Armors Labs

Sword Token (SWDTKN)

Smart Contract Audit

Sword Token (SWDTKN) Audit Summary

Project name : Sword Token (SWDTKN)

Project address: None

Audit URL : https://solscan.io/token/BkCwEZ5Zbv85dAT9rD6mu19i26RY72gBWETr1bb8Wssv

Commit : None

Project target : Sword Token (SWDTKN) Audit

Blockchain : Solana

Test result : PASSED

Audit Info

Audit NO : 0X202208260006

Audit Team : Armors Labs

Audit Proofreading: https://armors.io/#project-cases

Sword Token (SWDTKN) Audit

The Sword Token (SWDTKN) team asked us to review and audit their Sword Token (SWDTKN) project. We looked at the code and now publish our results.

Here is our assessment and recommendations, in order of importance.

Document information

Name	Auditor	Version	Date
Sword Token (SWDTKN) Audit	Rock, Sophia, Rushairer, Rico, David, Alice	1.0.0	2022-08-26

Audit results

1. This is fungible token.
2. Created by program-id(TokenkegQfeZyiNwAJbNbGKPFXCWuBvf9Ss623VQ5DA) https://spl.solana.com/token
3. mint permission disabled https://explorer.solana.com/tx/WuH159XBWjM9A46jPN5TQ3AiG3nhzvZc38u7QZa8BoA7k9FXXQbrZvxqSaGxYou7Up9kShEY5bX2kvJrndpi4gr

Note that as of the date of publishing, the above review reflects the current understanding of known security patterns as they relate to the Sword Token (SWDTKN). The above should not be construed as investment advice.

Based on the widely recognized security status of the current underlying blockchain and program, this audit report is valid for 3 months from the date of output.

Disclaimer

Armors Labs Reports is not and should not be regarded as an "approval" or "disapproval" of any particular project or team. These reports are not and should not be regarded as indicators of the economy or value of any "product" or "asset" created by any team. Armors do not cover testing or auditing the integration with external program or services (such as Unicrypt, Uniswap, PancakeSwap etc'...)

Armors Labs Reports represent an extensive auditing process intending to help our customers increase the quality of their code while reducing the high level of risk presented by cryptographic tokens and blockchain technology. Armors does not guarantee the safety or functionality of the technology agreed to be analyzed.

Armors Labs postulates that the information provided is not missing, tampered, deleted or hidden. If the information provided is missing, tampered, deleted, hidden or reflected in a way that is not consistent with the actual situation, Armors Labs shall not be responsible for the losses and adverse effects caused. Armors Labs Audits should not be used in any way to make decisions around investment or involvement with any particular project. These reports in no way provide investment advice, nor should be leveraged as investment advice of any sort.

Audited target file

file	md5
./src/processor.rs	17b63a2b4acf5e46368762e1a230de74
./src/entrypoint.rs	5c7b7c676dedd6b4f9c4b304589b04aa
./src/error.rs	eb8c029f1ed7bb9d2cc6e7853d5806c5
./src/lib.rs	44876a6986c16702ebf45609c751e2d3
./src/native_mint.rs	107bd09494956b9446fd77c905a44d7a
./src/state.rs	b721e9aab9050b2a16c90d2b9d8306e7
./src/instruction.rs	2a92fa187b755c9813c860d92dbe6ce8
./Cargo.toml	a0db318087de4182d6816868c3084db5
./Xargo.toml	581da06b21cec82134fd27774623434f
./program-id.md	d162362adc0d54a5af9136426391607e

Vulnerability analysis

Vulnerability distribution

vulnerability level	number
Critical severity	0
High severity	0
Medium severity	0
Low severity	0

Summary of audit results

Vulnerability	status
Re-Entrancy	safe
Arithmetic Over/Under Flows	safe
Replay Attacks	safe
Denial-of-Service (DoS)	safe
Race Conditions / Front Running	safe
Permission restrictions	safe
Variable declaration and scope	safe
External program Referencing	safe
Forged account	safe
Unchecked CALL Return Values	safe
Floating Points and Numerical Precision	safe
Business logic defect	safe

