Marginal DAO

Security Review

Review by:
Kaden, Security Researcher

April 25, 2024

Contents

1 Introduction **2**
- 1.1 Disclaimer 2
- 1.2 Risk assessment 2
  - 1.2.1 Severity Classification 2

2 Security Review Summary **3**

3 Findings **4**
- 3.1 Medium Risk 4
  - 3.1.1 Loss of precision possible due to `rewardRate` rounding 4
  - 3.1.2 Insufficient validation of amount provided to `notifyRewardAmount` 4
  - 3.1.3 DoS with block gas limit 5
- 3.2 Low Risk 5
  - 3.2.1 Consider using a two-step ownership transfer pattern 5
- 3.3 Gas Optimization 6
  - 3.3.1 Redundant `require` statement 6
  - 3.3.2 Redundant on-chain data 6
  - 3.3.3 Use `uint256` event params in place of smaller variations 6
  - 3.3.4 Mark state variables as `immutable` and `constant` where relevant 8
  - 3.3.5 Cache storage variable used more than once 8
  - 3.3.6 Circumstantially redundant `Reward` struct members 8
- 3.4 Informational 9
  - 3.4.1 Failing test cases 9
  - 3.4.2 Zero amounts accepted in `lock` and `free` 10
  - 3.4.3 Use `UPPER_CASE` naming convention for constants 11
  - 3.4.4 Missing NatSpec comments 11
  - 3.4.5 Outdated compiler version 11
  - 3.4.6 Include `rewardsToken` as `RewardAdded` event parameter 12

1 Introduction

1.1 Disclaimer

A security review a detailed evaluation of the security posture of the code at a particular moment based on the information available at the time of the review. While the review endeavors to identify and disclose all potential security issues, it cannot guarantee that every vulnerability will be detected or that the code will be entirely secure against all possible attacks. The assessment is conducted based on the specific commit and version of the code provided. Any subsequent modifications to the code may introduce new vulnerabilities that were absent during the initial review. Therefore, any changes made to the code require a new security review to ensure that the code remains secure. Please be advised that a security review is not a replacement for continuous security measures such as penetration testing, vulnerability scanning, and regular code reviews.

1.2 Risk assessment

Severity	Description
Critical	*Must* fix as soon as possible (if already deployed).
High	Leads to a loss of a significant portion (>10%) of assets in the protocol, or significant harm to a majority of users.
Medium	Global losses <10% or losses to only a subset of users, but still unacceptable.
Low	Losses will be annoying but bearable. Applies to things like griefing attacks that can be easily repaired or even gas inefficiencies.
Gas Optimization	Suggestions around gas saving practices.
Informational	Suggestions around best practices or readability.

1.2.1 Severity Classification

The severity of security issues found during the security review is categorized based on the above table. Critical findings have a high likelihood of being exploited and must be addressed immediately. High findings are almost certain to occur, easy to perform, or not easy but highly incentivized thus must be fixed as soon as possible.

Medium findings are conditionally possible or incentivized but are still relatively likely to occur and should be addressed. Low findings a rare combination of circumstances to exploit, or offer little to no incentive to exploit but are recommended to be addressed.

Lastly, some findings might represent objective improvements that should be addressed but do not impact the project's overall security (Gas and Informational findings).

2 Security Review Summary

Marginal is a permissionless spot and perpetual exchange that enables leverage on assets with an Uniswap V3 Oracle.

The specific contracts in scope for this review were:

Contract	Additional comments
MarginalToken.sol	Forked from Uni.sol
MultiRewards.sol	Forked from Curve.fi multi rewards
MultiRewardsFactory.sol	Forked from Uniswap staking rewards factory but adapted to Curve.fi multi rewards
StakingPoints.sol	

From Apr 22nd to Apr 24th the security researchers conducted a review of dao on commit hash 5b67cea5. The team identified a total of **16** issues in the following risk categories:

- Critical Risk: 0
- High Risk: 0
- Medium Risk: 3
- Low Risk: 1
- Gas Optimizations: 6
- Informational: 6

3 Findings

3.1 Medium Risk

3.1.1 Loss of precision possible due to `rewardRate` rounding

Severity: Medium Risk

Context: MultiRewards.sol#L164-L166, MultiRewards.sol#L168-L176

Description: In `MultiRewards.notifyRewardAmount`, we compute the `rewardRate` given the amount of reward tokens to distribute divided by the `rewardsDuration`, e.g.:

```
rewardData[_rewardsToken].rewardRate = reward.div(
    rewardData[_rewardsToken].rewardsDuration
);
```

Since we're using integer math, when we divide, we round the result down to the nearest integer. In most circumstances, this is not a concern and results in only dust amounts of rewards to be left in the contract after they've been distributed. However, in some edge cases this can result in a much more significant loss.

