SMART CONTRACT	https://github.com/blasmorkai/cw-plus-bm/tree/main/contracts/cw20-base

Scope: Contract instantiation and review of states and messages. (state.rs, msg.rs, contract.rs.)
Additional crate reviewed: https://github.com/blasmorkai/cw-plus-bm/packages/cw20/src/query.rs (query.rs)

CODE REVIEW

State.rs

```
#[derive(Serialize, Deserialize, Clone, PartialEq, JsonSchema, Debug)]
#[serde(rename_all = "snake_case")]
pub struct TokenInfo {
    pub name: String,
    pub symbol: String,
    pub decimals: u8,
    pub total_supply: Uint128,
    pub mint: Option<MinterData>,
}

#[derive(Serialize, Deserialize, Clone, PartialEq, JsonSchema, Debug)]
pub struct MinterData {
    pub minter: Addr,
    /// cap is how many more tokens can be issued by the minter
    pub cap: Option<Uint128>,
}

impl TokenInfo {
    // mint: Option<MinterData>
    // as_ref() casts Option<MinterData> to Option<&MinterData>
    // and_then(|| ) returns None if the option is None, otherwise calls f with the wrapped value and returns the
result
    pub fn get_cap(&self) -> Option<Uint128> {
        self.mint.as_ref().and_then(|v| v.cap)
    }
}

// Token for this cw20 contract is defined
pub const TOKEN_INFO: Item<TokenInfo> = Item::new("token_info");
// pub struct MarketingInfoResponse { project: Option<String>,description: Option<String>,logo:
Option<LogoInfo>,marketing: Option<Addr>,}
pub const MARKETING_INFO: Item<MarketingInfoResponse> = Item::new("marketing_info");
// pub enum Logo {Url(String),Embedded(EmbeddedLogo),}
pub const LOGO: Item<Logo> = Item::new("logo");

pub const BALANCES: Map<&Addr, Uint128> = Map::new("balance");
```

```
// pub struct AllowanceResponse {allowance Uint128, expires: Expiration}
// pub enum Expiration {AtHeight(u64),AtTime(Timestamp),Never {},}
pub const ALLOWANCES: Map<(&Addr, &Addr), AllowanceResponse> = Map::new("allowance");
// TODO: After https://github.com/CosmWasm/cw-plus/issues/670 is implemented, replace this with a
`MultiIndex` over `ALLOWANCES`
pub const ALLOWANCES_SPENDER: Map<(&Addr, &Addr), AllowanceResponse> =
    Map::new("allowance_spender");
```