program file

```
//! Program entrypoint

use crate::{error::TokenError, processor::Processor};
use solana_program::{
    account_info::AccountInfo, entrypoint, entrypoint::ProgramResult,
    program_error::PrintProgramError, pubkey::Pubkey,
};

entrypoint!(process_instruction);
fn process_instruction(
    program_id: &Pubkey,
    accounts: &[AccountInfo],
    instruction_data: &[u8],
) -> ProgramResult {
    if let Err(error) = Processor::process(program_id, accounts, instruction_data) {
        // catch the error so we can print it
        error.print::<TokenError>();
        return Err(error);
    }
    Ok(())
}
```

```
//! Error types

use num_derive::FromPrimitive;
use solana_program::{decode_error::DecodeError, program_error::ProgramError};
use thiserror::Error;

/// Errors that may be returned by the Token program.
#[derive(Clone, Debug, Eq, Error, FromPrimitive, PartialEq)]
```

```
pub enum TokenError {
    // 0
    /// Lamport balance below rent-exempt threshold.
    #[error("Lamport balance below rent-exempt threshold")]
    NotRentExempt,
    /// Insufficient funds for the operation requested.
    #[error("Insufficient funds")]
    InsufficientFunds,
    /// Invalid Mint.
    #[error("Invalid Mint")]
    InvalidMint,
    /// Account not associated with this Mint.
    #[error("Account not associated with this Mint")]
    MintMismatch,
    /// Owner does not match.
    #[error("Owner does not match")]
    OwnerMismatch,

    // 5
    /// This token's supply is fixed and new tokens cannot be minted.
    #[error("Fixed supply")]
    FixedSupply,
    /// The account cannot be initialized because it is already being used.
    #[error("Already in use")]
    AlreadyInUse,
    /// Invalid number of provided signers.
    #[error("Invalid number of provided signers")]
    InvalidNumberOfProvidedSigners,
    /// Invalid number of required signers.
    #[error("Invalid number of required signers")]
    InvalidNumberOfRequiredSigners,
    /// State is uninitialized.
    #[error("State is unititialized")]
    UninitializedState,

    // 10
    /// Instruction does not support native tokens
    #[error("Instruction does not support native tokens")]
    NativeNotSupported,
    /// Non-native account can only be closed if its balance is zero
    #[error("Non-native account can only be closed if its balance is zero")]
    NonNativeHasBalance,
    /// Invalid instruction
    #[error("Invalid instruction")]
    InvalidInstruction,
    /// State is invalid for requested operation.
    #[error("State is invalid for requested operation")]
    InvalidState,
    /// Operation overflowed
    #[error("Operation overflowed")]
    Overflow,

    // 15
    /// Account does not support specified authority type.
    #[error("Account does not support specified authority type")]
    AuthorityTypeNotSupported,
    /// This token mint cannot freeze accounts.
    #[error("This token mint cannot freeze accounts")]
    MintCannotFreeze,
    /// Account is frozen; all account operations will fail
    #[error("Account is frozen")]
    AccountFrozen,
    /// Mint decimals mismatch between the client and mint
    #[error("The provided decimals value different from the Mint decimals")]
    MintDecimalsMismatch,
    /// Instruction does not support non-native tokens
    #[error("Instruction does not support non-native tokens")]
    NonNativeNotSupported,
}
impl From<TokenError> for ProgramError {
    fn from(e: TokenError) -> Self {
        ProgramError::Custom(e as u32)
    }
}
impl<T> DecodeError<T> for TokenError {
    fn type_of() -> &'static str {
        "TokenError"
    }
}
```

```
#![deny(missing_docs)]
#![cfg_attr(not(test), forbid(unsafe_code))]

//! An ERC20-like Token program for the Solana blockchain

pub mod error;
pub mod instruction;
pub mod native_mint;
pub mod processor;
pub mod state;

#[cfg(not(feature = "no-entrypoint"))]
mod entrypoint;

// Export current sdk types for downstream users building with a different sdk version
pub use solana_program;
use solana_program::{entrypoint::ProgramResult, program_error::ProgramError, pubkey::Pubkey};

/// Convert the UI representation of a token amount (using the decimals field defined in its mint)
/// to the raw amount
pub fn ui_amount_to_amount(ui_amount: f64, decimals: u8) -> u64 {
    (ui_amount * 10_usize.pow(decimals as u32) as f64) as u64
}

/// Convert a raw amount to its UI representation (using the decimals field defined in its mint)
pub fn amount_to_ui_amount(amount: u64, decimals: u8) -> f64 {
    amount as f64 / 10_usize.pow(decimals as u32) as f64
}

solana_program::declare_id!("TokenkegQfeZyiNwAJbNbGKPFXCWuBvf9Ss623VQ5DA");

/// Checks that the supplied program ID is the correct one for SPL-token
pub fn check_program_account(spl_token_program_id: &Pubkey) -> ProgramResult {
    if spl_token_program_id != &id() {
        return Err(ProgramError::IncorrectProgramId);
    }
    Ok(())
}
```

```
//! State transition types

use crate::instruction::MAX_SIGNERS;
use arrayref::{array_mut_ref, array_ref, array_refs, mut_array_refs};
use num_enum::TryFromPrimitive;
use solana_program::{
    program_error::ProgramError,
    program_option::COption,
    program_pack::{IsInitialized, Pack, Sealed},
    pubkey::Pubkey,
};

/// Mint data.
#[repr(C)]
#[derive(Clone, Copy, Debug, Default, PartialEq)]
pub struct Mint {
    /// Optional authority used to mint new tokens. The mint authority may only be provided during
    /// mint creation. If no mint authority is present then the mint has a fixed supply and no
    /// further tokens may be minted.
    pub mint_authority: COption<Pubkey>,
    /// Total supply of tokens.
    pub supply: u64,
    /// Number of base 10 digits to the right of the decimal place.
    pub decimals: u8,
    /// Is `true` if this structure has been initialized
    pub is_initialized: bool,
    /// Optional authority to freeze token accounts.
    pub freeze_authority: COption<Pubkey>,
}
impl Sealed for Mint {}
impl IsInitialized for Mint {
    fn is_initialized(&self) -> bool {
        self.is_initialized
    }
}
impl Pack for Mint {
    const LEN: usize = 82;
    fn unpack_from_slice(src: &[u8]) -> Result<Self, ProgramError> {
        let src = array_ref![src, 0, 82];
        let (mint_authority, supply, decimals, is_initialized, freeze_authority) =
            array_refs![src, 36, 8, 1, 1, 36];
        let mint_authority = unpack_coption_key(mint_authority)?;
        let supply = u64::from_le_bytes(*supply);
        let decimals = decimals[0];
        let is_initialized = match is_initialized {
            [0] => false,
            [1] => true,
            _ => return Err(ProgramError::InvalidAccountData),
        };
        let freeze_authority = unpack_coption_key(freeze_authority)?;
        Ok(Mint {
            mint_authority,
            supply,
            decimals,
            is_initialized,
            freeze_authority,
        })
    }
    fn pack_into_slice(&self, dst: &mut [u8]) {
        let dst = array_mut_ref![dst, 0, 82];
        let (
            mint_authority_dst,
            supply_dst,
            decimals_dst,
            is_initialized_dst,
            freeze_authority_dst,
        ) = mut_array_refs![dst, 36, 8, 1, 1, 36];
        let &Mint {
            ref mint_authority,
            supply,
            decimals,
            is_initialized,
            ref freeze_authority,
        } = self;
        pack_coption_key(mint_authority, mint_authority_dst);
        *supply_dst = supply.to_le_bytes();
        decimals_dst[0] = decimals;
        is_initialized_dst[0] = is_initialized as u8;
        pack_coption_key(freeze_authority, freeze_authority_dst);
    }
}

/// Account data.
#[repr(C)]
#[derive(Clone, Copy, Debug, Default, PartialEq)]
pub struct Account {
    /// The mint associated with this account
    pub mint: Pubkey,
    /// The owner of this account.
    pub owner: Pubkey,
    /// The amount of tokens this account holds.
    pub amount: u64,
    /// If `delegate` is `Some` then `delegated_amount` represents
    /// the amount authorized by the delegate
    pub delegate: COption<Pubkey>,
    /// The account's state
    pub state: AccountState,
    /// If is_some, this is a native token, and the value logs the rent-exempt reserve. An Account
    /// is required to be rent-exempt, so the value is used by the Processor to ensure that wrapped
    /// SOL accounts do not drop below this threshold.
    pub is_native: COption<u64>,
    /// The amount delegated
    pub delegated_amount: u64,
    /// Optional authority to close the account.
    pub close_authority: COption<Pubkey>,
}
impl Account {
    /// Checks if account is frozen
    pub fn is_frozen(&self) -> bool {
        self.state == AccountState::Frozen
    }
    /// Checks if account is native
    pub fn is_native(&self) -> bool {
        self.is_native.is_some()
    }
}
impl Sealed for Account {}
impl IsInitialized for Account {
    fn is_initialized(&self) -> bool {
        self.state != AccountState::Uninitialized
    }
}
impl Pack for Account {
    const LEN: usize = 165;
    fn unpack_from_slice(src: &[u8]) -> Result<Self, ProgramError> {
        let src = array_ref![src, 0, 165];
        let (mint, owner, amount, delegate, state, is_native, delegated_amount, close_authority) =
            array_refs![src, 32, 32, 8, 36, 1, 12, 8, 36];
        Ok(Account {
            mint: Pubkey::new_from_array(*mint),
            owner: Pubkey::new_from_array(*owner),
            amount: u64::from_le_bytes(*amount),
            delegate: unpack_coption_key(delegate)?,
            state: AccountState::try_from_primitive(state[0])
                .or(Err(ProgramError::InvalidAccountData))?,
            is_native: unpack_coption_u64(is_native)?,
            delegated_amount: u64::from_le_bytes(*delegated_amount),
            close_authority: unpack_coption_key(close_authority)?,
        })
    }
    fn pack_into_slice(&self, dst: &mut [u8]) {
        let dst = array_mut_ref![dst, 0, 165];
        let (
            mint_dst,
            owner_dst,
            amount_dst,
            delegate_dst,
            state_dst,
            is_native_dst,
            delegated_amount_dst,
            close_authority_dst,
        ) = mut_array_refs![dst, 32, 32, 8, 36, 1, 12, 8, 36];
        let &Account {
            ref mint,
            ref owner,
            amount,
            ref delegate,
            state,
            ref is_native,
            delegated_amount,
            ref close_authority,
        } = self;
        mint_dst.copy_from_slice(mint.as_ref());
        owner_dst.copy_from_slice(owner.as_ref());
        *amount_dst = amount.to_le_bytes();
        pack_coption_key(delegate, delegate_dst);
        state_dst[0] = state as u8;
        pack_coption_u64(is_native, is_native_dst);
        *delegated_amount_dst = delegated_amount.to_le_bytes();
        pack_coption_key(close_authority, close_authority_dst);
    }
}

/// Account state.
#[repr(u8)]
#[derive(Clone, Copy, Debug, PartialEq, TryFromPrimitive)]
pub enum AccountState {
    /// Account is not yet initialized
    Uninitialized,
    /// Account is initialized; the account owner and/or delegate may perform permitted operations
    /// on this account
    Initialized,
    /// Account has been frozen by the mint freeze authority. Neither the account owner nor
    /// the delegate are able to perform operations on this account.
    Frozen,
}

impl Default for AccountState {
    fn default() -> Self {
        AccountState::Uninitialized
    }
}

/// Multisignature data.
#[repr(C)]
#[derive(Clone, Copy, Debug, Default, PartialEq)]
pub struct Multisig {
    /// Number of signers required
    pub m: u8,
    /// Number of valid signers
    pub n: u8,
    /// Is `true` if this structure has been initialized
    pub is_initialized: bool,
    /// Signer public keys
    pub signers: [Pubkey; MAX_SIGNERS],
}
impl Sealed for Multisig {}
impl IsInitialized for Multisig {
    fn is_initialized(&self) -> bool {
        self.is_initialized
    }
}
impl Pack for Multisig {
    const LEN: usize = 355;