For example, consider a circumstance where we use a low decimal `_rewardsToken`, e.g. GUSD (2 decimals), and a high `rewardsDuration`, e.g. 4 years (126,227,808 seconds). Any `reward` amount less than 1,262,278 GUSD would result in the `rewardRate` being rounded down to 0, permanently locking all the transferred GUSD.

We can also round down to non-zero amounts in less extreme circumstances which still result in some amount of tokens never being claimable. For example, with the same preconditions as above, a `reward` amount of 2,524,555 GUSD would give us: 252455500 / 126227808 = 1.999999081 which rounds down to a `rewardRate` of 1, resulting in about 50% of the GUSD being lost.

Recommendation: It's important that the amount of reward tokens to distribute is many multiples greater than the `rewardsDuration`. As such, it's recommended to include validation both that the `rewardsDuration` is not too high and that the `_rewardsToken` decimals are not too low.

A good option may be to implement a minimum `reward / rewardsDuration` multiple which would indicate the maximum possible relative precision loss, where if the minimum multiple is 1e6 then the maximum possible relative loss is <1/1e6. This should be enforced both in `MultiRewards` and `MultiRewardsFactory`, and in the `MultiRewardsFactory` it's important that the validation is implemented in `addReward` or else it may be possible to add rewards which can never be removed.

3.1.2 Insufficient validation of amount provided to `notifyRewardAmount`

Severity: Medium Risk

Context: MultiRewards.sol#L184-L189

Description: In `MultiRewards.notifyRewardAmount`, we validate that the `_rewardsToken` balance of the contract is sufficient to cover future reward emissions:

```
uint256 balance = IERC20(_rewardsToken).balanceOf(address(this));
require(
    rewardData[_rewardsToken].rewardRate <=
        balance.div(rewardData[_rewardsToken].rewardsDuration),
    "Provided reward too high"
);
```

While this does effectively validate that we have enough `_rewardsToken` balance to cover the future emissions, it fails to account for past emissions which have not yet been claimed. Since users must claim their emitted rewards by manually calling `getReward`, a significant portion of the `_rewardsToken` balance may be these unclaimed rewards. As a result, the above logic does not sufficiently validate that the contract balance will be sufficient to cover all reward emissions. This allows the invariant that there is sufficient `_rewardsToken` balance in the contract to cover all emissions to be broken, resulting in some users not receiving any rewards.

It's worth noting that under the expectation of only using this contract with `MultiRewardsFactory`, this check is somewhat redundant due to the fact that the amount transferred along with the `notifyRewardAmount` call is equal to the `reward` amount parameter:

```
IERC20(rewardsToken).safeTransfer(multiRewards, rewardAmount);
IMultiRewards(multiRewards).notifyRewardAmount(
    rewardsToken,
    rewardAmount
);
```

Recommendation: Rather than using the `safeTransfer` in `MultiRewardsFactory.notifyRewardAmount`, it's recommended that we instead use a `safeTransferFrom` of the `reward` amount in `MultiRewards.notifyRewardAmount` such that no matter how `MultiRewards` is used, we ensure that we transfer a sufficient amount to cover all reward emissions.

3.1.3 DoS with block gas limit

Severity: Medium Risk

Context: MultiRewards.sol#L201-L210

Description: `MultiRewards.updateReward` is responsible for reward accounting logic and is executed at the start of critical functions: `stake`, `withdraw`, `getReward`, and `notifyRewardAmount`. The modifier loops over every reward token, checkpointing reward emissions.

The problem with this logic is that `rewardTokens` is an array that can grow in size to an undefined capacity without the ability to reduce the size of the array. As a result, the gas costs required to execute this logic can also grow to an undefined capacity. If the gas costs ever exceed the block gas limit, the critical functions executing this logic will be permanently blocked. Since this modifier is used on every function responsible for removing staking and reward tokens from the contract, this would result in all staking and reward tokens being permanently locked in the contract.