Msg.rs

```
#[derive(Serialize, Deserialize, JsonSchema, Debug, Clone, PartialEq)]
pub struct InstantiateMarketingInfo {
    pub project: Option<String>,
    pub description: Option<String>,
    pub marketing: Option<String>,
    pub logo: Option<Logo>,
}

#[derive(Serialize, Deserialize, JsonSchema, Debug, Clone, PartialEq)]
#[cfg_attr(test, derive(Default))]
pub struct InstantiateMsg {
    pub name: String,
    pub symbol: String,
    pub decimals: u8,
    pub initial_balances: Vec<Cw20Coin>,
    pub mint: Option<MinterResponse>,
    pub marketing: Option<InstantiateMarketingInfo>,
}

impl InstantiateMsg {
    // Remaining minting cap if it exists. {minter, cap} -> {cap}
    pub fn get_cap(&self) -> Option<Uint128> {
        self.mint.as_ref().and_then(|v| v.cap)
    }

    // Check name and symbol
    pub fn validate(&self) -> StdResult<()> {
        // Check name, symbol, decimals
        if !self.has_valid_name() {
            return Err(StdError::generic_err(
                "Name is not in the expected format (3-50 UTF-8 bytes)",
            ));
        }
        if !self.has_valid_symbol() {
            return Err(StdError::generic_err(
                "Ticker symbol is not in expected format [a-zA-Z\\-]{3,12}",
            ));
        }
        if self.decimals > 18 {
            return Err(StdError::generic_err("Decimals must not exceed 18"));
        }
        Ok(())
    }

    fn has_valid_name(&self) -> bool {
        // returns a byte slice, then its length is challenged. Allowed: [3,50]
        let bytes = self.name.as_bytes();
        if bytes.len() < 3 || bytes.len() > 50 {
            return false;
        }
        true
    }

    fn has_valid_symbol(&self) -> bool {
        // returns a byte slice, then we make sure the symbols are allowed. Allowed [3,12]
        let bytes = self.symbol.as_bytes();
        if bytes.len() < 3 || bytes.len() > 12 {
            return false;
        }
        for byte in bytes.iter() {
            if (*byte != 45) && (*byte < 65 || *byte > 90) && (*byte < 97 || *byte > 122) {
                return false;
            }
        }
        true
    }
}

#[derive(Serialize, Deserialize, Clone, Debug, PartialEq, JsonSchema)]
#[serde(rename_all = "snake_case")]
pub enum QueryMsg {
    /// Returns the current balance of the given address, 0 if unset.
    /// Return type: BalanceResponse.   // struct BalanceResponse { balance: Uint128,}
    Balance { address: String },
    /// Returns metadata on the contract - name, decimals, supply, etc.
    /// Return type: TokenInfoResponse.   // struct TokenInfoResponse {name: String, symbol: String, decimals:
u8, total_supply: Uint128,}
    TokenInfo {},
    /// Only with "mintable" extension.
    /// Returns who can mint and the hard cap on maximum tokens after minting.
    /// Return type: MinterResponse.      // struct MinterResponse { minter: String, cap: Option<Uint128>,}
    Minter {},
    /// Only with "allowance" extension.
    /// Returns how much spender can use from owner account, 0 if unset.
    /// Return type: AllowanceResponse.   // struct AllAllowancesResponse {allowances: Vec<AllowanceInfo>,}
    ///                                   // struct AllowanceInfo { spender: String, allowance: Uint128, expires: Expiration,}
    Allowance { owner: String, spender: String },
    /// Only with "enumerable" extension (and "allowances")
    /// Returns all allowances this owner has approved. Supports pagination.
    /// Return type: AllAllowancesResponse.   // struct AllAllowancesResponse {allowances:
Vec<AllowanceInfo>,}
    ///                                       // struct AllowanceInfo { spender: String, allowance: Uint128, expires: Expiration,}
    AllAllowances {
        owner: String,
        start_after: Option<String>,
        limit: Option<u32>,
    },
    /// Only with "enumerable" extension (and "allowances")
    /// Returns all allowances this spender has been granted. Supports pagination.
    /// Return type: AllSpenderAllowancesResponse.  // struct AllSpenderAllowancesResponse {allowances:
Vec<SpenderAllowanceInfo>,}
    ///                                         // struct SpenderAllowanceInfo {owner: String, allowance: Uint128, expires:
Expiration,}
    AllSpenderAllowances {
        spender: String,
        start_after: Option<String>,
        limit: Option<u32>,
    },
    /// Only with "enumerable" extension
    /// Returns all accounts that have balances. Supports pagination.
    /// Return type: AllAccountsResponse.          // struct AllAccountsResponse {accounts: Vec<String>}
    AllAccounts {
        start_after: Option<String>,
        limit: Option<u32>,
    },
    /// Only with "marketing" extension
    /// Returns more metadata on the contract to display in the client:
    /// - description, logo, project url, etc.
    /// Return type: MarketingInfoResponse
    MarketingInfo {},
    /// Only with "marketing" extension
    /// Downloads the embedded logo data (if stored on chain). Errors if no logo data is stored for this
    /// contract.
    /// Return type: DownloadLogoResponse.
    DownloadLogo {},
}

#[derive(Serialize, Deserialize, JsonSchema)]
pub struct MigrateMsg {}

#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    fn validate_instantiatemsg_name() {
        // name length allowed [3,50]
        // Too short
        let mut msg = InstantiateMsg {
            name: str::repeat("a", 2),
            ..InstantiateMsg::default()
        };
        assert!(!msg.has_valid_name());

        // In the correct length range
        msg.name = str::repeat("a", 3);
        assert!(msg.has_valid_name());

        // Too long
        msg.name = str::repeat("a", 51);
        assert!(!msg.has_valid_name());
    }

    #[test]
    fn validate_instantiatemsg_symbol() {
        // symbol length Allowed [3,12]
        // Too short
        let mut msg = InstantiateMsg {
            symbol: str::repeat("a", 2),
            ..InstantiateMsg::default()
        };
        assert!(!msg.has_valid_symbol());

        // In the correct length range
        msg.symbol = str::repeat("a", 3);
        assert!(msg.has_valid_symbol());

        // Too long
        msg.symbol = str::repeat("a", 13);
        assert!(!msg.has_valid_symbol());

        // Legal chars [65,90] U [97,122]
        // Has illegal char.
        let illegal_chars = [[64u8], [91u8], [123u8]];
        illegal_chars.iter().for_each(|c| {
            let c = std::str::from_utf8(c).unwrap();
            // the character has to be repeated at least three times to be able to use msg.has_valid_sympol() and
not refused by length
            msg.symbol = str::repeat(c, 3);
            assert!(!msg.has_valid_symbol());
        });
    }
}
```

