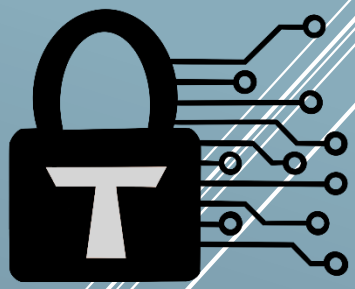
Trust Security



Smart Contract Audit

Mozaic.Fi xMozStaking

12/02/2024

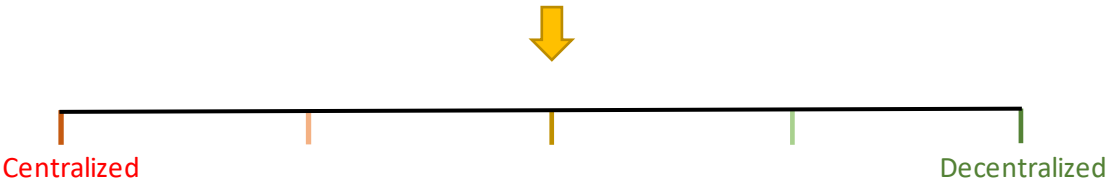
Executive summary

Category	Staking
Audited file count	1
Lines of Code	277
Auditor	MiloTruck
Time period	11/12/24 - 20/12/24

Findings

Severity	Total	Fixed	Acknowledged	Open
High	1	1	0	0
Medium	7	5	2	0
Low	2	2	0	0

Centralization score



Signature

Digitally signed by Trust
Date: 2024.02.12 13:53:26 +02'00'

Document properties

Versioning

Version	Date	Description
0.1	20/12/23	Client report
0.2	18/01/24	Mitigation review
0.3	12/02/24	Mitigation review #2

Contact

Trust

trust@trust-security.xyz

Introduction

Trust Security has conducted an audit at the customer's request. The audit is focused on uncovering security issues and additional bugs contained in the code defined in scope. Some additional recommendations have also been given when appropriate.

Scope

- contracts/xMozStaking.sol

Repository details

- **Repository URL:** https://github.com/Mozaic-fi/moz-staking
- **Commit hash:** 7c103f5921f5656c29861c3af68d0256cee75e5b
- **Mitigation review commit hash:** 1a6e718c9db748d30eb53988fafe29a6a60a2d3e
- **Mitigation review #2 commit hash:** f0acb279fbcf50495a9cae9029e6cb2224178db2

About Trust Security

Trust Security has been established by top-end blockchain security researcher Trust, in order to provide high quality auditing services. Since its inception it has safeguarded over 30 clients through private services and over 30 additional projects through bug bounty submissions.

About the Auditors

MiloTruck is a blockchain security researcher who specializes in smart contract security. Since March 2022, he has competed in over 25 auditing contests on Code4rena and won several of them against the best auditors in the field. He has also found multiple critical bugs in live protocols on Immunefi and is an active judge on Code4rena.

Disclaimer

Smart contracts are an experimental technology with many known and unknown risks. Trust Security assumes no responsibility for any misbehavior, bugs or exploits affecting the audited code or any part of the deployment phase.

Furthermore, it is known to all parties that changes to the audited code, including fixes of issues highlighted in this report, may introduce new issues and require further auditing.

Methodology

In general, the primary methodology used is manual auditing. The entire in-scope code has been deeply looked at and considered from different adversarial perspectives. Any additional dependencies on external code have also been reviewed.

Qualitative analysis

Metric	Rating	Comments
Code complexity	Moderate	Project is not complex, but some code could have been simplified.
Documentation	Mediocre	Project currently has limited documentation.
Best practices	Good	Project consistently adheres to industry standards.
Centralization risks	Good	Project has limited centralization risks.