Recommendation: Add and enforce a maximum length for the `rewardTokens` array, e.g. in `MultiRewards.addReward`:

```
// Revert if we've already reached the max length
if (rewardTokens.length == MAX_REWARD_LENGTH) revert EXCEEDS_MAX_REWARD_LENGTH();
```

3.2 Low Risk

3.2.1 Consider using a two-step ownership transfer pattern

Severity: Low Risk

Context: MarginalToken.sol#L56-L60

Description: In `MarginalToken`, we have a `setOwner` method, which immediately applies the given `_owner` address as the new `owner`.

```
function setOwner(address _owner) external {
    require(msg.sender == owner);
    emit OwnerChanged(owner, _owner);
    owner = _owner;
}
```

If the wrong `_owner` address is provided, ownership will be burned, causing the `mint` function to be permanently inaccessible.

Similarly, we inherit the `Ownable` contract in both the `MultiRewards` and `MultiRewardsFactory` contracts, which uses a similar mechanism which may also result in ownership being accidentally burned.

Recommendation: Consider using a two-step ownership transfer pattern.

3.3 Gas Optimization

3.3.1 Redundant `require` statement

Severity: Gas Optimization

Context: StakingPoints.sol#L64

Description: In `StakingPoints.free`, we include the following `require` statement prior to decrementing the `stake.balance`:

```
require(amount <= uint256(stake.balance), "amount > stake balance");
stake.balance -= uint224(amount);
```

This `require` statement is redundant due to the fact that if `amount > uint256(stake.balance)`, we will revert regardless on the following line due to an underflow.

Recommendation: Remove the `require` statement unless the revert message is more desirable than the gas savings.

3.3.2 Redundant on-chain data

Severity: Gas Optimization

Context: StakingPoints.sol#L20-L27

Description: `StakingPoints.stakes` is a mapping which keeps track of user balances and the block timestamp of their last update:

```
struct Stake {
    // balance at last update to stake
    uint224 balance;
    // timestamp of last update to stake
    uint32 blockTimestamp;
}
/// @inheritdoc IStakingPoints
mapping(address => Stake) public stakes;
```

The `blockTimestamp` does not appear to be referenced on-chain at any point, rather it seems that referencing the `block.timestamp` is only necessary in event emissions in `lock` and `free`. Assuming this is the case, we can simply map user addresses directly to their balances to save gas.

Recommendation: Map user addresses directly to their staked balance:

```
- struct Stake {
-     // balance at last update to stake
-     uint224 balance;
-     // timestamp of last update to stake
-     uint32 blockTimestamp;
- }
- /// @inheritdoc IStakingPoints
- mapping(address => Stake) public stakes;
+ mapping(address => uint256) public stakes;
```

Note that this also requires modifying current usage of the `stakes` mapping to accompany this change.

3.3.3 Use `uint256` event params in place of smaller variations

Severity: Gas Optimization

Context: StakingPoints.sol#L29-L38

Description: The events `Lock` and `Free` in `StakingPoints` use `uint32` and `uint224` parameters:

```
event Lock(
    address indexed sender,
    uint32 blockTimestampAfter,
    uint224 balanceAfter
);
event Free(
    address indexed sender,
    uint32 blockTimestampAfter,
    uint224 balanceAfter
);
```

To emit these events, we must first cast downcast the parameters from `uint256` values. Additionally, the compiler only works with 256 bit values so it upcasts them regardless.

We can test with the following contract that we save gas by using `uint256` values for the event parameters:

```
pragma solidity 0.8.17;

contract UintEvents {
    event Lock(
        address indexed sender,
        uint32 blockTimestampAfter,
        uint224 balanceAfter
    );

    event LockCheap(
        address indexed sender,
        uint256 blockTimestampAfter,
        uint256 balanceAfter
    );

    function lock(uint256 amount) external {
        emit Lock(msg.sender, uint32(block.timestamp), uint224(amount));
    }

    function lockCheap(uint256 amount) external {
        emit LockCheap(msg.sender, block.timestamp, amount);
    }
}
```

The result of calling `lock` v.s. `lockCheap` is an additional 39 gas.

