Generated by AuditX

About

The provided code represents a Solana program for liquid staking, specifically Marinade Finance. It allows users to stake SOL tokens and receive mSOL tokens in return, which represent their staked SOL. The program manages validator selection, stake account management, and liquidity pools for swapping between SOL and mSOL. It also includes features for delayed unstaking and emergency pauses.

Findings Severity breakdown

- Critical: 0
- High: 1
- Medium: 3
- Low: 6
- Gas: 2

Integer Overflow in Proportional Function

- **Title:** Integer Overflow in Proportional Function
- **Severity:** High
- **Description:** The `proportional` function in `calc.rs` performs multiplication and division using `u128` to avoid overflows. However, if `amount * numerator` exceeds the maximum value of `u128` before the division, an overflow will occur, leading to an incorrect result. While `try_from` is used to ensure the result fits within a `u64`, the overflow prior to this check can still lead to incorrect calculations.
- **Impact:** Incorrect calculations of shares, value, fees, or other proportional values, potentially leading to loss of funds for users or the protocol. For example, calculating the mSOL amount to

mint for a given SOL deposit could result in a user receiving fewer mSOL than they should.
- **Location:** calc.rs:16
- **Recommendation:** Implement checks to ensure `(amount as u128) * (numerator as u128)` does not exceed `u128::MAX` before the division occurs. If it does, return an error or use a different calculation method to prevent overflow. Consider using the `checked_mul` method and returning an error if `None` is returned.

```
pub fn proportional(amount: u64, numerator: u64, denominator: u64) -> R
    if denominator == 0 {
        return Ok(amount);
    }

    let amount_u128 = amount as u128;
    let numerator_u128 = numerator as u128;
    let denominator_u128 = denominator as u128;

    // Check for potential overflow before multiplication
    let product = amount_u128.checked_mul(numerator_u128).ok_or(error!(

    u64::try_from(product / denominator_u128)
        .map_err(|_| error!(MarinadeError::CalculationFailure))
}
```

Missing Paused Check in Multiple Functions

- **Title:** Missing Paused Check in Multiple Functions
- **Severity:** Medium
- **Description:** The `pause` and `resume` instructions exist, but many other key functions within the `marinade_finance` module do not check the `paused` state before execution. This means that even when the protocol is paused, some operations could still be performed, potentially leading to unintended consequences or exploits.
- **Impact:** The contract may not behave as expected during a paused state, potentially allowing malicious actors to bypass the intended restrictions and cause harm to the system.

- **Location:** All functions in `lib.rs` within the `marinade_finance` module except `pause` and `resume`.
- **Recommendation:** Add a check at the beginning of each function (except `pause` and `resume`) to ensure `ctx.accounts.state.paused` is `false`. If it is `true`, return an error.

```
pub fn deposit(ctx: Context<Deposit>, lamports: u64) -> Result<()> {
    check_context(&ctx)?;
    if ctx.accounts.state.paused {
        return err!(MarinadeError::ProgramIsPaused);
    }
    ctx.accounts.process(lamports)
}
```

Unprotected Realloc Instructions

- **Title:** Unprotected Realloc Instructions
- **Severity:** Medium
- **Description:** The instructions `realloc_validator_list` and `realloc_stake_list` allow the resizing of the validator and stake lists, respectively. These functions are not guarded by sufficient access control, potentially allowing an unauthorized actor to arbitrarily resize these lists. Resizing lists can cause unexpected behavior in the program, potentially leading to a denial of service or data corruption.
- **Impact:** Unauthorized resizing of validator and stake lists, potentially causing denial of service or data corruption.
- **Location:** `lib.rs`: `realloc_validator_list`, `realloc_stake_list`
- **Recommendation:** Implement access controls to ensure only authorized entities (e.g., the admin) can call `realloc_validator_list` and `realloc_stake_list`.

```
pub fn realloc_validator_list(ctx: Context<ReallocValidatorList>, capac
    check_context(&ctx)?;
    // ADD CHECK HERE
    require_keys_eq!(ctx.accounts.state.admin_authority, ctx.accounts.a
```

```
    ctx.accounts.process(capacity)
}
```