Findings

High severity findings

TRST-H-1: Decreasing user balances in stakingInfo breaks the reward debt mechanism

- **Category:** Logical flaws
- **Source:** xMozStaking.sol
- **Status:** Fixed

Description

Whenever a user calls a state-changing function, *synchronizeXMozBalance()* is called to reduce the user's balance in **stakingInfo** to match their actual xMoz balance.

xMozStaking.sol#L221-L226

```
// Check if the staked xMoz balance is greater than the actual xMoz balance
if (stakingInfo[user] > userXMozBalance) {
    // Adjust the total staked amount and update the staking info
    totalStakedAmount -= (stakingInfo[user] - userXMozBalance);
    stakingInfo[user] = userXMozBalance;
}
```

However, decreasing the staker's balance using *synchronizeXMozBalance()* breaks the reward debt mechanism in *accumulateReward()*.

rewardDebts is calculated for users based on their **stakingInfo** balance:

xMozStaking.sol#L119

```
rewardDebts[msg.sender][rewardTokens[i]] = stakingInfo[msg.sender]
    .mul(accUnitPerShare[rewardTokens[i]]).div(1e30);
```

It is then subtracted from the user's accrued rewards in *accumulateReward()*:

xMozStaking.sol#L161-L165

```
uint256 userStake = stakingInfo[msg.sender];
uint256 accPerShare = accUnitPerShare[token];
uint256 userRewardDebt = rewardDebts[msg.sender][token];

uint256 rewardAmount = userStake.mul(accPerShare).div(1e30).sub(userRewardDebt);
```

Since **userRewardDebt** is based on the user's balance before it was decreased while **userStake** is the decreased balance, it becomes possible for **userRewardDebt** to be greater than **userStake * accPerShare**. For example:

- Assume **accUnitPerShare** = 1e30.
- Alice has 100e18 xMoz and she calls *stake()* for her entire balance:
    - **stakingInfo** = 100e18

    - Therefore, **rewardDebts** = 100e18 * 1e30 / 1e30 = 100e18
- 50e18 of Alice's xMoz balance is burnt.
- Now, if Alice calls *unstake()*:
    - *synchronizeXMozBalance()* reduces **stakingInfo** to 50e18.
    - In *accumulateReward()*, **userStake** is 50e18 while **userRewardDebt** is 100e18.
    - Thus, the **rewardAmount** calculation reverts with an arithmetic underflow.

Since *accumulateReward*() is called in *stake(), unstake()* and *claimReward()*, all three functions will always revert for the user. These functions will only be callable when **userStake * accPerShare** increases above **userRewardDebt**, which could take extremely long depending on:

1. How much the user's balance was decreased by *synchronizeXMozBalance()*
2. The speed at which rewards accrue for the user.

Recommended mitigation

Instead of calculating and storing the user's reward amount as debt whenever *stake(), unstake()* or *claimReward()* is called, consider storing **accPerUnitShare** at that point in time.

In *accumulateReward()*, the amount of rewards can be calculated with the difference between the stored **accPerUnitShare** and the current **accPerUnitShare**.

For example:

- Assume **accUnitPerShare** = 1e30
- User calls *stake()* for 100e18 xMoz:
    - **stakingInfo** = 100e18
    - Store the current **accUnitPerShare**, which is 1e30.
- Some time passes, **accUnitPerShare** increases to 1.5e30.
- User calls *claimReward()*:
    - Their reward amount is calculated as 100e18 * (1.5e30 – 1e30) / 1e30, which is 50e18.

This approach does not use the staker's previous balance, as such, it will still work even if *synchronizeXMozBalance()* decreases **stakingInfo**.

Team response

Fixed as recommended.

Mitigation review

Verified, the contract now tracks by the user's **accUnitPerShare** instead of reward debt, which removes the risk of an arithmetic underflow occurring.

Medium severity findings

TRST-M-1: Reward accounting breaks for tokens re-added using *setRewardConfig()*

- **Category:** Logical flaws
- **Source:** xMozStaking.sol
- **Status:** Fixed

Description

In *setRewardConfig()*, if a reward token was previously in **rewardTokens** but is not in **_rewardTokens**, it will be removed and staked users will no longer accrue it as rewards. However, when reward tokens are removed, **accUnitPerShare**, which stores the accrued amount of tokens per share, is not deleted.