Recommendation: Use `uint256` event parameters:

```
  event Lock(
      address indexed sender,
-    uint32 blockTimestampAfter,
+   uint256 blockTimestampAfter,
-    uint224 balanceAfter
+   uint256 balanceAfter
  );
  event Free(
      address indexed sender,
-    uint32 blockTimestampAfter,
+   uint256 blockTimestampAfter,
-    uint224 balanceAfter
+    uint256 balanceAfter
  );
```

3.3.4 Mark state variables as `immutable` and `constant` where relevant

Severity: Gas Optimization

Context: MultiRewardsFactory.sol#L17-L18

Description: `stakingRewardsGenesis` is a state variable which is assigned in the constructor and cannot possibly be changed, thus it can be marked as `immutable`. Immutable variables are not stored in contract storage, thus do not require expensive `SLOAD`'s (2100 gas) for retrieval, instead they are stored at the end of the compiled contract bytecode where it is much cheaper to retrieve.

Similarly, `rewardsDuration` is initialized when defined and also can never be changed, thus it can be marked as `constant`. Constant variables are also not stored in contract storage and instead are placed directly in the bytecode where they are used.

Recommendation: Mark `stakingRewardsGenesis` and `rewardsDuration` as `immutable` and `constant`, respectively:

```
- uint256 public stakingRewardsGenesis;
+ uint256 public immutable stakingRewardsGenesis;
- uint256 public rewardsDuration = 30 days;
+ uint256 public constant REWARDS_DURATION = 30 days;
```

Note also that constants should use `UPPER_CASE` naming convention as can be seen in the above snippet.

3.3.5 Cache storage variable used more than once

Severity: Gas Optimization

Context: MultiRewards.sol#L79-L89

Description: `MultiRewards.rewardPerToken` references the `_totalSupply` state variable twice, requiring two SLOAD's. Instead, it would be more efficient to cache the value locally and reuse the cached variable.

Recommendation: Cache `_totalSupply` and use the cached variable, e.g.:

```
+ uint256 totalSupply = _totalSupply;
- if (_totalSupply == 0) {
+ if (totalSupply == 0) {
      return rewardData[_rewardsToken].rewardPerTokenStored;
  }
  return
      rewardData[_rewardsToken].rewardPerTokenStored.add(
          lastTimeRewardApplicable(_rewardsToken)
              .sub(rewardData[_rewardsToken].lastUpdateTime)
              .mul(rewardData[_rewardsToken].rewardRate)
              .mul(1e18)
-           .div(_totalSupply)
+           .div(totalSupply)
      );
```

3.3.6 Circumstantially redundant `Reward` struct members

Severity: Gas Optimization

Context: MultiRewards.sol#L48-L57

Description: In `MultiRewards.addReward`, we accept `_rewardsDistributor` and `_rewardsDuration` parameters which are added to the `rewardData` for the included `_rewardsToken`:

```
function addReward(
    address _rewardsToken,
    address _rewardsDistributor,
    uint256 _rewardsDuration
) public onlyOwner {
    require(rewardData[_rewardsToken].rewardsDuration == 0);
    rewardTokens.push(_rewardsToken);
    rewardData[_rewardsToken].rewardsDistributor = _rewardsDistributor;
    rewardData[_rewardsToken].rewardsDuration = _rewardsDuration;
}
```

However, when `MultiRewards` is used with `MultiRewardsFactory`, the provided `_rewardsDistributor` and `_rewardsDuration` values are always the same:

```
IMultiRewards(multiRewards).addReward(
    rewardsToken,
    // @audit always uses address(this) for _rewardsDistributor
    address(this),
    // @audit this is a constant value
    rewardsDuration
);
```

Because of this, if the expectation is to only use `MultiRewards` in tandem with `MultiRewardsFactory`, we can remove these members from the `Reward` struct and reference known constant values. For example, instead of validating that `MultiRewards.notifyRewardAmount` is only called by the `rewardsDistributor`, we can add the `onlyOwner` modifier to the function. Similarly, instead of referencing the `rewardsDuration` per token, we can reference a global value which applies to all of them.

Recommendation: Consider removing `rewardsDistributor` and/or `rewardsDuration` members from the `Reward` struct, carefully replacing their logic as explained above. Additionally, if this change is made, provide documentation to clarify that the `MultiRewards` contract is only expected to work in tandem with the `MultiRewardsFactory` contract.