    fn unpack_from_slice(src: &[u8]) -> Result<Self, ProgramError> {
        let src = array_ref![src, 0, 355];
        #[allow(clippy::ptr_offset_with_cast)]
        let (m, n, is_initialized, signers_flat) = array_refs![src, 1, 1, 1, 32 * MAX_SIGNERS];
        let mut result = Multisig {
            m: m[0],
            n: n[0],
            is_initialized: match is_initialized {
                [0] => false,
                [1] => true,
                _ => return Err(ProgramError::InvalidAccountData),
            },
            signers: [Pubkey::new_from_array([0u8; 32]); MAX_SIGNERS],
        };
        for (src, dst) in signers_flat.chunks(32).zip(result.signers.iter_mut()) {
            *dst = Pubkey::new(src);
        }
        Ok(result)
    }
    fn pack_into_slice(&self, dst: &mut [u8]) {
        let dst = array_mut_ref![dst, 0, 355];
        #[allow(clippy::ptr_offset_with_cast)]
        let (m, n, is_initialized, signers_flat) = mut_array_refs![dst, 1, 1, 1, 32 * MAX_SIGNERS];
        *m = [self.m];
        *n = [self.n];
        *is_initialized = [self.is_initialized as u8];
        for (i, src) in self.signers.iter().enumerate() {
            let dst_array = array_mut_ref![signers_flat, 32 * i, 32];
            dst_array.copy_from_slice(src.as_ref());
        }
    }
}

// Helpers
fn pack_coption_key(src: &COption<Pubkey>, dst: &mut [u8; 36]) {
    let (tag, body) = mut_array_refs![dst, 4, 32];
    match src {
        COption::Some(key) => {
            *tag = [1, 0, 0, 0];
            body.copy_from_slice(key.as_ref());
        }
        COption::None => {
            *tag = [0; 4];
        }
    }
}
fn unpack_coption_key(src: &[u8; 36]) -> Result<COption<Pubkey>, ProgramError> {
    let (tag, body) = array_refs![src, 4, 32];
    match *tag {
        [0, 0, 0, 0] => Ok(COption::None),
        [1, 0, 0, 0] => Ok(COption::Some(Pubkey::new_from_array(*body))),
        _ => Err(ProgramError::InvalidAccountData),
    }
}
fn pack_coption_u64(src: &COption<u64>, dst: &mut [u8; 12]) {
    let (tag, body) = mut_array_refs![dst, 4, 8];
    match src {
        COption::Some(amount) => {
            *tag = [1, 0, 0, 0];
            *body = amount.to_le_bytes();
        }
        COption::None => {
            *tag = [0; 4];
        }
    }
}
fn unpack_coption_u64(src: &[u8; 12]) -> Result<COption<u64>, ProgramError> {
    let (tag, body) = array_refs![src, 4, 8];
    match *tag {
        [0, 0, 0, 0] => Ok(COption::None),
        [1, 0, 0, 0] => Ok(COption::Some(u64::from_le_bytes(*body))),
        _ => Err(ProgramError::InvalidAccountData),
    }
}
```

```
//! The Mint that represents the native token

/// There are 10^9 lamports in one SOL
pub const DECIMALS: u8 = 9;

// The Mint for native SOL Token accounts
solana_program::declare_id!("So11111111111111111111111111111111111111112");

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

    #[test]
    fn test_decimals() {
        assert!(
            (lamports_to_sol(42) - crate::amount_to_ui_amount(42, DECIMALS)).abs() < f64::EPSILON
        );
        assert_eq!(
            sol_to_lamports(42.),
            crate::ui_amount_to_amount(42., DECIMALS)
        );
    }
}
```

```
//! Program state processor

use crate::{
    error::TokenError,
    instruction::{is_valid_signer_index, AuthorityType, TokenInstruction, MAX_SIGNERS},
    state::{Account, AccountState, Mint, Multisig},
};
use num_traits::FromPrimitive;
use solana_program::{
    account_info::{next_account_info, AccountInfo},
    decode_error::DecodeError,
    entrypoint::ProgramResult,
    msg,
    program_error::{PrintProgramError, ProgramError},
    program_option::COption,
    program_pack::{IsInitialized, Pack},
    pubkey::Pubkey,
    sysvar::{rent::Rent, Sysvar},
};

/// Program state handler.
pub struct Processor {}
impl Processor {
    fn _process_initialize_mint(
        accounts: &[AccountInfo],
        decimals: u8,
        mint_authority: Pubkey,
        freeze_authority: COption<Pubkey>,
        rent_sysvar_account: bool,
    ) -> ProgramResult {
        let account_info_iter = &mut accounts.iter();
        let mint_info = next_account_info(account_info_iter)?;
        let mint_data_len = mint_info.data_len();
        let rent = if rent_sysvar_account {
            Rent::from_account_info(next_account_info(account_info_iter)?)?
        } else {
            Rent::get()?
        };

        let mut mint = Mint::unpack_unchecked(&mint_info.data.borrow())?;
        if mint.is_initialized {
            return Err(TokenError::AlreadyInUse.into());
        }

        if !rent.is_exempt(mint_info.lamports(), mint_data_len) {
            return Err(TokenError::NotRentExempt.into());
        }

        mint.mint_authority = COption::Some(mint_authority);
        mint.decimals = decimals;
        mint.is_initialized = true;
        mint.freeze_authority = freeze_authority;

        Mint::pack(mint, &mut mint_info.data.borrow_mut())?;

        Ok(())
    }

    /// Processes an [InitializeMint](enum.TokenInstruction.html) instruction.
    pub fn process_initialize_mint(
        accounts: &[AccountInfo],
        decimals: u8,
        mint_authority: Pubkey,
        freeze_authority: COption<Pubkey>,
    ) -> ProgramResult {
        Self::_process_initialize_mint(accounts, decimals, mint_authority, freeze_authority, true)
    }

    /// Processes an [InitializeMint2](enum.TokenInstruction.html) instruction.
    pub fn process_initialize_mint2(
        accounts: &[AccountInfo],
        decimals: u8,
        mint_authority: Pubkey,
        freeze_authority: COption<Pubkey>,
    ) -> ProgramResult {
        Self::_process_initialize_mint(accounts, decimals, mint_authority, freeze_authority, false)
    }

    fn _process_initialize_account(
        accounts: &[AccountInfo],
        owner: Option<&Pubkey>,
        rent_sysvar_account: bool,
    ) -> ProgramResult {
        let account_info_iter = &mut accounts.iter();
        let new_account_info = next_account_info(account_info_iter)?;
        let mint_info = next_account_info(account_info_iter)?;
        let owner = if let Some(owner) = owner {
            owner
        } else {
            next_account_info(account_info_iter)?.key
        };
        let new_account_info_data_len = new_account_info.data_len();
        let rent = if rent_sysvar_account {
            Rent::from_account_info(next_account_info(account_info_iter)?)?
        } else {
            Rent::get()?
        };

        let mut account = Account::unpack_unchecked(&new_account_info.data.borrow())?;
        if account.is_initialized() {
            return Err(TokenError::AlreadyInUse.into());
        }

        if !rent.is_exempt(new_account_info.lamports(), new_account_info_data_len) {
            return Err(TokenError::NotRentExempt.into());
        }

        if *mint_info.key != crate::native_mint::id() {
            let _ = Mint::unpack(&mint_info.data.borrow_mut())
                .map_err(|_| Into::<ProgramError>::into(TokenError::InvalidMint))?;
        }

        account.mint = *mint_info.key;
        account.owner = *owner;
        account.delegate = COption::None;
        account.delegated_amount = 0;
        account.state = AccountState::Initialized;
        if *mint_info.key == crate::native_mint::id() {
            let rent_exempt_reserve = rent.minimum_balance(new_account_info_data_len);
            account.is_native = COption::Some(rent_exempt_reserve);
            account.amount = new_account_info
                .lamports()
                .checked_sub(rent_exempt_reserve)
                .ok_or(TokenError::Overflow)?;
        } else {
            account.is_native = COption::None;
            account.amount = 0;
        };

        Account::pack(account, &mut new_account_info.data.borrow_mut())?;

        Ok(())
    }

    /// Processes an [InitializeAccount](enum.TokenInstruction.html) instruction.
    pub fn process_initialize_account(accounts: &[AccountInfo]) -> ProgramResult {
        Self::_process_initialize_account(accounts, None, true)
    }

    /// Processes an [InitializeAccount2](enum.TokenInstruction.html) instruction.
    pub fn process_initialize_account2(accounts: &[AccountInfo], owner: Pubkey) -> ProgramResult {
        Self::_process_initialize_account(accounts, Some(&owner), true)
    }

    /// Processes an [InitializeAccount3](enum.TokenInstruction.html) instruction.
    pub fn process_initialize_account3(accounts: &[AccountInfo], owner: Pubkey) -> ProgramResult {
        Self::_process_initialize_account(accounts, Some(&owner), false)
    }

    fn _process_initialize_multisig(
        accounts: &[AccountInfo],
        m: u8,
        rent_sysvar_account: bool,
    ) -> ProgramResult {
        let account_info_iter = &mut accounts.iter();
        let multisig_info = next_account_info(account_info_iter)?;
        let multisig_info_data_len = multisig_info.data_len();
        let rent = if rent_sysvar_account {
            Rent::from_account_info(next_account_info(account_info_iter)?)?
        } else {
            Rent::get()?
        };

        let mut multisig = Multisig::unpack_unchecked(&multisig_info.data.borrow())?;
        if multisig.is_initialized {
            return Err(TokenError::AlreadyInUse.into());
        }

        if !rent.is_exempt(multisig_info.lamports(), multisig_info_data_len) {
            return Err(TokenError::NotRentExempt.into());
        }

        let signer_infos = account_info_iter.as_slice();
        multisig.m = m;
        multisig.n = signer_infos.len() as u8;
        if !is_valid_signer_index(multisig.n as usize) {
            return Err(TokenError::InvalidNumberOfProvidedSigners.into());
        }
        if !is_valid_signer_index(multisig.m as usize) {
            return Err(TokenError::InvalidNumberOfRequiredSigners.into());
        }
        for (i, signer_info) in signer_infos.iter().enumerate() {
            multisig.signers[i] = *signer_info.key;
        }
        multisig.is_initialized = true;

        Multisig::pack(multisig, &mut multisig_info.data.borrow_mut())?;

        Ok(())
    }

    /// Processes a [InitializeMultisig](enum.TokenInstruction.html) instruction.
    pub fn process_initialize_multisig(accounts: &[AccountInfo], m: u8) -> ProgramResult {
        Self::_process_initialize_multisig(accounts, m, true)
    }

    /// Processes a [InitializeMultisig2](enum.TokenInstruction.html) instruction.
    pub fn process_initialize_multisig2(accounts: &[AccountInfo], m: u8) -> ProgramResult {
        Self::_process_initialize_multisig(accounts, m, false)
    }

    /// Processes a [Transfer](enum.TokenInstruction.html) instruction.
    pub fn process_transfer(
        program_id: &Pubkey,
        accounts: &[AccountInfo],
        amount: u64,
        expected_decimals: Option<u8>,
    ) -> ProgramResult {
        let account_info_iter = &mut accounts.iter();

        let source_account_info = next_account_info(account_info_iter)?;

        let expected_mint_info = if let Some(expected_decimals) = expected_decimals {
            Some((next_account_info(account_info_iter)?, expected_decimals))
        } else {
            None
        };

        let dest_account_info = next_account_info(account_info_iter)?;
        let authority_info = next_account_info(account_info_iter)?;

        let mut source_account = Account::unpack(&source_account_info.data.borrow())?;
        let mut dest_account = Account::unpack(&dest_account_info.data.borrow())?;

        if source_account.is_frozen() || dest_account.is_frozen() {
            return Err(TokenError::AccountFrozen.into());
        }
        if source_account.amount < amount {
            return Err(TokenError::InsufficientFunds.into());
        }
        if source_account.mint != dest_account.mint {
            return Err(TokenError::MintMismatch.into());
        }

        if let Some((mint_info, expected_decimals)) = expected_mint_info {
            if source_account.mint != *mint_info.key {
                return Err(TokenError::MintMismatch.into());
            }

            let mint = Mint::unpack(&mint_info.data.borrow_mut())?;