This becomes a problem if a previously removed reward token is re-added to **rewardTokens,** since **accUnitPerShare** contains an old value but **rewardDebts** will be 0 for newly staked users.

xMozStaking.sol#L118-L120

```
for(uint i = 0; i < rewardTokens.length; i++) {
    rewardDebts[msg.sender][rewardTokens[i]] = stakingInfo[msg.sender]
        .mul(accUnitPerShare[rewardTokens[i]]).div(1e30);
}
```

For example:

- Assume xMoz is a reward token and its **accUnitPerShare** is currently 1e30.
- Owner calls *setRewardConfig()* and removes xMoz from **rewardTokens**.
- Alice calls *stake()* to stake 100e18 xMoz:
  - Since xMoz is not in **rewardTokens**, **rewardDebts** is not set for xMoz.
- Owner calls *setRewardConfig()* to add xMoz back to **rewardTokens** again.
- Alice calls *claimReward()*. In *accumulateReward()*:
  - **userStake** = 100e18, **accPerShare** = 1e48, **userRewardDebt** = 0
  - Therefore, she gets 100e18 xMoz as rewards.

Due to xMoz's old **accUnitPerShare**, Alice has gained 100 xMoz tokens although no time has passed since she staked.

Recommended mitigation

Ensure that tokens are not removed and then added back to **rewardTokens**. This can be achieved by refactoring the code to make it possible for tokens to be added to **rewardTokens**, but not removed.

Team response

Fixed by removing the ability to remove reward tokens. *setRewardConfig()* can only be called once to initialize **rewardTokens**, afterwards, only *addRewardToken()* can be used to add new reward tokens.

Mitigation review

addRewardToken() currently does not call *update()* in its logic. This makes it possible for users to accrue rewards from the newly added reward token instantly after *addRewardToken()* is called, for example:

- Assume **lastUpdateTime** was two weeks ago.
- *addRewardToken()* is called to add USDC as a new token. Since *update()* is not called, **lastUpdateTime** remains at two weeks ago.
- A user calls *update()* – this accrues two weeks' worth USDC as rewards since **lastUpdateTime** is two weeks ago, even though USDC was just added.

Consider calling *update()* before pushing the new reward token into **rewardTokens**:

xMozStaking.sol#L90-L92

```
  require(isExist == false, "XMozStaking: reward token already exist");
+ update();
  rewardTokens.push(_rewardToken);
  rewardAmountsPerWeek[_rewardToken] = _rewardAmountPerWeek;
```

Team response

Fixed as recommended.

Mitigation review #2

Verified, *update()* is now called in *addRewardToken()* before adding new reward tokens.

TRST-M-2: Removing reward tokens causes stakers to lose part of their rewards

- **Category:** Logical flaws
- **Source:** xMozStaking.sol
- **Status:** Fixed

Description

In *setRewardConfig()*, the owner has the ability to add and remove reward tokens from **rewardTokens**.

accumulateReward() loops through **rewardTokens** to calculate rewards for users:

xMozStaking.sol#L158-L168

```
function accumulateReward() internal {
    for (uint256 i = 0; i < rewardTokens.length; i++) {
        // ...
        uint256 rewardAmount = userStake.mul(accPerShare)
            .div(1e30).sub(userRewardDebt);
        accumulatedRewardAmounts[msg.sender][token] += rewardAmount;
    }
}
```

Additionally, *distributeReward()* loops through **rewardTokens** when sending rewards to users:

xMozStaking.sol#L170-L181

```
function distributeReward() internal {
    for (uint256 i = 0; i < rewardTokens.length; i++) {
        // ...
        if (userRewardAmount > 0) {
            safeRewardTransfer(token, msg.sender, userRewardAmount);
            accumulatedRewardAmounts[msg.sender][token] = 0;
        }
    }
}
```

Once a reward token is removed from **rewardTokens**, users will no longer be able to accrue or claim the token as rewards. This causes a loss of yield for stakers as they might have leftover amounts from reward periods before the token was removed.