3.4 Informational

3.4.1 Failing test cases

Severity: Informational

Context: ./tests/

Description: Upon running the test suite with `ape test -s -m "not fuzzing"`, several tests are failing due to `AssertionErrors`. It appears that the problem is simply that the revert messages are not as expected and not anything more significant, however, it's important to have a passing test suite regardless for improved maintainability.

Output log:

```
====================================================================== short test summary info
↪  ======================================================================
FAILED tests/mrgl/test_mrgl_mint.py::test_mrgl_mint__reverts_when_not_minter_role - AssertionError: Expected
↪  revert message 'not minter' but got 'revert: not minter'.
FAILED tests/mrgl/test_mrgl_mint.py::test_mrgl_mint__reverts_when_not_allowed_yet - AssertionError: Expected
↪  revert message 'minting not allowed yet' but got 'revert: minting not allowed yet'.
FAILED tests/mrgl/test_mrgl_mint.py::test_mrgl_mint__reverts_when_to_zero_address - AssertionError: Expected
↪  revert message 'minting to zero address' but got 'revert: minting to zero address'.
FAILED tests/mrgl/test_mrgl_mint.py::test_mrgl_mint__reverts_when_exceed_mint_cap - AssertionError: Expected
↪  revert message 'exceeded mint cap' but got 'revert: exceeded mint cap'.
FAILED tests/multirewards/factory/test_multirewards_factory_add_reward.py::test_multirewards_factory_add_rewar ⌋
↪  d__reverts_when_not_owner - AssertionError: Expected revert message 'Ownable: caller is not the owner' but
↪  got 'revert: Ownable: caller is not the owner'.
FAILED tests/multirewards/factory/test_multirewards_factory_add_reward.py::test_multirewards_factory_add_rewar ⌋
↪  d__reverts_when_not_deployed - AssertionError: Expected revert message 'MultiRewardsFactory::addReward:
↪  not deployed' but got 'revert: MultiRewardsFactory::addReward: not deployed'.
FAILED tests/multirewards/factory/test_multirewards_factory_add_reward.py::test_multirewards_factory_add_rewar ⌋
↪  d__reverts_when_already_added - AssertionError: Expected revert message 'MultiRewardsFactory::addReward:
↪  already added' but got 'revert: MultiRewardsFactory::addReward: already added'.
FAILED tests/multirewards/factory/test_multirewards_factory_constructor.py::test_multirewards_factory_construc ⌋
↪  tor__reverts_when_genesis_too_soon - AssertionError: Expected revert message
↪  'MultiRewardsFactory::constructor: genesis too soon' but got 'revert: MultiRewardsFactory::constructor:
↪  genesis too soon'.
FAILED tests/multirewards/factory/test_multirewards_factory_deploy.py::test_multirewards_factory_deploy__rever ⌋
↪  ts_when_not_owner - AssertionError: Expected revert message 'Ownable: caller is not the owner' but got
↪  'revert: Ownable: caller is not the owner'.
FAILED tests/multirewards/factory/test_multirewards_factory_deploy.py::test_multirewards_factory_deploy__rever ⌋
↪  ts_when_already_deployed - AssertionError: Expected revert message 'MultiRewardsFactory::deploy: already
↪  deployed' but got 'revert: MultiRewardsFactory::deploy: already deployed'.
FAILED tests/multirewards/factory/test_multirewards_factory_notify_reward_amount.py::test_multirewards_factory ⌋
↪  _notify_reward_amount__reverts_when_not_ready - AssertionError: Expected revert message
↪  'MultiRewardsFactory::notifyRewardAmount: not ready' but got 'revert:
↪  MultiRewardsFactory::notifyRewardAmount: not ready'.
FAILED tests/multirewards/factory/test_multirewards_factory_notify_reward_amount.py::test_multirewards_factory ⌋
↪  _notify_reward_amount__reverts_when_not_deployed - AssertionError: Expected revert message
↪  'MultiRewardsFactory::notifyRewardAmount: not deployed' but got 'revert:
↪  MultiRewardsFactory::notifyRewardAmount: not de...
FAILED tests/multirewards/rewards/test_multirewards_add_reward.py::test_multirewards_add_reward__reverts_when_ ⌋
↪  not_owner - AssertionError: Expected revert message 'Ownable: caller is not the owner' but got 'revert:
↪  Ownable: caller is not the owner'.
FAILED tests/points/test_points_free.py::test_points_free__reverts_when_greater_than_stake_balance -
↪  AssertionError: Expected revert message 'amount > stake balance' but got 'revert: amount > stake balance'.
FAILED tests/points/test_points_lock.py::test_points_lock__reverts_when_stake_balance_greater_than_uint224_max
↪  - AssertionError: Expected revert message 'SafeCast: value doesn't fit in 224 bits' but got 'revert:
↪  SafeCast: value doesn't fit in 224 bits'.
================================================================= 15 failed, 33 passed in 12.42s
↪  =================================================================
```

Recommendation: Fix the failing tests.