            if expected_decimals != mint.decimals {
                return Err(TokenError::MintDecimalsMismatch.into());
            }
        }

        let self_transfer = source_account_info.key == dest_account_info.key;

        match source_account.delegate {
            COption::Some(ref delegate) if authority_info.key == delegate => {
                Self::validate_owner(
                    program_id,
                    delegate,
                    authority_info,
                    account_info_iter.as_slice(),
                )?;
                if source_account.delegated_amount < amount {
                    return Err(TokenError::InsufficientFunds.into());
                }
                if !self_transfer {
                    source_account.delegated_amount = source_account
                        .delegated_amount
                        .checked_sub(amount)
                        .ok_or(TokenError::Overflow)?;
                    if source_account.delegated_amount == 0 {
                        source_account.delegate = COption::None;
                    }
                }
            }
            _ => Self::validate_owner(
                program_id,
                &source_account.owner,
                authority_info,
                account_info_iter.as_slice(),
            )?,
        };

        // This check MUST occur just before the amounts are manipulated
        // to ensure self-transfers are fully validated
        if self_transfer {
            return Ok(());
        }

        source_account.amount = source_account
            .amount
            .checked_sub(amount)
            .ok_or(TokenError::Overflow)?;
        dest_account.amount = dest_account
            .amount
            .checked_add(amount)
            .ok_or(TokenError::Overflow)?;

        if source_account.is_native() {
            let source_starting_lamports = source_account_info.lamports();
            **source_account_info.lamports.borrow_mut() = source_starting_lamports
                .checked_sub(amount)
                .ok_or(TokenError::Overflow)?;

            let dest_starting_lamports = dest_account_info.lamports();
            **dest_account_info.lamports.borrow_mut() = dest_starting_lamports
                .checked_add(amount)
                .ok_or(TokenError::Overflow)?;
        }

        Account::pack(source_account, &mut source_account_info.data.borrow_mut())?;
        Account::pack(dest_account, &mut dest_account_info.data.borrow_mut())?;

        Ok(())
    }

    /// Processes an [Approve](enum.TokenInstruction.html) instruction.
    pub fn process_approve(
        program_id: &Pubkey,
        accounts: &[AccountInfo],
        amount: u64,
        expected_decimals: Option<u8>,
    ) -> ProgramResult {
        let account_info_iter = &mut accounts.iter();

        let source_account_info = next_account_info(account_info_iter)?;

        let expected_mint_info = if let Some(expected_decimals) = expected_decimals {
            Some((next_account_info(account_info_iter)?, expected_decimals))
        } else {
            None
        };
        let delegate_info = next_account_info(account_info_iter)?;
        let owner_info = next_account_info(account_info_iter)?;

        let mut source_account = Account::unpack(&source_account_info.data.borrow())?;

        if source_account.is_frozen() {
            return Err(TokenError::AccountFrozen.into());
        }

        if let Some((mint_info, expected_decimals)) = expected_mint_info {
            if source_account.mint != *mint_info.key {
                return Err(TokenError::MintMismatch.into());
            }

            let mint = Mint::unpack(&mint_info.data.borrow_mut())?;
            if expected_decimals != mint.decimals {
                return Err(TokenError::MintDecimalsMismatch.into());
            }
        }
        Self::validate_owner(
            program_id,
            &source_account.owner,
            owner_info,
            account_info_iter.as_slice(),
        )?;

        source_account.delegate = COption::Some(*delegate_info.key);
        source_account.delegated_amount = amount;

        Account::pack(source_account, &mut source_account_info.data.borrow_mut())?;

        Ok(())
    }

    /// Processes an [Revoke](enum.TokenInstruction.html) instruction.
    pub fn process_revoke(program_id: &Pubkey, accounts: &[AccountInfo]) -> ProgramResult {
        let account_info_iter = &mut accounts.iter();
        let source_account_info = next_account_info(account_info_iter)?;

        let mut source_account = Account::unpack(&source_account_info.data.borrow())?;

        let owner_info = next_account_info(account_info_iter)?;

        if source_account.is_frozen() {
            return Err(TokenError::AccountFrozen.into());
        }

        Self::validate_owner(
            program_id,
            &source_account.owner,
            owner_info,
            account_info_iter.as_slice(),
        )?;

        source_account.delegate = COption::None;
        source_account.delegated_amount = 0;

        Account::pack(source_account, &mut source_account_info.data.borrow_mut())?;

        Ok(())
    }

    /// Processes a [SetAuthority](enum.TokenInstruction.html) instruction.
    pub fn process_set_authority(
        program_id: &Pubkey,
        accounts: &[AccountInfo],
        authority_type: AuthorityType,
        new_authority: COption<Pubkey>,
    ) -> ProgramResult {
        let account_info_iter = &mut accounts.iter();
        let account_info = next_account_info(account_info_iter)?;
        let authority_info = next_account_info(account_info_iter)?;

        if account_info.data_len() == Account::get_packed_len() {
            let mut account = Account::unpack(&account_info.data.borrow())?;

            if account.is_frozen() {
                return Err(TokenError::AccountFrozen.into());
            }

            match authority_type {
                AuthorityType::AccountOwner => {
                    Self::validate_owner(
                        program_id,
                        &account.owner,
                        authority_info,
                        account_info_iter.as_slice(),
                    )?;

                    if let COption::Some(authority) = new_authority {
                        account.owner = authority;
                    } else {
                        return Err(TokenError::InvalidInstruction.into());
                    }

                    account.delegate = COption::None;
                    account.delegated_amount = 0;

                    if account.is_native() {
                        account.close_authority = COption::None;
                    }
                }
                AuthorityType::CloseAccount => {
                    let authority = account.close_authority.unwrap_or(account.owner);
                    Self::validate_owner(
                        program_id,
                        &authority,
                        authority_info,
                        account_info_iter.as_slice(),
                    )?;
                    account.close_authority = new_authority;
                }
                _ => {
                    return Err(TokenError::AuthorityTypeNotSupported.into());
                }
            }
            Account::pack(account, &mut account_info.data.borrow_mut())?;
        } else if account_info.data_len() == Mint::get_packed_len() {
            let mut mint = Mint::unpack(&account_info.data.borrow())?;
            match authority_type {
                AuthorityType::MintTokens => {
                    // Once a mint's supply is fixed, it cannot be undone by setting a new
                    // mint_authority
                    let mint_authority = mint
                        .mint_authority
                        .ok_or(Into::<ProgramError>::into(TokenError::FixedSupply))?;
                    Self::validate_owner(
                        program_id,
                        &mint_authority,
                        authority_info,
                        account_info_iter.as_slice(),
                    )?;
                    mint.mint_authority = new_authority;
                }
                AuthorityType::FreezeAccount => {
                    // Once a mint's freeze authority is disabled, it cannot be re-enabled by
                    // setting a new freeze_authority
                    let freeze_authority = mint
                        .freeze_authority
                        .ok_or(Into::<ProgramError>::into(TokenError::MintCannotFreeze))?;
                    Self::validate_owner(
                        program_id,
                        &freeze_authority,
                        authority_info,
                        account_info_iter.as_slice(),
                    )?;
                    mint.freeze_authority = new_authority;
                }
                _ => {
                    return Err(TokenError::AuthorityTypeNotSupported.into());
                }
            }
            Mint::pack(mint, &mut account_info.data.borrow_mut())?;
        } else {
            return Err(ProgramError::InvalidArgument);
        }

        Ok(())
    }

    /// Processes a [MintTo](enum.TokenInstruction.html) instruction.
    pub fn process_mint_to(
        program_id: &Pubkey,
        accounts: &[AccountInfo],
        amount: u64,
        expected_decimals: Option<u8>,
    ) -> ProgramResult {
        let account_info_iter = &mut accounts.iter();
        let mint_info = next_account_info(account_info_iter)?;
        let dest_account_info = next_account_info(account_info_iter)?;
        let owner_info = next_account_info(account_info_iter)?;

        let mut dest_account = Account::unpack(&dest_account_info.data.borrow())?;
        if dest_account.is_frozen() {
            return Err(TokenError::AccountFrozen.into());
        }

        if dest_account.is_native() {
            return Err(TokenError::NativeNotSupported.into());
        }
        if mint_info.key != &dest_account.mint {
            return Err(TokenError::MintMismatch.into());
        }

        let mut mint = Mint::unpack(&mint_info.data.borrow())?;
        if let Some(expected_decimals) = expected_decimals {
            if expected_decimals != mint.decimals {
                return Err(TokenError::MintDecimalsMismatch.into());
            }
        }

        match mint.mint_authority {
            COption::Some(mint_authority) => Self::validate_owner(
                program_id,
                &mint_authority,
                owner_info,
                account_info_iter.as_slice(),
            )?,
            COption::None => return Err(TokenError::FixedSupply.into()),
        }

        dest_account.amount = dest_account
            .amount
            .checked_add(amount)
            .ok_or(TokenError::Overflow)?;

        mint.supply = mint
            .supply
            .checked_add(amount)
            .ok_or(TokenError::Overflow)?;

        Account::pack(dest_account, &mut dest_account_info.data.borrow_mut())?;
        Mint::pack(mint, &mut mint_info.data.borrow_mut())?;

        Ok(())
    }
    /// Processes a [Burn](enum.TokenInstruction.html) instruction.
    pub fn process_burn(
        program_id: &Pubkey,
        accounts: &[AccountInfo],
        amount: u64,
        expected_decimals: Option<u8>,
    ) -> ProgramResult {
        let account_info_iter = &mut accounts.iter();

        let source_account_info = next_account_info(account_info_iter)?;
        let mint_info = next_account_info(account_info_iter)?;
        let authority_info = next_account_info(account_info_iter)?;

        let mut source_account = Account::unpack(&source_account_info.data.borrow())?;
        let mut mint = Mint::unpack(&mint_info.data.borrow())?;

        if source_account.is_frozen() {
            return Err(TokenError::AccountFrozen.into());
        }
        if source_account.is_native() {
            return Err(TokenError::NativeNotSupported.into());
        }
        if source_account.amount < amount {
