SMART CONTRACT	https://github.com/blasmorkai/cw-plus-bm/tree/main/contracts/cw4-stake

Scope: contract.rs (execute entry point)

CODE REVIEW

contract.rs

```
// And declare a custom Error variant for the ones where you will want to make use of
it
#[cfg_attr(not(feature = "library"), entry_point)]
pub fn execute(
   deps: DepsMut,
   env: Env,
   info: MessageInfo,
   msg: ExecuteMsg,
) -> Result<Response, ContractError> {
   let api = deps.api;
   match msg {
       ExecuteMsg::UpdateAdmin { admin } => {
           Ok(ADMIN.execute_update_admin(deps, info, maybe_addr(api, admin)?)?)
       }
       ExecuteMsg::AddHook { addr } => {
           Ok(HOOKS.execute_add_hook(&ADMIN, deps, info, api.addr_validate(&addr)?)?)
       }
       ExecuteMsg::RemoveHook { addr } => {
           Ok(HOOKS.execute_remove_hook(&ADMIN, deps, info,
api.addr_validate(&addr)?)?)
       }
       // Info.funds is passed as a Balance parameter to make use of its helper
functions.
       // Balance is of type Native(NativeBalance) -- struct NativeBalance(pub
Vec<Coin>);
       ExecuteMsg::Bond {} => execute_bond(deps, env, Balance::from(info.funds),
info.sender),
       ExecuteMsg::Unbond { tokens: amount } => execute_unbond(deps, env, info,
amount),
       ExecuteMsg::Claim {} => execute_claim(deps, env, info),
       // execute_receive will end up calling execute_bond with balance :
Cw20CoinVerified
       // execute_bond (.... balance: Cw20CoinVerified {contract_address,
Cw20ReceiveMsg...amount }
```

```
        //                       sender: user that requested the cw20 contract to send
this)
        ExecuteMsg::Receive(msg) => execute_receive(deps, env, info, msg),
    }
}

pub fn execute_bond(
    deps: DepsMut,
    env: Env,
    amount: Balance,
    sender: Addr,
) -> Result<Response, ContractError> {
    let cfg = CONFIG.load(deps.storage)?;

    // ensure the sent denom was proper
    // NOTE: those clones are not needed (if we move denom, we return early),
    // but the compiler cannot see that (yet...)
     // enum Denom {Native(String), Cw20(Addr),}
    // enum Balance { Native(NativeBalance), Cw20(Cw20CoinVerified),}
        // struct NativeBalance(pub Vec<Coin>);
        // struct Cw20CoinVerified {pub address: Addr, pub amount: Uint128,}

    let amount = match (&cfg.denom, &amount) {
        // If we did set up the config denom as String and the info.funds transferred
with the message is a Vec<Coin>
        (Denom::Native(want), Balance::Native(have)) => must_pay_funds(have, want),
        // If we did set up the config denom as an Addr, and the balance is {Addr,
Uint128}
        // This will deal with ExecuteMsg::Receive(msg) => execute_receive(deps, env,
info, msg) =>
        (Denom::Cw20(want), Balance::Cw20(have)) => {
            if want == &have.address {          // Making sure that balance.address is
the same set up on CONFIG.denom.
                Ok(have.amount)                 // Returning balance.amount
            } else {
                Err(ContractError::InvalidDenom(want.into()))
            }
        }
        _ => Err(ContractError::MixedNativeAndCw20(
            "Invalid address or denom".to_string(),
        )),
    }?;

    // If we get here we already know that the denom sent is the one set on CONFIG. We
only have to worry on what to do with amount.
    // update the sender's stake. Getting back the new total amount of tokens bonded
by that user.
    let new_stake = STAKE.update(deps.storage, &sender, |stake| -> StdResult<_> {
            Ok(stake.unwrap_or_default() + amount)      // If the sender has not staked
before it triggers .unwrap_or_DEFAULT()
    })?;

    // MEMBERS and TOTAL is updated,
    // List of Submessages are returned to inform registered Hook addresses of sender
weight changes.
    let messages = update_membership(
        deps.storage,
        sender.clone(),
        new_stake,
        &cfg,
        env.block.height,
    )?;

    Ok(Response::new()
        .add_submessages(messages)
        .add_attribute("action", "bond")
        .add_attribute("amount", amount)
        .add_attribute("sender", sender))
}

pub fn execute_receive(
    deps: DepsMut,
    env: Env,
    info: MessageInfo,
    wrapper: Cw20ReceiveMsg,