Missing List Capacity Check on Realloc Instructions

- **Title:** Missing List Capacity Check on Realloc Instructions
- **Severity:** Medium
- **Description:** When reallocating the validator and stake lists, the code doesn't verify if the new capacity `capacity` is less than or equal to the current number of items in the list (`count`). If `capacity` is set to be smaller than the current number of elements in the list, the realloc operation will lead to data loss.
- **Impact:** Data loss can occur when reallocating lists to a size smaller than the current number of elements.
- **Location:** `ReallocValidatorList.process` , `ReallocStakeList.process` in instruction definitions.
- **Recommendation:** Before reallocating the list, add a check to ensure that `capacity` is not less than the current `count` . If it is, return an error. This check is already implemented, but it needs to be placed at the correct location within `process` .

Within instruction processing logic (hypothetical `ReallocValidatorList.process`):

```
impl<'info> ReallocValidatorList<'info> {
    pub fn process(&mut self, capacity: u32) -> Result<()> {
        let validator_system = &mut self.state.validator_system;
        let current_count = validator_system.validator_list.count;

        if capacity < current_count {
            return err!(MarinadeError::ShrinkingListWithDeletingContent
        }

        // ... existing reallocation logic ...
        Ok(())
    }
}
```

Insufficient Input Validation: Min Stake Account Delegation

- **Title:** Insufficient Input Validation: Min Stake Account Delegation
- **Severity:** Low
- **Description:** The program has a `min_stake` parameter in the `State` account and checks for minimum deposits (`min_deposit`). There is no validation to ensure that `min_stake` is greater than the cost of rent exemption for accounts, which is `rent_exempt_for_token_acc` in the `State` account.
- **Impact:** Stake accounts could be created with a balance so low that the account cannot operate. Also, there is a check in `WithdrawStakeAccount` that prevents withdrawing if the remainder is less than `min_stake`. Therefore, if `min_stake` is too low, someone could withdraw with a fee and create many stake accounts with balances less than the rent exemption amount. These accounts are useless since they cannot be used.
- **Location:** `State` and instructions related to stake account management.
- **Recommendation:** Add a check to the `initialize` instruction and any instruction that modifies `min_stake` to ensure that `min_stake` is always greater than `rent_exempt_for_token_acc`. Specifically check `min_stake > rent_exempt_for_token_acc`.

Inconsistent use of safe math operations

- **Title:** Inconsistent use of safe math operations
- **Severity:** Low
- **Description:** The code uses both standard arithmetic operators (`+`, `-`, `*`, `/`) and safe arithmetic methods like `checked_add`, `checked_sub`, `checked_mul`, and `checked_div` inconsistently. Using safe arithmetic operations consistently helps prevent integer overflow and underflow vulnerabilities. Inconsistencies make it harder to reason about the correctness and security of the code.

For example, the function `Fee::apply` does not use safe math while other functions do.
- **Impact:** Potential integer overflow and underflow vulnerabilities.
- **Location:** All files
- **Recommendation:** Review the code and ensure that safe arithmetic operations (`checked_add`, `checked_sub`, `checked_mul`, `checked_div`, etc.) are used consistently, especially in arithmetic operations involving user-controlled inputs or critical calculations.

Missing or Insufficient Documentation

- **Title:** Missing or Insufficient Documentation
- **Severity:** Low
- **Description:** The code lacks comprehensive documentation, especially for complex logic and critical functions. The absence of clear comments and explanations makes the code harder to understand, audit, and maintain.
- **Impact:** Increased risk of errors, vulnerabilities, and difficulties in understanding the contract's behavior. This also makes it difficult for external auditors to analyze the code and identify potential issues.
- **Location:** All files.
- **Recommendation:** Add detailed comments to explain the purpose, functionality, and security considerations of each function, especially those involving complex logic or critical operations. Document the expected inputs, outputs, and potential error conditions.

Unused Error Code

- **Title:** Unused Error Code
- **Severity:** Low
- **Description:** The `MarinadeError` enum contains an unused error code: `NotUsed6027`.

- **Impact:** This code adds to the binary size and could cause confusion for developers.
- **Location:** error.rs:55
- **Recommendation:** Remove the unused error code `NotUsed6027` from `MarinadeError`.