Recommended mitigation

Consider refactoring the code to ensure the owner cannot remove tokens from **rewardTokens**.

Team response

Fixed by removing the ability to remove reward tokens. *setRewardConfig()* can only be called once to initialize **rewardTokens**, afterwards, only *addRewardToken()* can be used to add new reward tokens.

Mitigation review

Verified, the scenario described above is no longer possible as the owner cannot remove reward tokens.

TRST-M-3: Rewards distributed per week will be smaller than rewardAmountsPerWeek

- **Category:** Logical flaws
- **Source:** xMozStaking.sol
- **Status:** Acknowledged

Description

In *update()*, the amount of rewards to distribute is divided by **totalStakedAmount**:

xMozStaking.sol#L208-L213

```
uint256 supply = totalStakedAmount;
for (uint256 i = 0; i < rewardTokens.length; i++) {
    uint256 rewardInPeriod = durationInPeriods
        .mul(rewardAmountsPerWeek[rewardTokens[i]]);
    uint256 rewardPerShare = rewardInPeriod.mul(1e30).div(supply);
    accUnitPerShare[rewardTokens[i]] += rewardPerShare;
}
```

Since xMoz is not transferred to the contract when staking, a staker's xMoz balance can decrease while staked. Afterwards, when the user calls a state-changing function, *synchronizeXMozBalance()* reduces the user's staked amount and **totalStakedAmount** accordingly.

However, since *synchronizeXMozBalance()* is only called when the user interacts with the contract, **totalStakedAmount** will always be inflated above the actual total amount of everyone's stake.

For example:

- Alice calls *stake()* for 1000 xMoz tokens.
- 500 of her tokens are burnt.
- Before she calls any function in the contract, **totalStakedAmount** is 1000 but the actual total amount staked is 500.

As such, dividing by **totalStakedAmount** as shown above will cause **accUnitPerShare** to be smaller than what it should be, resulting in less rewards for stakers.

Recommended mitigation

Consider documenting that **rewardAmountsPerWeek** is not the exact amount of rewards distributed to stakers weekly.

Team response

Acknowledged.

TRST-M-4: Accruing rewards weekly allows users to gain rewards without fully staking

- **Category:** Logical flaws
- **Source:** xMozStaking.sol
- **Status:** Acknowledged

Description

In *update()*, rewards only accrue when one or more weeks has passed since **lastUpdateTime**:

xMozStaking.sol#L199-L216

```
function update() internal {
    uint256 _delta = block.timestamp - lastUpdateTime;
    uint256 durationInPeriods = _delta / 1 weeks;
    if(durationInPeriods > 0) {
        // Code to update lastUpdateTime and accPerUnitShare here
    }
}
```

This makes it possible for users to stake without updating **lastUpdateTime** or **accPerUnitShare**. An attacker can abuse this to earn rewards without staking for a prolonged duration by doing the following:

- Wait until **_delta** is slightly below 1 week.
- Call *stake()*, and since **durationInPeriods** is 0, the update is skipped.
- Wait a short while until **_delta** increases above 1 week.
- Call *unstake()*, which updates **accPerUnitShare** and accumulates one week's worth of rewards.

By repeating this every week, an attacker can essentially earn the same amount of rewards as other stakers without actually staking for any duration.

Recommended mitigation

Consider distributing rewards per block, instead of weekly.

Team response

Fixed by allowing users to call *unstake()* only after they have staked for a week with this check.