            return Err(TokenError::InsufficientFunds.into());
        }
        if mint_info.key != &source_account.mint {
            return Err(TokenError::MintMismatch.into());
        }

        if let Some(expected_decimals) = expected_decimals {
            if expected_decimals != mint.decimals {
                return Err(TokenError::MintDecimalsMismatch.into());
            }
        }

        match source_account.delegate {
            COption::Some(ref delegate) if authority_info.key == delegate => {
                Self::validate_owner(
                    program_id,
                    delegate,
                    authority_info,
                    account_info_iter.as_slice(),
                )?;

                if source_account.delegated_amount < amount {
                    return Err(TokenError::InsufficientFunds.into());
                }
                source_account.delegated_amount = source_account
                    .delegated_amount
                    .checked_sub(amount)
                    .ok_or(TokenError::Overflow)?;
                if source_account.delegated_amount == 0 {
                    source_account.delegate = COption::None;
                }
            }
            _ => Self::validate_owner(
                program_id,
                &source_account.owner,
                authority_info,
                account_info_iter.as_slice(),
            )?,
        }

        source_account.amount = source_account
            .amount
            .checked_sub(amount)
            .ok_or(TokenError::Overflow)?;
        mint.supply = mint
            .supply
            .checked_sub(amount)
            .ok_or(TokenError::Overflow)?;

        Account::pack(source_account, &mut source_account_info.data.borrow_mut())?;
        Mint::pack(mint, &mut mint_info.data.borrow_mut())?;

        Ok(())
    }

    /// Processes a [CloseAccount](enum.TokenInstruction.html) instruction.
    pub fn process_close_account(program_id: &Pubkey, accounts: &[AccountInfo]) -> ProgramResult {
        let account_info_iter = &mut accounts.iter();
        let source_account_info = next_account_info(account_info_iter)?;
        let dest_account_info = next_account_info(account_info_iter)?;
        let authority_info = next_account_info(account_info_iter)?;

        let mut source_account = Account::unpack(&source_account_info.data.borrow())?;
        if !source_account.is_native() && source_account.amount != 0 {
            return Err(TokenError::NonNativeHasBalance.into());
        }

        let authority = source_account
            .close_authority
            .unwrap_or(source_account.owner);
        Self::validate_owner(
            program_id,
            &authority,
            authority_info,
            account_info_iter.as_slice(),
        )?;

        let dest_starting_lamports = dest_account_info.lamports();
        **dest_account_info.lamports.borrow_mut() = dest_starting_lamports
            .checked_add(source_account_info.lamports())
            .ok_or(TokenError::Overflow)?;

        **source_account_info.lamports.borrow_mut() = 0;
        source_account.amount = 0;

        Account::pack(source_account, &mut source_account_info.data.borrow_mut())?;

        Ok(())
    }

    /// Processes a [FreezeAccount](enum.TokenInstruction.html) or a
    /// [ThawAccount](enum.TokenInstruction.html) instruction.
    pub fn process_toggle_freeze_account(
        program_id: &Pubkey,
        accounts: &[AccountInfo],
        freeze: bool,
    ) -> ProgramResult {
        let account_info_iter = &mut accounts.iter();
        let source_account_info = next_account_info(account_info_iter)?;
        let mint_info = next_account_info(account_info_iter)?;
        let authority_info = next_account_info(account_info_iter)?;

        let mut source_account = Account::unpack(&source_account_info.data.borrow())?;
        if freeze && source_account.is_frozen() || !freeze && !source_account.is_frozen() {
            return Err(TokenError::InvalidState.into());
        }
        if source_account.is_native() {
            return Err(TokenError::NativeNotSupported.into());
        }
        if mint_info.key != &source_account.mint {
            return Err(TokenError::MintMismatch.into());
        }

        let mint = Mint::unpack(&mint_info.data.borrow_mut())?;
        match mint.freeze_authority {
            COption::Some(authority) => Self::validate_owner(
                program_id,
                &authority,
                authority_info,
                account_info_iter.as_slice(),
            ),
            COption::None => Err(TokenError::MintCannotFreeze.into()),
        }?;

        source_account.state = if freeze {
            AccountState::Frozen
        } else {
            AccountState::Initialized
        };

        Account::pack(source_account, &mut source_account_info.data.borrow_mut())?;

        Ok(())
    }

    /// Processes a [SyncNative](enum.TokenInstruction.html) instruction
    pub fn process_sync_native(program_id: &Pubkey, accounts: &[AccountInfo]) -> ProgramResult {
        let account_info_iter = &mut accounts.iter();
        let native_account_info = next_account_info(account_info_iter)?;

        if native_account_info.owner != program_id {
            return Err(ProgramError::IncorrectProgramId);
        }
        let mut native_account = Account::unpack(&native_account_info.data.borrow())?;

        if let COption::Some(rent_exempt_reserve) = native_account.is_native {
            let new_amount = native_account_info
                .lamports()
                .checked_sub(rent_exempt_reserve)
                .ok_or(TokenError::Overflow)?;
            if new_amount < native_account.amount {
                return Err(TokenError::InvalidState.into());
            }
            native_account.amount = new_amount;
        } else {
            return Err(TokenError::NonNativeNotSupported.into());
        }

        Account::pack(native_account, &mut native_account_info.data.borrow_mut())?;
        Ok(())
    }

    /// Processes an [Instruction](enum.Instruction.html).
    pub fn process(program_id: &Pubkey, accounts: &[AccountInfo], input: &[u8]) -> ProgramResult {
        let instruction = TokenInstruction::unpack(input)?;

        match instruction {
            TokenInstruction::InitializeMint {
                decimals,
                mint_authority,
                freeze_authority,
            } => {
                msg!("Instruction: InitializeMint");
                Self::process_initialize_mint(accounts, decimals, mint_authority, freeze_authority)
            }
            TokenInstruction::InitializeMint2 {
                decimals,
                mint_authority,
                freeze_authority,
            } => {
                msg!("Instruction: InitializeMint2");
                Self::process_initialize_mint2(accounts, decimals, mint_authority, freeze_authority)
            }
            TokenInstruction::InitializeAccount => {
                msg!("Instruction: InitializeAccount");
                Self::process_initialize_account(accounts)
            }
            TokenInstruction::InitializeAccount2 { owner } => {
                msg!("Instruction: InitializeAccount2");
                Self::process_initialize_account2(accounts, owner)
            }
            TokenInstruction::InitializeAccount3 { owner } => {
                msg!("Instruction: InitializeAccount3");
                Self::process_initialize_account3(accounts, owner)
            }
            TokenInstruction::InitializeMultisig { m } => {
                msg!("Instruction: InitializeMultisig");
                Self::process_initialize_multisig(accounts, m)
            }
            TokenInstruction::InitializeMultisig2 { m } => {
                msg!("Instruction: InitializeMultisig2");
                Self::process_initialize_multisig2(accounts, m)
            }
            TokenInstruction::Transfer { amount } => {
                msg!("Instruction: Transfer");
                Self::process_transfer(program_id, accounts, amount, None)
            }
            TokenInstruction::Approve { amount } => {
                msg!("Instruction: Approve");
                Self::process_approve(program_id, accounts, amount, None)
            }
            TokenInstruction::Revoke => {
                msg!("Instruction: Revoke");
                Self::process_revoke(program_id, accounts)
            }
            TokenInstruction::SetAuthority {
                authority_type,
                new_authority,
            } => {
                msg!("Instruction: SetAuthority");
                Self::process_set_authority(program_id, accounts, authority_type, new_authority)
            }
            TokenInstruction::MintTo { amount } => {
                msg!("Instruction: MintTo");
                Self::process_mint_to(program_id, accounts, amount, None)
            }
            TokenInstruction::Burn { amount } => {
                msg!("Instruction: Burn");
                Self::process_burn(program_id, accounts, amount, None)
            }
            TokenInstruction::CloseAccount => {
                msg!("Instruction: CloseAccount");
                Self::process_close_account(program_id, accounts)
            }
            TokenInstruction::FreezeAccount => {
                msg!("Instruction: FreezeAccount");
                Self::process_toggle_freeze_account(program_id, accounts, true)
            }
            TokenInstruction::ThawAccount => {
                msg!("Instruction: ThawAccount");
                Self::process_toggle_freeze_account(program_id, accounts, false)
            }
            TokenInstruction::TransferChecked { amount, decimals } => {
                msg!("Instruction: TransferChecked");
                Self::process_transfer(program_id, accounts, amount, Some(decimals))
            }
            TokenInstruction::ApproveChecked { amount, decimals } => {
                msg!("Instruction: ApproveChecked");
                Self::process_approve(program_id, accounts, amount, Some(decimals))
            }
            TokenInstruction::MintToChecked { amount, decimals } => {
                msg!("Instruction: MintToChecked");
                Self::process_mint_to(program_id, accounts, amount, Some(decimals))
            }
            TokenInstruction::BurnChecked { amount, decimals } => {
                msg!("Instruction: BurnChecked");
                Self::process_burn(program_id, accounts, amount, Some(decimals))
            }
            TokenInstruction::SyncNative => {
                msg!("Instruction: SyncNative");
                Self::process_sync_native(program_id, accounts)
            }
        }
    }

    /// Validates owner(s) are present
    pub fn validate_owner(
        program_id: &Pubkey,
        expected_owner: &Pubkey,
        owner_account_info: &AccountInfo,
        signers: &[AccountInfo],
    ) -> ProgramResult {
        if expected_owner != owner_account_info.key {
            return Err(TokenError::OwnerMismatch.into());
        }
        if program_id == owner_account_info.owner
            && owner_account_info.data_len() == Multisig::get_packed_len()
        {
            let multisig = Multisig::unpack(&owner_account_info.data.borrow())?;
            let mut num_signers = 0;
            let mut matched = [false; MAX_SIGNERS];
            for signer in signers.iter() {
                for (position, key) in multisig.signers[0..multisig.n as usize].iter().enumerate() {
                    if key == signer.key && !matched[position] {
                        if !signer.is_signer {
                            return Err(ProgramError::MissingRequiredSignature);
                        }
                        matched[position] = true;
                        num_signers += 1;
                    }
                }
            }
            if num_signers < multisig.m {
                return Err(ProgramError::MissingRequiredSignature);
            }
            return Ok(());
        } else if !owner_account_info.is_signer {