Potential Gas Optimization: Inlining Small Functions

- **Title:** Potential Gas Optimization: Inlining Small Functions
- **Severity:** Gas
- **Description:** The code uses several small functions, such as `value_from_shares` in `calc.rs`, that are essentially aliases for other functions. Inlining these functions can reduce gas costs by avoiding the overhead of function calls.
- **Impact:** Reduced gas costs for contract execution.
- **Location:** `calc.rs:20`
- **Recommendation:** Consider inlining the `value_from_shares` function directly into the places where it is called to avoid function call overhead. Since the function is annotated with `#[inline]`, the compiler will most likely inline the function already.

Redundant Ownership Checks in `check_token_source_account`

- **Title:** Redundant Ownership Checks in `check_token_source_account`
- **Severity:** Low
- **Description:** In `checks.rs`, the function `check_token_source_account` contains an `else if` that checks if `*authority == source_account.owner`. If the condition `source_account.delegate.contains(authority)` is false, it only makes sense to continue if `authority` matches the `source_account.owner`.
- **Impact:** Slight unnecessary gas cost due to redundant check.
- **Location:** checks.rs:120

- **Recommendation:** Convert the `else if` to an `else`.

```
pub fn check_token_source_account<'info>(
    source_account: &Account<'info, TokenAccount>,
    authority: &Pubkey,
    token_amount: u64,
) -> Result<()> {
    if source_account.delegate.contains(authority) {
        // if delegated, check delegated amount
        // delegated_amount & delegate must be set on the user's msol a
        require_lte!(
            token_amount,
            source_account.delegated_amount,
            MarinadeError::NotEnoughUserFunds
        );
    } else {
        require_lte!(
            token_amount,
            source_account.amount,
            MarinadeError::NotEnoughUserFunds
        );
    }
    Ok(())
}
```

Potential Gas Optimization: Remove `check_context` calls

- **Title:** Potential Gas Optimization: Remove `check_context` calls
- **Severity:** Gas
- **Description:** `check_context` is called at the beginning of every instruction, where it checks for the correct program ID and that there are no remaining accounts. These checks are not necessary.
- **Impact:** Reduced gas costs for contract execution.
- **Location:** All functions in `lib.rs` within the `marinade_finance` module.
- **Recommendation:** Remove the `check_context` call in all the instruction functions.

Detailed Analysis

- **Architecture:** The contract is well-structured, using modules for different functionalities such as admin, crank, delayed unstake, liq pool, management, user, and state. It employs a clear separation of concerns, with distinct modules for calculations, checks, events, instructions, and state management.
- **Code Quality:** The code generally follows best practices and conventions, with consistent naming and formatting. However, there is a lack of comprehensive documentation, which makes the code harder to understand and audit. The inconsistent use of safe math operations is also a concern.
- **Centralization Risks:** The contract relies on a central `admin_authority` for key operations such as changing authorities, configuring parameters, and pausing/resuming the contract. This central control point introduces centralization risks. If the admin key is compromised, an attacker could take control of the protocol.
- **Systemic Risks:** The contract depends on external protocols such as the SPL token program and the Solana clock. Failures or vulnerabilities in these external protocols could impact the contract's functionality. The contract's integration with validators also introduces systemic risks. The behavior of validators can impact the contract's performance and security.
- **Testing & Verification:** The provided code does not include any unit tests or integration tests. The lack of testing makes it difficult to assess the contract's correctness and security.

Final Recommendations

1. **Implement overflow checks:** Add overflow checks in the `proportional` function to prevent potential integer overflows.
2. **Add paused checks:** Implement checks in all relevant functions to ensure the contract is paused when it should be.
3. **Implement access controls:** Add access controls to sensitive functions to restrict access to authorized entities only.

4. **Add check on realloc capacity:** Add checks on realloc instructions to prevent list capacity to be less than the current count.
5. **Validate stake amount:** Validate `min_stake` against `rent_exempt_for_token_acc`.
6. **Use safe math consistently:** Ensure consistent use of safe math operations throughout the code.
7. **Add documentation:** Add comprehensive documentation to explain the contract's logic and security considerations.
8. **Remove unused code:** Remove the unused error code.
9. **Inline small functions:** consider inlining small functions to reduce gas costs.
10. **Remove redundant code:** Remove redundant ownership checks.
11. **Remove check context calls:** Remove unnecessary `check_context` calls.
12. **Add unit tests:** Implement comprehensive unit tests and integration tests to verify the contract's functionality and security.