Mitigation review

This fix is not comprehensive. Users will still be able to gain one additional week's worth of rewards by staking right before **durationInPeriods** increases from zero to one. For example:

- Wait until **timeSinceLastUpdate** is slightly below 1 week.
- Call *stake()*, and since **durationInPeriods** is 0, the update is skipped.
- Wait a short while until **_ timeSinceLastUpdate** increases above 1 week.
- Call *claimRewards()*, which accumulates one week's worth of rewards to the user.
- Wait another 1 week to unstake, which gives the user another week's worth of rewards.
- As such, the user receives 2 weeks' worth of rewards although he has only staked for 1 week.

Consider implementing the fix recommended above instead.

Team response

Acknowledged. We are okay with this since users are still incentivized to convert Moz into xMoz tokens.

TRST-M-5: Reentrancy risk in *distributeReward()* can lead to double claiming of rewards

- **Category:** Reentrancy attacks
- **Source:** xMozStaking.sol
- **Status:** Fixed

Description

distributeReward() transfers tokens to users before resetting **accumulatedRewardAmounts** in a loop:

xMozStaking.sol#L170-L181

```
function distributeReward() internal {
    for (uint256 i = 0; i < rewardTokens.length; i++) {
        // ...
        if (userRewardAmount > 0) {
            safeRewardTransfer(token, msg.sender, userRewardAmount);
            accumulatedRewardAmounts[msg.sender][token] = 0;
        }
    }
}
```

This is a violation of the Checks-Effects-Interaction pattern. If **token** happens to be one that gives the receiver execution flow, such as an ERC-777 token, it becomes possible for an attacker to drain all of the contract's reward tokens by re-entering the *claimReward()* function.

Recommended mitigation

Perform the token transfer after resetting **accumulatedRewardAmounts**:

```
  if (userRewardAmount > 0) {
-     safeRewardTransfer(token, msg.sender, userRewardAmount);
      accumulatedRewardAmounts[msg.sender][token] = 0;
+     safeRewardTransfer(token, msg.sender, userRewardAmount);
  }
```

Team response

Fixed as recommended.

Mitigation review

Verified, **accumulatedRewardAmounts** is now updated before transferring tokens.

TRST-M-6: Reentrancy risk in *claimReward()* and *unstake()*

- **Category:** Reentrancy attacks
- **Source:** xMozStaking.sol
- **Status:** Fixed

Description

claimReward() transfers tokens to users using *distributeReward()* before updating **lastAccUnitPerShare** in a loop:

xMozStaking.sol#L171-L177

```
distributeReward();

// Update user's reward debts
for (uint256 i = 0; i < rewardTokens.length; i++) {
    address token = rewardTokens[i];
    lastAccUnitPerShare[msg.sender][token] = accUnitPerShare[token];
}
```

This is a violation of the Checks-Effects-Interaction pattern.

If one of the reward tokens happens to be one that gives the receiver execution flow, such as an ERC-777 token, it becomes possible for an attacker to drain all of the contract's reward tokens by re-entering the *claimReward()* function.

This applies to *unstake()* as well since it follows the same pattern.

Recommended mitigation

In *claimReward()* and *unstake()*, consider calling *distributeReward()* after updating **lastAccUnitPerShare**:

```
- distributeReward();

  // Update user's reward debts
  for (uint256 i = 0; i < rewardTokens.length; i++) {
      address token = rewardTokens[i];
      lastAccUnitPerShare[msg.sender][token] = accUnitPerShare[token];
  }
+ distributeReward();
```

Team response

Fixed by refactoring the logic that updates **lastAccUnitPerShare** into the *updateLastAccUnitPerShare()* function, and calling it before *distributeReward()* in both functions.

Mitigation review

Verified, *distributeReward()* is now called after **lastAccUnitPerShare** so exploiting reentrancy is no longer possible.