3.4.2 Zero amounts accepted in `lock` and `free`

Severity: Informational

Context: StakingPoints.sol#L50, StakingPoints.sol#L62

Description: Both `StakingPoints.lock` and `StakingPoints.free` allow for a zero `amount` parameter to be provided. This results in execution successfully completing without any on-chain state changing.

Considering how points are tracked off-chain by tracking events, this may cause the indexing logic to behave unexpectedly.

Recommendation: Revert in both `lock` and `free` if the `amount == 0`.

3.4.3 Use `UPPER_CASE` naming convention for constants

Severity: Informational

Context: MarginalToken.sol#L10-L17

Description: In `MarginalToken`, we assign the following constants:

```
/// @notice Initial number of tokens in circulation
uint256 public constant initialSupply = 1_000_000_000e18; // 1 billion

/// @notice Minimum time between mints
uint256 public constant minimumTimeBetweenMints = 1 days * 365;

/// @notice Cap on the percentage of totalSupply that can be minted at each mint
uint256 public constant mintCap = 2;
```

The standard casing convention for constants in Solidity is to use UPPER_CASE. This improves readability by allowing developers to easily identify that a given variable is a constant.

Recommendation: Rename the constants with UPPER_CASE naming convention:

```
  /// @notice Initial number of tokens in circulation
- uint256 public constant initialSupply = 1_000_000_000e18; // 1 billion
+ uint256 public constant INITIAL_SUPPLY = 1_000_000_000e18; // 1 billion

  /// @notice Minimum time between mints
- uint256 public constant minimumTimeBetweenMints = 1 days * 365;
+ uint256 public constant MINIMUM_TIME_BETWEEN_MINTS = 1 days * 365;

  /// @notice Cap on the percentage of totalSupply that can be minted at each mint
- uint256 public constant mintCap = 2;
+ uint256 public constant MINT_CAP = 2;
```

3.4.4 Missing NatSpec comments

Severity: Informational

Context: MultiRewards.sol, MultiRewardsFactory.sol

Description: The `MultiRewards` and `MultiRewardsFactory` contract both lack NatSpec comments. Adding NatSpec comments can improve readability for developers and improve user experience on external applications such as Etherscan.

Recommendation: Implement NatSpec comments in the `MultiRewards` and `MultiRewardsFactory` contracts.

3.4.5 Outdated compiler version

Severity: Informational

Context: MultiRewards.sol#L2, MultiRewardsFactory.sol#L2

Description: Both `MultiRewards` and `MultiRewardsFactory` use Solidity v0.5.17. In general it's recommended to avoid using old compiler versions as bug fixes are added over time.

In evaluating known Solidity bugs, there did not appear to be any bugs which are concerning in the context of these contracts.

Recommendation: Consider using a newer compiler version.

Note: This consideration should also take into account the potential risk of modifying forked code to support the version change.

3.4.6 Include `rewardsToken` as `RewardAdded` event parameter

Severity: Informational

Context: MultiRewards.sol#L195

Description: In `MultiRewards.notifyRewardAmount`, we emit the `RewardAdded` event at the end of execution to log the amount of reward tokens added:

```
emit RewardAdded(reward);
```

This doesn't take into account the fact that there may be multiple different reward tokens and the actual one in use is not logged. This can make it much harder to index the total amount of rewards added for individual tokens which may be desirable.

Recommendation: Include a `rewardsToken` param on the `RewardAdded` event and pass the `_rewardsToken` when emitted in `notifyRewardAmount`, e.g.:

```
- event RewardAdded(uint256 reward);
+ event RewardAdded(uint256 reward, address indexed rewardsToken);
```

```
- emit RewardAdded(reward);
+ emit RewardAdded(reward, _rewardsToken);
```