Improved Code with Security Comments

```
 // File: calc.rs
//! Common calculations

use crate::error::MarinadeError;
use anchor_lang::prelude::{error, Result};
use std::convert::TryFrom;

/// calculate amount*numerator/denominator
/// as value  = shares * share_price where share_price=total_value/tota
/// or shares = amount_value / share_price where share_price=total_valu
///     => shares = amount_value * 1/share_price where 1/share_price=to
pub fn proportional(amount: u64, numerator: u64, denominator: u64) -> R
    if denominator == 0 {
        return Ok(amount);
    }

    let amount_u128 = amount as u128;
```

```
    let numerator_u128 = numerator as u128;
    let denominator_u128 = denominator as u128;

    // Check for potential overflow before multiplication
    let product = amount_u128.checked_mul(numerator_u128).ok_or(error!(

    u64::try_from(product / denominator_u128)
        .map_err(|_| error!(MarinadeError::CalculationFailure))
}

#[inline] //alias for proportional
pub fn value_from_shares(shares: u64, total_value: u64, total_shares: u
    proportional(shares, total_value, total_shares)
}

pub fn shares_from_value(value: u64, total_value: u64, total_shares: u6
    if total_shares == 0 {
        //no shares minted yet / First mint
        Ok(value)
    } else {
        proportional(value, total_shares, total_value)
    }
}
```

```
 // File: lib.rs
#![cfg_attr(not(debug_assertions), deny(warnings))]

use anchor_lang::prelude::*;

use error::MarinadeError;

pub mod calc;
pub mod checks;
pub mod error;
pub mod events;
pub mod instructions;
pub mod state;

use instructions::*;

#[cfg(not(feature = "no-entrypoint"))]
use solana_security_txt::security_txt;
pub use state::State;
```

```
declare_id!("MarBmsSgKXdrN1egZf5sqe1TMai9K1rChYNDJgjq7aD");

#[cfg(not(feature = "no-entrypoint"))]
security_txt! {
    name: "Marinade Liquid Staking",
    project_url: "https://marinade.finance",
    contacts: "link:https://docs.marinade.finance/marinade-dao,link:htt
    policy: "https://docs.marinade.finance/marinade-protocol/security",
    preferred_languages: "en",
    source_code: "https://github.com/marinade-finance/liquid-staking-pr
    source_release: "v2.0",
    auditors: "https://docs.marinade.finance/marinade-protocol/security
}

//----------------------------------------------------
#[program]
pub mod marinade_finance {

    use super::*;

    //--------------------------------------------------------------------
    // Base Instructions
    //--------------------------------------------------------------------
    // Includes: initialization, contract parameters
    // basic user functions: (liquid)stake, liquid-unstake
    // liq-pool: add-liquidity, remove-liquidity
    // Validator list management
    //--------------------------------------------------------------------

    pub fn initialize(ctx: Context<Initialize>, data: InitializeData) -
        //check_context(&ctx)?; // Removed check_context to optimize ga
        ctx.accounts
            .process(data, *ctx.bumps.get("reserve_pda").unwrap())?;
        Ok(())
    }

    pub fn change_authority(
        ctx: Context<ChangeAuthority>,
        data: ChangeAuthorityData,
    ) -> Result<()> {
        //check_context(&ctx)?; // Removed check_context to optimize ga
        if ctx.accounts.state.paused {
            return err!(MarinadeError::ProgramIsPaused);
        }
        ctx.accounts.process(data)
```

```
    }

    pub fn add_validator(ctx: Context<AddValidator>, score: u32) -> Res
        //check_context(&ctx)?; // Removed check_context to optimize ga
        if ctx.accounts.state.paused {
            return err!(MarinadeError::ProgramIsPaused);
        }
        ctx.accounts.process(score)
    }

    pub fn remove_validator(
        ctx: Context<RemoveValidator>,
        index: u32,
        validator_vote: Pubkey,
    ) -> Result<()> {
        //check_context(&ctx)?; // Removed check_context to optimize ga
        if ctx.accounts.state.paused {
            return err!(MarinadeError::ProgramIsPaused);
        }
        ctx.accounts.process(index, validator_vote)
    }