TRST-M-7: Unsafe token transfers in *safeRewardTransfer()*

- **Category:** Logical flaws
- **Source:** xMozStaking.sol
- **Status:** Fixed

Description

safeRewardTransfer() uses *transfer()* wrapped in a try-catch to transfer reward tokens:

xMozStaking.sol#L224-L234

```
// Use try-catch to handle transfer failures
if(rewardAmount > 0) {
    try IERC20(_rewardToken).transfer(_to, rewardAmount) {
    } catch {}
}

// Check if fee is greater than 0 before transferring to treasury
if (fee > 0) {
    try IERC20(_rewardToken).transfer(treasury, fee) {
    } catch {}
}
```

However, this will revert for ERC-20 tokens that do not return a bool when *transfer()* is called, such as USDT, since the IERC20 interface expects a bool to be returned.

Additionally, the *transfer()* could silently fail when it is not supposed to – for example, if a user calls *distributeReward()* with too little gas and *transfer()* reverts with an out-of-gas error, *distributeReward()* would not revert. This results in a loss of rewards for users.

Recommended mitigation

Consider using the following function to perform token transfers in *safeRewardTransfer()*:

```
function _safeTransfer(address token, address to, uint256 value) internal {
    require(token.code.length != 0, "token address has no code");

    (bool success, bytes memory data) = token.call(
        abi.encodeCall(IERC20.transfer, (to, value))
    );

    for (uint256 i; i < skippedTokens.length; i++) {
        if (token == skippedTokens[i]) {
            return;
        }
    }

    require(success, "transfer reverted");
    require(data.length == 0 || abi.decode(data, (bool)), "transfer returned false");
}
```

Note that **skippedTokens** is an address[] state variable for the admin to specify which tokens that are allowed to fail. For example, if USDC blacklisted the xMozStaking contract, its address should be added to **skippedTokens**.

Team response

Fixed as recommended.

Mitigation review

Verified, reward token transfers are now performed using the *_safeTransfer()* function.

Low severity findings

TRST-L-1: *distributeReward()* breaks if one of the token transfers reverts

- **Category:** Logical flaws
- **Source:** xMozStaking.sol
- **Status:** Fixed

Description

The only way for users to claim their rewards is through *claimReward()*, which sends reward tokens to users using *distributeReward()* in a loop:

xMozStaking.sol#L170-L181

```
function distributeReward() internal {
    for (uint256 i = 0; i < rewardTokens.length; i++) {
        // ...
        if (userRewardAmount > 0) {
            safeRewardTransfer(token, msg.sender, userRewardAmount);
            accumulatedRewardAmounts[msg.sender][token] = 0;
        }
    }
}
```

If one of the token transfers happen to revert (e.g. reward token is USDC, the user becomes blacklisted), the entire call to *claimReward()* will revert. This makes it impossible for the user to claim rewards that are in other tokens.

Recommended mitigation

Consider adding a function to allow users to claim rewards by tokens individually:

```
function claimRewardForToken(address token) external {
    uint256 rewardAmount = accumulatedRewardAmounts[msg.sender][token];
    safeRewardTransfer(token, msg.sender, rewardAmount);
    accumulatedRewardAmounts[msg.sender][token] = 0;
}
```

Team response

Fixed by adding the recommended function.

Mitigation review

Verified, *claimRewardForToken()* now allows users to claim their rewards by tokens instead of all at once.

TRST-L-2: *getClaimableAmounts()* doesn't synchronize the user's balance

- **Category:** Logical flaws
- **Source:** xMozStaking.sol

- **Status:** Fixed

Description

getClaimableAmounts() is meant to calculate the amount of reward tokens a user will receive if he calls *claimReward()*.

However, the function does not synchronize the user's balance when calculating reward amounts, which *claimReward()* does. As such, *getClaimableAmounts*() might return reward amounts higher than the actual claimable amount.

Recommended mitigation

Use the user's actual xMoz balance instead of his staked amount if **stakingInfo** is larger:

xMozStaking.sol#L276-L278

```
+ uint256 xMozBalance = IERC20(xMoz).balanceOf(user);
+ uint256 stakedBalance = stakingInfo[user];
+ uint256 userStake = stakedBalance > xMozBalance ? xMozBalance : stakedBalance;
  for (uint256 i = 0; i < rewardTokens.length; i++) {
      address token = rewardTokens[i];
-     uint256 userStake = stakingInfo[user];
```

Team response

Fixed as recommended.