            return Err(ProgramError::MissingRequiredSignature);
        }
        Ok(())
    }
}

impl PrintProgramError for TokenError {
    fn print<E>(&self)
    where
        E: 'static + std::error::Error + DecodeError<E> + PrintProgramError + FromPrimitive,
    {
        match self {
            TokenError::NotRentExempt => msg!("Error: Lamport balance below rent-exempt threshold"),
            TokenError::InsufficientFunds => msg!("Error: insufficient funds"),
            TokenError::InvalidMint => msg!("Error: Invalid Mint"),
            TokenError::MintMismatch => msg!("Error: Account not associated with this Mint"),
            TokenError::OwnerMismatch => msg!("Error: owner does not match"),
            TokenError::FixedSupply => msg!("Error: the total supply of this token is fixed"),
            TokenError::AlreadyInUse => msg!("Error: account or token already in use"),
            TokenError::InvalidNumberOfProvidedSigners => {
                msg!("Error: Invalid number of provided signers")
            }
            TokenError::InvalidNumberOfRequiredSigners => {
                msg!("Error: Invalid number of required signers")
            }
            TokenError::UninitializedState => msg!("Error: State is uninitialized"),
            TokenError::NativeNotSupported => {
                msg!("Error: Instruction does not support native tokens")
            }
            TokenError::NonNativeHasBalance => {
                msg!("Error: Non-native account can only be closed if its balance is zero")
            }
            TokenError::InvalidInstruction => msg!("Error: Invalid instruction"),
            TokenError::InvalidState => msg!("Error: Invalid account state for operation"),
            TokenError::Overflow => msg!("Error: Operation overflowed"),
            TokenError::AuthorityTypeNotSupported => {
                msg!("Error: Account does not support specified authority type")
            }
            TokenError::MintCannotFreeze => msg!("Error: This token mint cannot freeze accounts"),
            TokenError::AccountFrozen => msg!("Error: Account is frozen"),
            TokenError::MintDecimalsMismatch => {
                msg!("Error: decimals different from the Mint decimals")
            }
            TokenError::NonNativeNotSupported => {
                msg!("Error: Instruction does not support non-native tokens")
            }
        }
    }
}
```

```
//! Instruction types

use crate::{check_program_account, error::TokenError};
use solana_program::{
    instruction::{AccountMeta, Instruction},
    program_error::ProgramError,
    program_option::COption,
    pubkey::Pubkey,
    sysvar,
};
use std::convert::TryInto;
use std::mem::size_of;

/// Minimum number of multisignature signers (min N)
pub const MIN_SIGNERS: usize = 1;
/// Maximum number of multisignature signers (max N)
pub const MAX_SIGNERS: usize = 11;

/// Instructions supported by the token program.
#[repr(C)]
#[derive(Clone, Debug, PartialEq)]
pub enum TokenInstruction {
    /// Initializes a new mint and optionally deposits all the newly minted
    /// tokens in an account.
    ///
    /// The `InitializeMint` instruction requires no signers and MUST be
    /// included within the same Transaction as the system program's
    /// `CreateAccount` instruction that creates the account being initialized.
    /// Otherwise another party can acquire ownership of the uninitialized
    /// account.
    ///
    /// Accounts expected by this instruction:
    ///
    ///   0. `[writable]` The mint to initialize.
    ///   1. `[]` Rent sysvar
    ///
    InitializeMint {
        /// Number of base 10 digits to the right of the decimal place.
        decimals: u8,
        /// The authority/multisignature to mint tokens.
        mint_authority: Pubkey,
        /// The freeze authority/multisignature of the mint.
        freeze_authority: COption<Pubkey>,
    },
    /// Initializes a new account to hold tokens.  If this account is associated
    /// with the native mint then the token balance of the initialized account
    /// will be equal to the amount of SOL in the account. If this account is
    /// associated with another mint, that mint must be initialized before this
    /// command can succeed.
    ///
    /// The `InitializeAccount` instruction requires no signers and MUST be
    /// included within the same Transaction as the system program's
    /// `CreateAccount` instruction that creates the account being initialized.
    /// Otherwise another party can acquire ownership of the uninitialized
    /// account.
    ///
    /// Accounts expected by this instruction:
    ///
    ///   0. `[writable]`  The account to initialize.
    ///   1. `[]` The mint this account will be associated with.
    ///   2. `[]` The new account's owner/multisignature.
    ///   3. `[]` Rent sysvar
    InitializeAccount,
    /// Initializes a multisignature account with N provided signers.
    ///
    /// Multisignature accounts can used in place of any single owner/delegate
    /// accounts in any token instruction that require an owner/delegate to be
    /// present.  The variant field represents the number of signers (M)
    /// required to validate this multisignature account.
    ///
    /// The `InitializeMultisig` instruction requires no signers and MUST be
    /// included within the same Transaction as the system program's
    /// `CreateAccount` instruction that creates the account being initialized.
    /// Otherwise another party can acquire ownership of the uninitialized
    /// account.
    ///
    /// Accounts expected by this instruction:
    ///
    ///   0. `[writable]` The multisignature account to initialize.
    ///   1. `[]` Rent sysvar
    ///   2. ..2+N. `[]` The signer accounts, must equal to N where 1 <= N <=
    ///      11.
    InitializeMultisig {
        /// The number of signers (M) required to validate this multisignature
        /// account.
        m: u8,
    },
    /// Transfers tokens from one account to another either directly or via a
    /// delegate.  If this account is associated with the native mint then equal
    /// amounts of SOL and Tokens will be transferred to the destination
    /// account.
    ///
    /// Accounts expected by this instruction:
    ///
    ///   * Single owner/delegate
    ///   0. `[writable]` The source account.
    ///   1. `[writable]` The destination account.
    ///   2. `[signer]` The source account's owner/delegate.
    ///
    ///   * Multisignature owner/delegate
    ///   0. `[writable]` The source account.
    ///   1. `[writable]` The destination account.
    ///   2. `[]` The source account's multisignature owner/delegate.
    ///   3. ..3+M `[signer]` M signer accounts.
    Transfer {
        /// The amount of tokens to transfer.
        amount: u64,
    },
    /// Approves a delegate.  A delegate is given the authority over tokens on
    /// behalf of the source account's owner.
    ///
    /// Accounts expected by this instruction:
    ///
    ///   * Single owner
    ///   0. `[writable]` The source account.
    ///   1. `[]` The delegate.
    ///   2. `[signer]` The source account owner.
    ///
    ///   * Multisignature owner
    ///   0. `[writable]` The source account.
    ///   1. `[]` The delegate.
    ///   2. `[]` The source account's multisignature owner.
    ///   3. ..3+M `[signer]` M signer accounts
    Approve {
        /// The amount of tokens the delegate is approved for.
        amount: u64,
    },
    /// Revokes the delegate's authority.
    ///
    /// Accounts expected by this instruction:
    ///
    ///   * Single owner
    ///   0. `[writable]` The source account.
    ///   1. `[signer]` The source account owner.
    ///
    ///   * Multisignature owner
    ///   0. `[writable]` The source account.
    ///   1. `[]` The source account's multisignature owner.
    ///   2. ..2+M `[signer]` M signer accounts
    Revoke,
    /// Sets a new authority of a mint or account.
    ///
    /// Accounts expected by this instruction:
    ///
    ///   * Single authority
    ///   0. `[writable]` The mint or account to change the authority of.
    ///   1. `[signer]` The current authority of the mint or account.
    ///
    ///   * Multisignature authority
    ///   0. `[writable]` The mint or account to change the authority of.
    ///   1. `[]` The mint's or account's current multisignature authority.
    ///   2. ..2+M `[signer]` M signer accounts
    SetAuthority {
        /// The type of authority to update.
        authority_type: AuthorityType,
        /// The new authority
        new_authority: COption<Pubkey>,
    },
    /// Mints new tokens to an account.  The native mint does not support
    /// minting.
    ///
    /// Accounts expected by this instruction:
    ///
    ///   * Single authority
    ///   0. `[writable]` The mint.
    ///   1. `[writable]` The account to mint tokens to.
    ///   2. `[signer]` The mint's minting authority.
    ///
    ///   * Multisignature authority
    ///   0. `[writable]` The mint.
    ///   1. `[writable]` The account to mint tokens to.
    ///   2. `[]` The mint's multisignature mint-tokens authority.
    ///   3. ..3+M `[signer]` M signer accounts.
    MintTo {
        /// The amount of new tokens to mint.
        amount: u64,
    },
    /// Burns tokens by removing them from an account.  `Burn` does not support
    /// accounts associated with the native mint, use `CloseAccount` instead.
    ///
    /// Accounts expected by this instruction:
    ///
    ///   * Single owner/delegate
    ///   0. `[writable]` The account to burn from.
    ///   1. `[writable]` The token mint.
    ///   2. `[signer]` The account's owner/delegate.
    ///
    ///   * Multisignature owner/delegate
    ///   0. `[writable]` The account to burn from.
    ///   1. `[writable]` The token mint.
    ///   2. `[]` The account's multisignature owner/delegate.
    ///   3. ..3+M `[signer]` M signer accounts.
    Burn {
        /// The amount of tokens to burn.
        amount: u64,
    },
    /// Close an account by transferring all its SOL to the destination account.
    /// Non-native accounts may only be closed if its token amount is zero.
    ///
    /// Accounts expected by this instruction:
    ///
    ///   * Single owner
    ///   0. `[writable]` The account to close.
    ///   1. `[writable]` The destination account.
    ///   2. `[signer]` The account's owner.
    ///
    ///   * Multisignature owner
    ///   0. `[writable]` The account to close.
    ///   1. `[writable]` The destination account.
    ///   2. `[]` The account's multisignature owner.
    ///   3. ..3+M `[signer]` M signer accounts.
    CloseAccount,
    /// Freeze an Initialized account using the Mint's freeze_authority (if
    /// set).
    ///
    /// Accounts expected by this instruction:
    ///
    ///   * Single owner
    ///   0. `[writable]` The account to freeze.
    ///   1. `[]` The token mint.
    ///   2. `[signer]` The mint freeze authority.
    ///
    ///   * Multisignature owner
    ///   0. `[writable]` The account to freeze.
    ///   1. `[]` The token mint.
    ///   2. `[]` The mint's multisignature freeze authority.
    ///   3. ..3+M `[signer]` M signer accounts.
    FreezeAccount,
    /// Thaw a Frozen account using the Mint's freeze_authority (if set).
    ///
    /// Accounts expected by this instruction:
    ///
    ///   * Single owner
    ///   0. `[writable]` The account to freeze.
    ///   1. `[]` The token mint.
    ///   2. `[signer]` The mint freeze authority.
    ///
    ///   * Multisignature owner
    ///   0. `[writable]` The account to freeze.
    ///   1. `[]` The token mint.
    ///   2. `[]` The mint's multisignature freeze authority.
    ///   3. ..3+M `[signer]` M signer accounts.
    ThawAccount,

    /// Transfers tokens from one account to another either directly or via a
    /// delegate.  If this account is associated with the native mint then equal
    /// amounts of SOL and Tokens will be transferred to the destination
    /// account.
    ///