) -> Result<Response, ContractError> {
    // info.sender is the address of the cw20 contract (that re-sent this message).
    // wrapper.sender is the address of the user that requested the cw20 contract to
send this.
    // This cannot be fully trusted (the cw20 contract can fake it), so only use it
for actions
    // in the address's favor (like paying/bonding tokens, not withdrawls)
    let msg: ReceiveMsg = from_slice(&wrapper.msg)?;
    // We create a Balance of type Cw20CoinVerified {contract_address,
Cw20ReceiveMsg...amount }
    let balance = Balance::Cw20(Cw20CoinVerified {
        address: info.sender,
        amount: wrapper.amount,
    });
    let api = deps.api;
    match msg {
        ReceiveMsg::Bond {} => {
            // execute_bond (.... balance: Cw20CoinVerified {contract_address,
Cw20ReceiveMsg...amount }
            //                    sender: user that requested the cw20 contract to
send this)
```

```rust
            execute_bond(deps, env, balance, api.addr_validate(&wrapper.sender)?)
        }
    }
}

pub fn execute_unbond(
    deps: DepsMut,
    env: Env,
    info: MessageInfo,
    amount: Uint128,
) -> Result<Response, ContractError> {
    // reduce the sender's stake - aborting if insufficient as STAKE value is
unsigned.
    let new_stake = STAKE.update(deps.storage, &info.sender, |stake| -> StdResult<_> {
        Ok(stake.unwrap_or_default().checked_sub(amount)?)
    })?;

    // provide them a claim
    let cfg = CONFIG.load(deps.storage)?;
    // Map<&Addr, Vec<Claim>> The info.sender will get its Vec<Claim> updated by
adding/pushing a Claim { amount, release_at }
    CLAIMS.create_claim(
        deps.storage,
        &info.sender,
        amount,
        cfg.unbonding_period.after(&env.block),
    )?;

    // Update MEMBERS, TOTAL and return a Vec<SubMsg> to alert the Hooks
    let messages = update_membership(
        deps.storage,
        info.sender.clone(),
        new_stake,
        &cfg,
        env.block.height,
    )?;

    Ok(Response::new()
        .add_submessages(messages)
        .add_attribute("action", "unbond")
        .add_attribute("amount", amount)
        .add_attribute("sender", info.sender))
}

// If only one coin is sent with the message and the denom is the one set up in the
staking contract, it returns the amount to bond.
pub fn must_pay_funds(balance: &NativeBalance, denom: &str) -> Result<Uint128,
ContractError> {
   match balance.0.len() {                              // How many coins have been
transferred? How many elements in Vec<Coin>?
       0 => Err(ContractError::NoFunds {}),
       1 => {
           let balance = &balance.0;         // balance -> Vec<Coin>
           let payment = balance[0].amount;     // payment -> the amount from the
first coin transferred
           if balance[0].denom == denom {                  // Making sure that the first
coin denom is the denom wanting to stake
               Ok(payment)                                 // the amount of coins
transferred/to_be_bonded is returned
           } else {
               Err(ContractError::MissingDenom(denom.to_string())) //The coin
transferred is not the one registed in CONFIG.
           }
       }
       _ => Err(ContractError::ExtraDenoms(denom.to_string())), // More than one coin
transferred with the message.
   }
}

// Update MEMBERS, TOTAL and return a Vec<SubMsg> to alert the Hooks of the weight
changes
fn update_membership(
   storage: &mut dyn Storage,
   sender: Addr,
   new_stake: Uint128,
   cfg: &Config,
   height: u64,
) -> StdResult<Vec<SubMsg>> {
   // update their membership weight
   let new = calc_weight(new_stake, cfg);
   let old = MEMBERS.may_load(storage, &sender)?;

   // short-circuit if no change
   if new == old {
       return Ok(vec![]);
   }
   // otherwise, record change of weight
   match new.as_ref() {
       Some(w) => MEMBERS.save(storage, &sender, w, height),  // The weight is saved
for a certain sender and height
       None => MEMBERS.remove(storage, &sender, height),                  // No
weight so... delete entry for sender and height
   }?;

   // update total. Increasing it by the difference of current and previous balance.
   TOTAL.update(storage, |total| -> StdResult<_> {