TO BETTER UNDERSTAND THIS MESSAGES, WE STUDY AS WELL:
pack

cw20 - query.rs

```
#[derive(Serialize, Deserialize, Clone, PartialEq, JsonSchema, Debug)]
#[serde(rename_all = "snake_case")]
pub enum Cw20QueryMsg {
    /// Returns the current balance of the given address, 0 if unset.
    /// Return type: BalanceResponse.   // struct BalanceResponse { balance: Uint128,}
    Balance { address: String },
    /// Returns metadata on the contract - name, decimals, supply, etc.
    /// Return type: TokenInfoResponse.  // struct TokenInfoResponse {name: String, symbol: String, decimals:
u8, total_supply: Uint128,}
    TokenInfo {},
    /// Only with "allowance" extension.
    /// Returns how much spender can use from owner account, 0 if unset.
    /// Return type: AllowanceResponse.  // pub struct AllowanceResponse {allowance Uint128, expires:
Expiration}
```

```
                                    // pub enum Expiration {AtHeight(u64),AtTime(Timestamp),Never {},}
    Allowance { owner: String, spender: String },
    /// Only with "mintable" extension.
    /// Returns who can mint and the hard cap on maximum tokens after minting.
    /// Return type: MinterResponse.    // struct MinterResponse { minter: String, cap: Option<Uint128>,}
    Minter {},
    /// Only with "marketing" extension
    /// Returns more metadata on the contract to display in the client:
    /// - description, logo, project url, etc.
    /// Return type: MarketingInfoResponse.   // pub struct MarketingInfoResponse { project:
Option<String>,description: Option<String>,logo: Option<LogoInfo>,marketing: Option<Addr>,}
    MarketingInfo {},
    /// Only with "marketing" extension
    /// Downloads the embedded logo data (if stored on chain). Errors if no logo data stored for
    /// this contract.
    /// Return type: DownloadLogoResponse.  // pub struct DownloadLogoResponse { pub mime_type: String,
pub data: Binary,}
    DownloadLogo {},
    /// Only with "enumerable" extension (and "allowances")
    /// Returns all allowances this owner has approved. Supports pagination.
    /// Return type: AllAllowancesResponse.     // struct AllAllowancesResponse {allowances:
Vec<AllowanceInfo>,}
                                         // struct AllowanceInfo { spender: String, allowance: Uint128, expires:
Expiration,}
    AllAllowances {
        owner: String,
        start_after: Option<String>,
        limit: Option<u32>,
    },
    /// Only with "enumerable" extension
    /// Returns all accounts that have balances. Supports pagination.
    /// Return type: AllAccountsResponse.     // struct AllAccountsResponse {accounts: Vec<String>}
    AllAccounts {
        start_after: Option<String>,
        limit: Option<u32>,
    },
}

#[derive(Serialize, Deserialize, Clone, PartialEq, JsonSchema, Debug)]
pub struct BalanceResponse {
    pub balance: Uint128,
}

#[derive(Serialize, Deserialize, Clone, PartialEq, JsonSchema, Debug)]
pub struct TokenInfoResponse {
    pub name: String,
    pub symbol: String,
    pub decimals: u8,
    pub total_supply: Uint128,
}

#[derive(Serialize, Deserialize, Clone, PartialEq, JsonSchema, Debug, Default)]
pub struct AllowanceResponse {
    pub allowance: Uint128,
```

```rust
    pub expires: Expiration,
}

#[derive(Serialize, Deserialize, Clone, PartialEq, JsonSchema, Debug)]
pub struct MinterResponse {
    pub minter: String,
    /// cap is a hard cap on total supply that can be achieved by minting.
    /// Note that this refers to total_supply.
    /// If None, there is unlimited cap.
    pub cap: Option<Uint128>,
}
```