    pub fn set_validator_score(
        ctx: Context<SetValidatorScore>,
        index: u32,
        validator_vote: Pubkey,
        score: u32,
    ) -> Result<()> {
        //check_context(&ctx)?; // Removed check_context to optimize ga
        if ctx.accounts.state.paused {
            return err!(MarinadeError::ProgramIsPaused);
        }
        ctx.accounts.process(index, validator_vote, score)
    }

    pub fn config_validator_system(
        ctx: Context<ConfigValidatorSystem>,
        extra_runs: u32,
    ) -> Result<()> {
        //check_context(&ctx)?; // Removed check_context to optimize ga
        if ctx.accounts.state.paused {
            return err!(MarinadeError::ProgramIsPaused);
        }
        ctx.accounts.process(extra_runs)
    }
```

```
// deposit AKA stake, AKA deposit_sol
pub fn deposit(ctx: Context<Deposit>, lamports: u64) -> Result<()>
    //check_context(&ctx)?; // Removed check_context to optimize ga
    if ctx.accounts.state.paused {
        return err!(MarinadeError::ProgramIsPaused);
    }
    ctx.accounts.process(lamports)
}

// SPL stake pool like
pub fn deposit_stake_account(
    ctx: Context<DepositStakeAccount>,
    validator_index: u32,
) -> Result<()> {
    //check_context(&ctx)?; // Removed check_context to optimize ga
    if ctx.accounts.state.paused {
        return err!(MarinadeError::ProgramIsPaused);
    }
    ctx.accounts.process(validator_index)
}

pub fn liquid_unstake(ctx: Context<LiquidUnstake>, msol_amount: u64
    //check_context(&ctx)?; // Removed check_context to optimize ga
    if ctx.accounts.state.paused {
        return err!(MarinadeError::ProgramIsPaused);
    }
    ctx.accounts.process(msol_amount)
}

pub fn add_liquidity(ctx: Context<AddLiquidity>, lamports: u64) ->
    //check_context(&ctx)?; // Removed check_context to optimize ga
    if ctx.accounts.state.paused {
        return err!(MarinadeError::ProgramIsPaused);
    }
    ctx.accounts.process(lamports)
}

pub fn remove_liquidity(ctx: Context<RemoveLiquidity>, tokens: u64)
    //check_context(&ctx)?; // Removed check_context to optimize ga
    if ctx.accounts.state.paused {
        return err!(MarinadeError::ProgramIsPaused);
    }
    ctx.accounts.process(tokens)
}
```

```
    pub fn config_lp(ctx: Context<ConfigLp>, params: ConfigLpParams) ->
        //check_context(&ctx)?; // Removed check_context to optimize ga
        if ctx.accounts.state.paused {
            return err!(MarinadeError::ProgramIsPaused);
        }
        ctx.accounts.process(params)
    }

    pub fn config_marinade(
        ctx: Context<ConfigMarinade>,
        params: ConfigMarinadeParams,
    ) -> Result<()> {
        //check_context(&ctx)?; // Removed check_context to optimize ga
        if ctx.accounts.state.paused {
            return err!(MarinadeError::ProgramIsPaused);
        }
        ctx.accounts.process(params)
    }

    //-----------------------------------------------------------------
    // Advanced instructions: deposit-stake-account, Delayed-Unstake
    // backend/bot "crank" related functions:
    // * order_unstake (starts stake-account deactivation)
    // * withdraw (delete & withdraw from a deactivated stake-account)
    // * update (compute stake-account rewards & update mSOL price)
    //-----------------------------------------------------------------

    pub fn order_unstake(ctx: Context<OrderUnstake>, msol_amount: u64)
        //check_context(&ctx)?; // Removed check_context to optimize ga
        if ctx.accounts.state.paused {
            return err!(MarinadeError::ProgramIsPaused);
        }
        ctx.accounts.process(msol_amount)
    }

    pub fn claim(ctx: Context<Claim>) -> Result<()> {
        //check_context(&ctx)?; // Removed check_context to optimize ga
        if ctx.accounts.state.paused {
            return err!(MarinadeError::ProgramIsPaused);
        }
        ctx.accounts.process()
    }

    pub fn stake_reserve(ctx: Context<StakeReserve>, validator_index: u
```

```
        //check_context(&ctx)?; // Removed check_context to optimize ga
        if ctx.accounts.state.paused {