```

```rust
        Ok(total + new.unwrap_or_default() - old.unwrap_or_default())
    })?;

    // MemberDiff::new(...) returns MemberDiff { key: addr.into(), old: old_weight,
new: new_weight, }
    // alert the hooks
    let diff = MemberDiff::new(sender, old, new);

    // struct MemberChangedHookMsg {diffs: Vec<MemberDiff>}
    // MemberChangedHookMsg.one(diff: MemberDiff) -> Self { MemberChangedHookMsg {
diffs: vec![diff] }}

    // It SEEMS it creates a submessage for all the hook addresses informing of
MemberDiff { key: addr.into(), old: old_weight, new: new_weight, }
    // These submessages will be returned to the caller function to send them from
there.
    HOOKS.prepare_hooks(storage, |h| {
        MemberChangedHookMsg::one(diff.clone())
            .into_cosmos_msg(h)
            .map(SubMsg::new)
    })
}

// Getting a weight based on the total amount of tokens bonded/staked
// Tokens per weight is not affected by the number of tokens bonded, it is a
constant.
fn calc_weight(stake: Uint128, cfg: &Config) -> Option<u64> {
    if stake < cfg.min_bond {
        None
    } else {
        let w = stake.u128() / (cfg.tokens_per_weight.u128());
        Some(w as u64)
    }
}

pub fn execute_claim(
    deps: DepsMut,
    env: Env,
    info: MessageInfo,
) -> Result<Response, ContractError> {
    // CLAIMS.claim_tokens() - Claims(Map<&Addr, Vec<Claim>>)
    // Update CLAIMS , addr:info.sender will only keep the Claims that have not
expired registered in CLAIMS.
    // Return the addition of the tokens that have already expired and whose claims
are no longer stored in Vec<Claim>
    let release = CLAIMS.claim_tokens(deps.storage, &info.sender, &env.block, None)?;
    if release.is_zero() {
        return Err(ContractError::NothingToClaim {});
    }

    let config = CONFIG.load(deps.storage)?;
    let (amount_str, message) = match &config.denom {
        // Config { ... pub denom: Denom, ..} //enum Denom {Native(String),
Cw20(Addr),}
        //  The contract initially received tokens on the ExecuteMsg with info.funds
(No Cw20ReceiveMsg)
        Denom::Native(denom) => {
            let amount_str = coin_to_string(release, denom.as_str());
            let amount = coins(release.u128(), denom);

            // Send the tokens to info.sender, token claimer
            let message = SubMsg::new(BankMsg::Send {
                to_address: info.sender.to_string(),
                amount,
            });
            (amount_str, message)
        }
        //  The contract initially received tokens on a Cw20ReceiveMsg, from another
contract
        //  addr is the contract_address
        Denom::Cw20(addr) => {
            let amount_str = coin_to_string(release, addr.as_str());
            let transfer = Cw20ExecuteMsg::Transfer {
                recipient: info.sender.clone().into(),
                amount: release,
            };

            // Send a Cw20ExecuteMsg::Transfer to a contract that should relay it to
info.sender
            let message = SubMsg::new(WasmMsg::Execute {
                contract_addr: addr.into(),
                msg: to_binary(&transfer)?,
                funds: vec![],
            });
            (amount_str, message)
        }
    };

    Ok(Response::new()
        .add_submessage(message)
        .add_attribute("action", "claim")
        .add_attribute("tokens", amount_str)
        .add_attribute("sender", info.sender))
}

#[inline]
```

```rust
fn coin_to_string(amount: Uint128, denom: &str) -> String {
    format!("{} {}", amount, denom)
}
```

GENERAL QUESTIONS

1. **What are the concepts (borrowing, ownership, vectors etc)**
   The Concepts in the code are Structs, Item, Map, Vectors, Tupples, Ownerships, Enum (Options, Results), Coins, Claims, Duration.
   When setting up the state, a few structures like Admin, Hooks, Claims are based on cw_storage_plus Items and Maps.

2. **What is the organization?**
   Entry points and the functions taking care of its messages.

3. **What is the contract doing? What is the mechanism?**
   Implementation of bond, unbond, claim and receive ExecuteMsgs on the staking contract.

4. **How could it be better? More efficient? Safer?**
   Nothing that I can think of.