The contract file is named `FeeDistibutionVault.sol` (missing an 'r' in "Distribution"), while the contract itself is correctly named `FeeDistributionVault`.

Recommendation

Rename the file from `FeeDistibutionVault.sol` to `FeeDistributionVault.sol` to match the actual contract name.

IMM-INSIGHT-02

Use of precomputed address to execute operations before actual deployment #7

Id	IMM-INSIGHT-02
Severity	INSIGHT
Category	Informational
Status	Acknowledged

Description

In the current implementation of `mintPool()`, the CREATE2-based address is precomputed using `Create2.computeAddress(...)`, and this address is immediately used in downstream logic (minting NFTs, registering pools, emitting events), before the proxy is actually deployed:

```
TypeScript
address poolAddress = Create2.computeAddress(...);
ownershipNFT.mintPoolNFT(poolOwner, poolAddress, initialTokenURI);
token.registerNewPool(poolAddress,  msg.sender);
...
BeaconProxy proxy = new BeaconProxy{salt: salt}(...);
```

Recommendation

Although it does not have a direct security impact, this issue is marked as informational because it is considered good practice to first deploy the contract and then use the deployed address.

IMM-INSIGHT-03

Unused custom error in `FeeDistributionVault` reduces code clarity and increases gas costs #28

Id	IMM-INSIGHT-03
Severity	INSIGHT
Category	Informational
Status	Fixed in https://github.com/matchain/contracts/pull/5

Description

The `FeeDistributionVault` contract defines a custom error `TransferFailed()` on line 49, but this error is never used anywhere in the contract implementation.

The contract uses OpenZeppelin's `SafeERC20` library for token transfers, which handles transfer failures internally by reverting with its own error messages.

Recommendation

Remove the unused `TransferFailed` error from the contract to improve code clarity and reduce bytecode size.

IMM-INSIGHT-04

State updates in `distribute()` function deviates from CEI pattern best practices #29

Id	IMM-INSIGHT-04
Severity	INSIGHT
Category	Informational
Status	Fixed in https://github.com/matchain/contracts/pull/4

Description

The `distribute()` function in the `FeeDistributionVault` contract interleaves external calls (`matToken.safeTransfer()`) with local state updates (`distributedAmount += share`). While this implementation is protected by the `nonReentrant` modifier, it deviates from the Checks-Effects-Interactions (CEI) pattern best practice.

Recommendation

Refactor the `distribute()` function to strictly follow the CEI pattern by calculating all shares first, then performing all external transfers afterward. This would improve code quality and reduce the risk of introducing vulnerabilities in future updates.