            return err!(MarinadeError::ProgramIsPaused);
        }
        ctx.accounts.process(validator_index)
    }

    pub fn update_active(
        ctx: Context<UpdateActive>,
        stake_index: u32,
        validator_index: u32,
    ) -> Result<()> {
        //check_context(&ctx)?; // Removed check_context to optimize ga
        if ctx.accounts.state.paused {
            return err!(MarinadeError::ProgramIsPaused);
        }
        ctx.accounts.process(stake_index, validator_index)
    }
    pub fn update_deactivated(ctx: Context<UpdateDeactivated>, stake_in
        //check_context(&ctx)?; // Removed check_context to optimize ga
        if ctx.accounts.state.paused {
            return err!(MarinadeError::ProgramIsPaused);
        }
        ctx.accounts.process(stake_index)
    }

    pub fn deactivate_stake(
        ctx: Context<DeactivateStake>,
        stake_index: u32,
        validator_index: u32,
    ) -> Result<()> {
        //check_context(&ctx)?; // Removed check_context to optimize ga
        if ctx.accounts.state.paused {
            return err!(MarinadeError::ProgramIsPaused);
        }
        ctx.accounts.process(stake_index, validator_index)
    }

    pub fn emergency_unstake(
        ctx: Context<EmergencyUnstake>,
        stake_index: u32,
        validator_index: u32,
    ) -> Result<()> {
        //check_context(&ctx)?; // Removed check_context to optimize ga
        if ctx.accounts.state.paused {
```

```
            return err!(MarinadeError::ProgramIsPaused);
        }
        ctx.accounts.process(stake_index, validator_index)
    }

    pub fn partial_unstake(
        ctx: Context<PartialUnstake>,
        stake_index: u32,
        validator_index: u32,
        desired_unstake_amount: u64,
    ) -> Result<()> {
        //check_context(&ctx)?; // Removed check_context to optimize ga
        if ctx.accounts.state.paused {
            return err!(MarinadeError::ProgramIsPaused);
        }
        ctx.accounts
            .process(stake_index, validator_index, desired_unstake_amou
    }

    pub fn merge_stakes(
        ctx: Context<MergeStakes>,
        destination_stake_index: u32,
        source_stake_index: u32,
        validator_index: u32,
    ) -> Result<()> {
        //check_context(&ctx)?; // Removed check_context to optimize ga
        if ctx.accounts.state.paused {
            return err!(MarinadeError::ProgramIsPaused);
        }
        ctx.accounts
            .process(destination_stake_index, source_stake_index, valid
    }

    pub fn redelegate(
        ctx: Context<ReDelegate>,
        stake_index: u32,
        source_validator_index: u32,
        dest_validator_index: u32,
    ) -> Result<()> {
        //check_context(&ctx)?; // Removed check_context to optimize ga
        if ctx.accounts.state.paused {
            return err!(MarinadeError::ProgramIsPaused);
        }
        ctx.accounts
            .process(stake_index, source_validator_index, dest_validato
```

```
    }

    // emergency pauses the contract
    pub fn pause(ctx: Context<EmergencyPause>) -> Result<()> {
        //check_context(&ctx)?; // Removed check_context to optimize ga
        // No paused check needed for pause instruction
        ctx.accounts.pause()
    }

    // resumes the contract
    pub fn resume(ctx: Context<EmergencyPause>) -> Result<()> {
        //check_context(&ctx)?; // Removed check_context to optimize ga
        // No paused check needed for resume instruction
        ctx.accounts.resume()
    }

    // immediate withdraw of an active stake account - feature can be e
    pub fn withdraw_stake_account(
        ctx: Context<WithdrawStakeAccount>,
        stake_index: u32,
        validator_index: u32,
        msol_amount: u64,
        beneficiary: Pubkey,
    ) -> Result<()> {
        //check_context(&ctx)?; // Removed check_context to optimize ga
        if ctx.accounts.state.paused {
            return err!(MarinadeError::ProgramIsPaused);
        }
        ctx.accounts
            .process(stake_index, validator_index, msol_amount, benefic
    }

    pub fn realloc_validator_list(ctx: Context<ReallocValidatorList>, c