Mitigation review

The recommended fix was not comprehensive, it did not decrease **supply** although the user's staked amount is decreased. This is what happens when *synchronizeXMozBalance()* is called, as such, the calculated claimable amount will be different.

Consider applying the following fix instead:

xMozStaking.sol#L315-L316

```
  uint256 supply = totalStakedAmount;
+ uint256 stakedBalance = stakingInfo[user];
+ uint256 xMozBalance = IERC20(xMoz).balanceOf(user);
+ if (stakedBalance > xMozBalance) {
+     supply -= stakedBalance - xMozBalance;
+     stakedBalance = xMozBalance;
+ }
  for (uint256 i = 0; i < rewardTokens.length; i++) {
```

Team response

Fixed as recommended.

Mitigation review #2

Verified, **supply** is now decreased alongside the user's **stakedBalance**.

Additional recommendations

TRST-R-1: Use 10_000 for better readability

Consider using 10_000 for readability:

xMozStaking.sol#L14

```
- uint256 public constant BP_DENOMINATOR = 10000;
+ uint256 public constant BP_DENOMINATOR = 10_000;
```

TRST-R-2: xMoz can be declared immutable

Since **xMoz** is changed only in the constructor, it can be declared as immutable:

xMozStaking.sol#L32-L33

```
  // Address of the staked token
- address public xMoz;
+ address public immutable xMoz;
```

TRST-R-3: Duplicated rewardDebts logic should be in an internal function

The following code appears in three functions – *stake()*, *unstake()* and *claimReward()*.

```
// Update user's reward debts
for (uint256 i = 0; i < rewardTokens.length; i++) {
    address token = rewardTokens[i];
    uint256 userRewardDebt = stakingInfo[msg.sender]
        .mul(accUnitPerShare[token]).div(1e30);
    rewardDebts[msg.sender][token] = userRewardDebt;
}
```

Consider moving the code above into an internal function, and using that internal function in *stake()*, *unstake()* and *claimReward()*.

TRST-R-4: Gas savings in *accumulateReward()*

In *accumulateReward()*, the following line can be moved outside the loop:

xMozStaking.sol#L158-L162

```
  function accumulateReward() internal {
+     uint256 userStake = stakingInfo[msg.sender];
      for (uint256 i = 0; i < rewardTokens.length; i++) {
          address token = rewardTokens[i];
-         uint256 userStake = stakingInfo[msg.sender];
```

TRST-R-5: Check if treasury fee is non-zero in *safeRewardTransfer()*

In *safeRewardTransfer()*, consider checking if **treasuryFeeBP** is non-zero to prevent performing any unnecessary transfers:

xMozStaking.sol#L188-L193

```
- if (treasury != address(0)) {
+ if (treasury != address(0) && treasuryFeeBP != 0) {
      // Code to transfer reward and treasury fee
  } else {
```

Centralization risks

TRST-CR-1: Missing maximum number of reward tokens in *setRewardConfig()*

The owner of the xMozStaking contract sets **rewardTokens** using *setRewardConfig()*. All state-changing functions in the contract, such as *update()* or *stake()*, iterate through **rewardTokens** with a for-loop. However, if the owner ever adds too many addresses to **rewardTokens**, there is a possibility of these functions consuming too much gas and reverting with an out-of-gas error. This results in DOS for the contract as all user functions will not be callable.

Consider implementing a maximum length for **rewardTokens**, which determines the maximum number of reward tokens for the contract.

TRST-CR-2: xMozStaking risks

xMozStaking.sol should be considered partially centralized.

The owner address can:

- Set **rewardTokens** and **rewardAmountsPerWeek**, which allows the owner to prevent users from receiving rewards, even if they have already accrued.
- As mentioned in TRST-CR-1, the owner can also DOS all user functions in the contract.