    /// This instruction differs from Transfer in that the token mint and
    /// decimals value is checked by the caller.  This may be useful when
    /// creating transactions offline or within a hardware wallet.
    ///
    /// Accounts expected by this instruction:
    ///
    ///   * Single owner/delegate
    ///   0. `[writable]` The source account.
    ///   1. `[]` The token mint.
    ///   2. `[writable]` The destination account.
    ///   3. `[signer]` The source account's owner/delegate.
    ///
    ///   * Multisignature owner/delegate
    ///   0. `[writable]` The source account.
    ///   1. `[]` The token mint.
    ///   2. `[writable]` The destination account.
    ///   3. `[]` The source account's multisignature owner/delegate.
    ///   4. ..4+M `[signer]` M signer accounts.
    TransferChecked {
        /// The amount of tokens to transfer.
        amount: u64,
        /// Expected number of base 10 digits to the right of the decimal place.
        decimals: u8,
    },
    /// Approves a delegate.  A delegate is given the authority over tokens on
    /// behalf of the source account's owner.
    ///
    /// This instruction differs from Approve in that the token mint and
    /// decimals value is checked by the caller.  This may be useful when
    /// creating transactions offline or within a hardware wallet.
    ///
    /// Accounts expected by this instruction:
    ///
    ///   * Single owner
    ///   0. `[writable]` The source account.
    ///   1. `[]` The token mint.
    ///   2. `[]` The delegate.
    ///   3. `[signer]` The source account owner.
    ///
    ///   * Multisignature owner
    ///   0. `[writable]` The source account.
    ///   1. `[]` The token mint.
    ///   2. `[]` The delegate.
    ///   3. `[]` The source account's multisignature owner.
    ///   4. ..4+M `[signer]` M signer accounts
    ApproveChecked {
        /// The amount of tokens the delegate is approved for.
        amount: u64,
        /// Expected number of base 10 digits to the right of the decimal place.
        decimals: u8,
    },
    /// Mints new tokens to an account.  The native mint does not support
    /// minting.
    ///
    /// This instruction differs from MintTo in that the decimals value is
    /// checked by the caller.  This may be useful when creating transactions
    /// offline or within a hardware wallet.
    ///
    /// Accounts expected by this instruction:
    ///
    ///   * Single authority
    ///   0. `[writable]` The mint.
    ///   1. `[writable]` The account to mint tokens to.
    ///   2. `[signer]` The mint's minting authority.
    ///
    ///   * Multisignature authority
    ///   0. `[writable]` The mint.
    ///   1. `[writable]` The account to mint tokens to.
    ///   2. `[]` The mint's multisignature mint-tokens authority.
    ///   3. ..3+M `[signer]` M signer accounts.
    MintToChecked {
        /// The amount of new tokens to mint.
        amount: u64,
        /// Expected number of base 10 digits to the right of the decimal place.
        decimals: u8,
    },
    /// Burns tokens by removing them from an account.  `BurnChecked` does not
    /// support accounts associated with the native mint, use `CloseAccount`
    /// instead.
    ///
    /// This instruction differs from Burn in that the decimals value is checked
    /// by the caller. This may be useful when creating transactions offline or
    /// within a hardware wallet.
    ///
    /// Accounts expected by this instruction:
    ///
    ///   * Single owner/delegate
    ///   0. `[writable]` The account to burn from.
    ///   1. `[writable]` The token mint.
    ///   2. `[signer]` The account's owner/delegate.
    ///
    ///   * Multisignature owner/delegate
    ///   0. `[writable]` The account to burn from.
    ///   1. `[writable]` The token mint.
    ///   2. `[]` The account's multisignature owner/delegate.
    ///   3. ..3+M `[signer]` M signer accounts.
    BurnChecked {
        /// The amount of tokens to burn.
        amount: u64,
        /// Expected number of base 10 digits to the right of the decimal place.
        decimals: u8,
    },
    /// Like InitializeAccount, but the owner pubkey is passed via instruction data
    /// rather than the accounts list. This variant may be preferable when using
    /// Cross Program Invocation from an instruction that does not need the owner's
    /// `AccountInfo` otherwise.
    ///
    /// Accounts expected by this instruction:
    ///
    ///   0. `[writable]`  The account to initialize.
    ///   1. `[]` The mint this account will be associated with.
    ///   3. `[]` Rent sysvar
    InitializeAccount2 {
        /// The new account's owner/multisignature.
        owner: Pubkey,
    },
    /// Given a wrapped / native token account (a token account containing SOL)
    /// updates its amount field based on the account's underlying `lamports`.
    /// This is useful if a non-wrapped SOL account uses `system_instruction::transfer`
    /// to move lamports to a wrapped token account, and needs to have its token
    /// `amount` field updated.
    ///
    /// Accounts expected by this instruction:
    ///
    ///   0. `[writable]`  The native token account to sync with its underlying lamports.
    SyncNative,
    /// Like InitializeAccount2, but does not require the Rent sysvar to be provided
    ///
    /// Accounts expected by this instruction:
    ///
    ///   0. `[writable]`  The account to initialize.
    ///   1. `[]` The mint this account will be associated with.
    InitializeAccount3 {
        /// The new account's owner/multisignature.
        owner: Pubkey,
    },
    /// Like InitializeMultisig, but does not require the Rent sysvar to be provided
    ///
    /// Accounts expected by this instruction:
    ///
    ///   0. `[writable]` The multisignature account to initialize.
    ///   1. ..1+N. `[]` The signer accounts, must equal to N where 1 <= N <=
    ///      11.
    InitializeMultisig2 {
        /// The number of signers (M) required to validate this multisignature
        /// account.
        m: u8,
    },
    /// Like InitializeMint, but does not require the Rent sysvar to be provided
    ///
    /// Accounts expected by this instruction:
    ///
    ///   0. `[writable]` The mint to initialize.
    ///
    InitializeMint2 {
        /// Number of base 10 digits to the right of the decimal place.
        decimals: u8,
        /// The authority/multisignature to mint tokens.
        mint_authority: Pubkey,
        /// The freeze authority/multisignature of the mint.
        freeze_authority: COption<Pubkey>,
    },
}
impl TokenInstruction {
    /// Unpacks a byte buffer into a [TokenInstruction](enum.TokenInstruction.html).
    pub fn unpack(input: &[u8]) -> Result<Self, ProgramError> {
        use TokenError::InvalidInstruction;

        let (&tag, rest) = input.split_first().ok_or(InvalidInstruction)?;
        Ok(match tag {
            0 => {
                let (&decimals, rest) = rest.split_first().ok_or(InvalidInstruction)?;
                let (mint_authority, rest) = Self::unpack_pubkey(rest)?;
                let (freeze_authority, _rest) = Self::unpack_pubkey_option(rest)?;
                Self::InitializeMint {
                    mint_authority,
                    freeze_authority,
                    decimals,
                }
            }
            1 => Self::InitializeAccount,
            2 => {
                let &m = rest.get(0).ok_or(InvalidInstruction)?;
                Self::InitializeMultisig { m }
            }
            3 | 4 | 7 | 8 => {
                let amount = rest
                    .get(..8)
                    .and_then(|slice| slice.try_into().ok())
                    .map(u64::from_le_bytes)
                    .ok_or(InvalidInstruction)?;
                match tag {
                    3 => Self::Transfer { amount },
                    4 => Self::Approve { amount },
                    7 => Self::MintTo { amount },
                    8 => Self::Burn { amount },
                    _ => unreachable!(),
                }
            }
            5 => Self::Revoke,
            6 => {
                let (authority_type, rest) = rest
                    .split_first()
                    .ok_or_else(|| ProgramError::from(InvalidInstruction))
                    .and_then(|(&t, rest)| Ok((AuthorityType::from(t)?, rest)))?;
                let (new_authority, _rest) = Self::unpack_pubkey_option(rest)?;

                Self::SetAuthority {
                    authority_type,
                    new_authority,
                }
            }
            9 => Self::CloseAccount,
            10 => Self::FreezeAccount,
            11 => Self::ThawAccount,
            12 => {
                let (amount, rest) = rest.split_at(8);
                let amount = amount
                    .try_into()
                    .ok()
                    .map(u64::from_le_bytes)
                    .ok_or(InvalidInstruction)?;
                let (&decimals, _rest) = rest.split_first().ok_or(InvalidInstruction)?;

                Self::TransferChecked { amount, decimals }
            }
            13 => {
                let (amount, rest) = rest.split_at(8);
                let amount = amount
                    .try_into()
                    .ok()
                    .map(u64::from_le_bytes)
                    .ok_or(InvalidInstruction)?;
                let (&decimals, _rest) = rest.split_first().ok_or(InvalidInstruction)?;

                Self::ApproveChecked { amount, decimals }
            }
            14 => {
                let (amount, rest) = rest.split_at(8);
                let amount = amount
                    .try_into()
                    .ok()
                    .map(u64::from_le_bytes)
                    .ok_or(InvalidInstruction)?;
                let (&decimals, _rest) = rest.split_first().ok_or(InvalidInstruction)?;

                Self::MintToChecked { amount, decimals }
            }
            15 => {
                let (amount, rest) = rest.split_at(8);
                let amount = amount
                    .try_into()
                    .ok()
                    .map(u64::from_le_bytes)
                    .ok_or(InvalidInstruction)?;
                let (&decimals, _rest) = rest.split_first().ok_or(InvalidInstruction)?;

                Self::BurnChecked { amount, decimals }
            }
            16 => {
                let (owner, _rest) = Self::unpack_pubkey(rest)?;
                Self::InitializeAccount2 { owner }
            }
            17 => Self::SyncNative,
            18 => {
                let (owner, _rest) = Self::unpack_pubkey(rest)?;
                Self::InitializeAccount3 { owner }
            }
            19 => {
                let &m = rest.get(0).ok_or(InvalidInstruction)?;
                Self::InitializeMultisig2 { m }
            }
            20 => {
                let (&decimals, rest) = rest.split_first().ok_or(InvalidInstruction)?;
                let (mint_authority, rest) = Self::unpack_pubkey(rest)?;
                let (freeze_authority, _rest) = Self::unpack_pubkey_option(rest)?;
                Self::InitializeMint2 {
                    mint_authority,
                    freeze_authority,
                    decimals,
                }
            }
            _ => return Err(TokenError::InvalidInstruction.into()),
        })
    }

    /// Packs a [TokenInstruction](enum.TokenInstruction.html) into a byte buffer.
    pub fn pack(&self) -> Vec<u8> {
        let mut buf = Vec::with_capacity(size_of::<Self>());
        match self {
            &Self::InitializeMint {
                ref mint_authority,
                ref freeze_authority,
                decimals,
            } => {
                buf.push(0);
                buf.push(decimals);
                buf.extend_from_slice(mint_authority.as_ref());
                Self::pack_pubkey_option(freeze_authority, &mut buf);
            }
            Self::InitializeAccount => buf.push(1),
            &Self::InitializeMultisig { m } => {