        //check_context(&ctx)?; // Removed check_context to optimize ga
        if ctx.accounts.state.paused {
            return err!(MarinadeError::ProgramIsPaused);
        }
        // Add check here to ensure only admin can call this function
        require_keys_eq!(ctx.accounts.state.admin_authority, ctx.accoun
        ctx.accounts.process(capacity)
    }

    pub fn realloc_stake_list(ctx: Context<ReallocStakeList>, capacity:
        //check_context(&ctx)?; // Removed check_context to optimize ga
        if ctx.accounts.state.paused {
```

```
            return err!(MarinadeError::ProgramIsPaused);
        }
        // Add check here to ensure only admin can call this function
        require_keys_eq!(ctx.accounts.state.admin_authority, ctx.accoun
        ctx.accounts.process(capacity)
    }
}
```

```
// File: checks.rs
use crate::MarinadeError;
use anchor_lang::prelude::*;
use anchor_lang::solana_program::stake::state::StakeState;
use anchor_spl::token::{Mint, TokenAccount};

pub fn check_owner_program<'info, A: ToAccountInfo<'info>>(
    account: &A,
    owner: &Pubkey,
    field_name: &str,
) -> Result<()> {
    let actual_owner = account.to_account_info().owner;
    if actual_owner == owner {
        Ok(())
    } else {
        msg!(
            "Invalid {} owner_program: expected {} got {}",
            field_name,
            owner,
            actual_owner
        );
        Err(Error::from(ProgramError::InvalidArgument)
            .with_account_name(field_name)
            .with_pubkeys((*actual_owner, *owner))
            .with_source(source!()))
    }
}

pub fn check_mint_authority(mint: &Mint, mint_authority: &Pubkey, field
    if mint.mint_authority.contains(mint_authority) {
        Ok(())
    } else {
        msg!(
            "Invalid {} mint authority {}. Expected {}",
            field_name,
            mint.mint_authority.unwrap_or_default(),
```

```
            mint_authority
        );
        Err(Error::from(ProgramError::InvalidAccountData).with_source(s
    }
}

pub fn check_freeze_authority(mint: &Mint, field_name: &str) -> Result<
    if mint.freeze_authority.is_none() {
        Ok(())
    } else {
        msg!("Mint {} must have freeze authority not set", field_name);
        Err(Error::from(ProgramError::InvalidAccountData).with_source(s
    }
}

pub fn check_mint_empty(mint: &Mint, field_name: &str) -> Result<()> {
    if mint.supply == 0 {
        Ok(())
    } else {
        msg!("Non empty mint {} supply: {}", field_name, mint.supply);
        Err(Error::from(ProgramError::InvalidArgument).with_source(sour
    }
}

pub fn check_token_mint(token: &TokenAccount, mint: &Pubkey, field_name
    if token.mint == *mint {
        Ok(())
    } else {
        msg!(
            "Invalid token {} mint {}. Expected {}",
            field_name,
            token.mint,
            mint
        );
        Err(Error::from(ProgramError::InvalidAccountData).with_source(s
    }
}

pub fn check_token_owner(token: &TokenAccount, owner: &Pubkey, field_na
    if token.owner == *owner {
        Ok(())
    } else {
        msg!(
            "Invalid token account {} owner {}. Expected {}",
            field_name,
```

```
            token.owner,
            owner
        );
        Err(Error::from(ProgramError::InvalidAccountData).with_source(s
    }
}

// check that the account is delegated and to the right validator
// also that the stake amount is updated
pub fn check_stake_amount_and_validator(
    stake_state: &StakeState,
    expected_stake_amount: u64,
    validator_vote_pubkey: &Pubkey,
) -> Result<()> {
    let currently_staked = if let Some(delegation) = stake_state.delega
        require_keys_eq!(
            delegation.voter_pubkey,
            *validator_vote_pubkey,
            MarinadeError::WrongValidatorAccountOrIndex
        );
        delegation.stake
    } else {
        return err!(MarinadeError::StakeNotDelegated);
    };
    // do not allow to operate on an account where last_update_delegate
    if currently_staked != expected_stake_amount {
        msg!(
            "Operation on a stake account not
```