Contract.rs

```rust
// version info for migration info
const CONTRACT_NAME: &str = "crates.io:cw20-base";
const CONTRACT_VERSION: &str = env!("CARGO_PKG_VERSION");

const LOGO_SIZE_CAP: usize = 5 * 1024;


#[cfg_attr(not(feature = "library"), entry_point)]
pub fn instantiate(
    mut deps: DepsMut,
    _env: Env,
    _info: MessageInfo,
    msg: InstantiateMsg,
) -> Result<Response, ContractError> {
    set_contract_version(deps.storage, CONTRACT_NAME, CONTRACT_VERSION)?;
    // check valid token info
    msg.validate()?;
    // create initial accounts
    let total_supply = create_accounts(&mut deps, &msg.initial_balances)?;

    // check that total supply has not exceeded the minting cap
    if let Some(limit) = msg.get_cap() {
        if total_supply > limit {
            return Err(StdError::generic_err("Initial supply greater than cap").into());
        }
    }

    // check that the minter address (if set) is valid
    let mint = match msg.mint {
        Some(m) => Some(MinterData {
            minter: deps.api.addr_validate(&m.minter)?,
            cap: m.cap,
        }),
        None => None,
    };
    // store token info
    let data = TokenInfo {
        name: msg.name,
        symbol: msg.symbol,
        decimals: msg.decimals,
        total_supply,
        mint,
    };
    TOKEN_INFO.save(deps.storage, &data)?;

// If the option with marketing has info, we verify it and save it into LOGO and MARKETING_INFO state
    if let Some(marketing) = msg.marketing {
        let logo = if let Some(logo) = marketing.logo {
            verify_logo(&logo)?;
            LOGO.save(deps.storage, &logo)?;

            match logo {                    // The logo will be used to build MarketingInfoResponse {}
                Logo::Url(url) => Some(LogoInfo::Url(url)),
                Logo::Embedded(_) => Some(LogoInfo::Embedded),
            }
        } else {
            None
        };

        let data = MarketingInfoResponse {
            project: marketing.project,
            description: marketing.description,
            marketing: marketing
                .marketing
                .map(|addr| deps.api.addr_validate(&addr))
                .transpose()?,
            logo,
        };
        MARKETING_INFO.save(deps.storage, &data)?;
    }

    Ok(Response::default())
}

pub fn create_accounts(
    deps: &mut DepsMut,
    accounts: &[Cw20Coin],
) -> Result<Uint128, ContractError> {
    // struct Cw20Coin {address: String, amount: Uint128,}
    // The accounts are saved in the BALANCES Map
    validate_accounts(accounts)?;

    let mut total_supply = Uint128::zero();
    for row in accounts {
        let address = deps.api.addr_validate(&row.address)?;
        BALANCES.save(deps.storage, &address, &row.amount)?;
        total_supply += row.amount;
    }
    Ok(total_supply)
}

pub fn validate_accounts(accounts: &[Cw20Coin]) -> Result<(), ContractError> {
    // struct Cw20Coin {address: String, amount: Uint128,}
    // Create an array of the address, sorting and removing consecutive repeated elements
    let mut addresses = accounts.iter().map(|c| &c.address).collect::<Vec<_>>();
    addresses.sort();
    addresses.dedup();

    // if any addresses has been removed, there was more than one entry for at least one account
    if addresses.len() != accounts.len() {
        Err(ContractError::DuplicateInitialBalanceAddresses {})
    } else {
        Ok(())
    }
}
```

GENERAL QUESTIONS

1. **What are the concepts (borrowing, ownership, vectors etc)**
   From previous code reviews, this code implements methods on the struct InstantiateMsg that help to validate that the parameters provided follow certain rules/standards.

2. **What is the organization?**
   Cosmwasm contract that makes use of structs and enums defined in the cw20 contract.

3. **What is the contract doing? What is the mechanism?**
   It creates a token ledger based on information provided (name, symbol, decimals, total_supply, minter/minting cap (optional)) validating addresses and parameters. Balances and token info is stored in the blockchain.
   When the token ledger is created, an optional initial ledger information addresses/balance is provided (validating those accounts).

4. **How could it be better? More efficient? Safer?**
   No improvement can be suggested based on my current knowledge.