                buf.push(2);
                buf.push(m);
            }
            &Self::Transfer { amount } => {
                buf.push(3);
                buf.extend_from_slice(&amount.to_le_bytes());
            }
            &Self::Approve { amount } => {
                buf.push(4);
                buf.extend_from_slice(&amount.to_le_bytes());
            }
            &Self::MintTo { amount } => {
                buf.push(7);
                buf.extend_from_slice(&amount.to_le_bytes());
            }
            &Self::Burn { amount } => {
                buf.push(8);
                buf.extend_from_slice(&amount.to_le_bytes());
            }
            Self::Revoke => buf.push(5),
            Self::SetAuthority {
                authority_type,
                ref new_authority,
            } => {
                buf.push(6);
                buf.push(authority_type.into());
                Self::pack_pubkey_option(new_authority, &mut buf);
            }
            Self::CloseAccount => buf.push(9),
            Self::FreezeAccount => buf.push(10),
            Self::ThawAccount => buf.push(11),
            &Self::TransferChecked { amount, decimals } => {
                buf.push(12);
                buf.extend_from_slice(&amount.to_le_bytes());
                buf.push(decimals);
            }
            &Self::ApproveChecked { amount, decimals } => {
                buf.push(13);
                buf.extend_from_slice(&amount.to_le_bytes());
                buf.push(decimals);
            }
            &Self::MintToChecked { amount, decimals } => {
                buf.push(14);
                buf.extend_from_slice(&amount.to_le_bytes());
                buf.push(decimals);
            }
            &Self::BurnChecked { amount, decimals } => {
                buf.push(15);
                buf.extend_from_slice(&amount.to_le_bytes());
                buf.push(decimals);
            }
            &Self::InitializeAccount2 { owner } => {
                buf.push(16);
                buf.extend_from_slice(owner.as_ref());
            }
            &Self::SyncNative => {
                buf.push(17);
            }
            &Self::InitializeAccount3 { owner } => {
                buf.push(18);
                buf.extend_from_slice(owner.as_ref());
            }
            &Self::InitializeMultisig2 { m } => {
                buf.push(19);
                buf.push(m);
            }
            &Self::InitializeMint2 {
                ref mint_authority,
                ref freeze_authority,
                decimals,
            } => {
                buf.push(20);
                buf.push(decimals);
                buf.extend_from_slice(mint_authority.as_ref());
                Self::pack_pubkey_option(freeze_authority, &mut buf);
            }
        };
        buf
    }

    fn unpack_pubkey(input: &[u8]) -> Result<(Pubkey, &[u8]), ProgramError> {
        if input.len() >= 32 {
            let (key, rest) = input.split_at(32);
            let pk = Pubkey::new(key);
            Ok((pk, rest))
        } else {
            Err(TokenError::InvalidInstruction.into())
        }
    }

    fn unpack_pubkey_option(input: &[u8]) -> Result<(COption<Pubkey>, &[u8]), ProgramError> {
        match input.split_first() {
            Option::Some((&0, rest)) => Ok((COption::None, rest)),
            Option::Some((&1, rest)) if rest.len() >= 32 => {
                let (key, rest) = rest.split_at(32);
                let pk = Pubkey::new(key);
                Ok((COption::Some(pk), rest))
            }
            _ => Err(TokenError::InvalidInstruction.into()),
        }
    }

    fn pack_pubkey_option(value: &COption<Pubkey>, buf: &mut Vec<u8>) {
        match *value {
            COption::Some(ref key) => {
                buf.push(1);
                buf.extend_from_slice(&key.to_bytes());
            }
            COption::None => buf.push(0),
        }
    }
}

/// Specifies the authority type for SetAuthority instructions
#[repr(u8)]
#[derive(Clone, Debug, PartialEq)]
pub enum AuthorityType {
    /// Authority to mint new tokens
    MintTokens,
    /// Authority to freeze any account associated with the Mint
    FreezeAccount,
    /// Owner of a given token account
    AccountOwner,
    /// Authority to close a token account
    CloseAccount,
}

impl AuthorityType {
    fn into(&self) -> u8 {
        match self {
            AuthorityType::MintTokens => 0,
            AuthorityType::FreezeAccount => 1,
            AuthorityType::AccountOwner => 2,
            AuthorityType::CloseAccount => 3,
        }
    }

    fn from(index: u8) -> Result<Self, ProgramError> {
        match index {
            0 => Ok(AuthorityType::MintTokens),
            1 => Ok(AuthorityType::FreezeAccount),
            2 => Ok(AuthorityType::AccountOwner),
            3 => Ok(AuthorityType::CloseAccount),
            _ => Err(TokenError::InvalidInstruction.into()),
        }
    }
}

/// Creates a `InitializeMint` instruction.
pub fn initialize_mint(
    token_program_id: &Pubkey,
    mint_pubkey: &Pubkey,
    mint_authority_pubkey: &Pubkey,
    freeze_authority_pubkey: Option<&Pubkey>,
    decimals: u8,
) -> Result<Instruction, ProgramError> {
    check_program_account(token_program_id)?;
    let freeze_authority = freeze_authority_pubkey.cloned().into();
    let data = TokenInstruction::InitializeMint {
        mint_authority: *mint_authority_pubkey,
        freeze_authority,
        decimals,
    }
    .pack();

    let accounts = vec![
        AccountMeta::new(*mint_pubkey, false),
        AccountMeta::new_readonly(sysvar::rent::id(), false),
    ];

    Ok(Instruction {
        program_id: *token_program_id,
        accounts,
        data,
    })
}

/// Creates a `InitializeMint2` instruction.
pub fn initialize_mint2(
    token_program_id: &Pubkey,
    mint_pubkey: &Pubkey,
    mint_authority_pubkey: &Pubkey,
    freeze_authority_pubkey: Option<&Pubkey>,
    decimals: u8,
) -> Result<Instruction, ProgramError> {
    check_program_account(token_program_id)?;
    let freeze_authority = freeze_authority_pubkey.cloned().into();
    let data = TokenInstruction::InitializeMint2 {
        mint_authority: *mint_authority_pubkey,
        freeze_authority,
        decimals,
    }
    .pack();

    let accounts = vec![AccountMeta::new(*mint_pubkey, false)];

    Ok(Instruction {
        program_id: *token_program_id,
        accounts,
        data,
    })
}

/// Creates a `InitializeAccount` instruction.
pub fn initialize_account(
    token_program_id: &Pubkey,
    account_pubkey: &Pubkey,
    mint_pubkey: &Pubkey,
    owner_pubkey: &Pubkey,
) -> Result<Instruction, ProgramError> {
    check_program_account(token_program_id)?;
    let data = TokenInstruction::InitializeAccount.pack();

    let accounts = vec![
        AccountMeta::new(*account_pubkey, false),
        AccountMeta::new_readonly(*mint_pubkey, false),
        AccountMeta::new_readonly(*owner_pubkey, false),
        AccountMeta::new_readonly(sysvar::rent::id(), false),
    ];

    Ok(Instruction {
        program_id: *token_program_id,
        accounts,
        data,
    })
}

/// Creates a `InitializeAccount2` instruction.
pub fn initialize_account2(
    token_program_id: &Pubkey,
    account_pubkey: &Pubkey,
    mint_pubkey: &Pubkey,
    owner_pubkey: &Pubkey,
) -> Result<Instruction, ProgramError> {
    check_program_account(token_program_id)?;
    let data = TokenInstruction::InitializeAccount2 {
        owner: *owner_pubkey,
    }
    .pack();

    let accounts = vec![
        AccountMeta::new(*account_pubkey, false),
        AccountMeta::new_readonly(*mint_pubkey, false),
        AccountMeta::new_readonly(sysvar::rent::id(), false),
    ];

    Ok(Instruction {
        program_id: *token_program_id,
        accounts,
        data,
    })
}

/// Creates a `InitializeAccount3` instruction.
pub fn initialize_account3(
    token_program_id: &Pubkey,
    account_pubkey: &Pubkey,
    mint_pubkey: &Pubkey,
    owner_pubkey: &Pubkey,
) -> Result<Instruction, ProgramError> {
    check_program_account(token_program_id)?;
    let data = TokenInstruction::InitializeAccount3 {
        owner: *owner_pubkey,
    }
        .pack();

    let accounts = vec![
        AccountMeta::new(*account_pubkey, false),
        AccountMeta::new_readonly(*mint_pubkey, false),
    ];

    Ok(Instruction {
        program_id: *token_program_id,
        accounts,
        data,
    })
}

/// Creates a `InitializeMultisig` instruction.
pub fn initialize_multisig(
    token_program_id: &Pubkey,
    multisig_pubkey: &Pubkey,
    signer_pubkeys: &[&Pubkey],
    m: u8,
) -> Result<Instruction, ProgramError> {
    check_program_account(token_program_id)?;
    if !is_valid_signer_index(m as usize)
        || !is_valid_signer_index(signer_pubkeys.len())
        || m as usize > signer_pubkeys.len()
    {
        return Err(ProgramError::MissingRequiredSignature);
    }
    let data = TokenInstruction::InitializeMultisig { m }.pack();

    let mut accounts = Vec::with_capacity(1 + 1 + signer_pubkeys.len());
    accounts.push(AccountMeta::new(*multisig_pubkey, false));
    accounts.push(AccountMeta::new_readonly(sysvar::rent::id(), false));
    for signer_pubkey in signer_pubkeys.iter() {
        accounts.push(AccountMeta::new_readonly(**signer_pubkey, false));
    }

    Ok(Instruction {
        program_id: *token_program_id,
        accounts,
        data,
    })
}

/// Creates a `InitializeMultisig2` instruction.
pub fn initialize_multisig2(
    token_program_id: &Pubkey,
    multisig_pubkey: &Pubkey,
    signer_pubkeys: &[&Pubkey],
    m: u8,
) -> Result<Instruction, ProgramError> {
    check_program_account(token_program_id)?;
    if !is_valid_signer_index(m as usize)
        || !is_valid_signer_index(signer_pubkeys.len())
        || m as usize > signer_pubkeys.len()
    {
        return Err(ProgramError::MissingRequiredSignature);
    }
    let data = TokenInstruction::InitializeMultisig2 { m }.pack();

    let mut accounts = Vec::with_capacity(1 + 1 + signer_pubkeys.len());
    accounts.push(AccountMeta::new(*multisig_pubkey, false));
    for signer_pubkey in signer_pubkeys.iter() {
        accounts.push(AccountMeta::new_readonly(**signer_pubkey, false));
    }
    Ok(Instruction {
        program_id: *token_program_id,
        accounts,
        data,
    })
}

/// Creates a `Transfer` instruction.
pub fn transfer(
    token_program_id: &Pubkey,
    source_pubkey: &Pubkey,
    destination_pubkey: &Pubkey,
    authority_pubkey: &Pubkey,
    signer_pubkeys: &[&Pubkey],
    amount: u64,
) -> Result<Instruction, ProgramError> {
    check_program_account(token_program_id)?;
    let data = TokenInstruction::Transfer { amount }.pack();

    let mut accounts = Vec::with_capacity(3 + signer_pubkeys.len());
    accounts.push(AccountMeta::new(*source_pubkey, false));
    accounts.push(AccountMeta::new(*destination_pubkey, false));
    accounts.push(AccountMeta::new_readonly(
        *authority_pubkey,
        signer_pubkeys.is_empty(),
    ));
    for signer_pubkey in signer_pubkeys.iter() {
        accounts.push(AccountMeta::new_readonly(**signer_pubkey, true));
    }

    Ok(Instruction {
        program_id: *token_program_id,
        accounts,
        data,
    })
}

/// Creates an `Approve` instruction.
pub fn approve(
    token_program_id: &Pubkey,
    source_pubkey: &Pubkey,
    delegate_pubkey: &Pubkey,
    owner_pubkey: &Pubkey,
    signer_pubkeys: &[&Pubkey],
    amount: u64,
) -> Result<Instruction, ProgramError> {
    check_program_account(token_program_id)?;
    let data = TokenInstruction::Approve { amount }.pack();

    let mut accounts = Vec::with_capacity(3 + signer_pubkeys.len());
    accounts.push(AccountMeta::new(*source_pubkey, false));
    accounts.push(AccountMeta::new_readonly(*delegate_pubkey, false));
    accounts.push(AccountMeta::new_readonly(
        *owner_pubkey,
        signer_pubkeys.is_empty(),
    ));
    for signer_pubkey in signer_pubkeys.iter() {
        accounts.push(AccountMeta::new_readonly(**signer_pubkey, true));
    }

    Ok(Instruction {
        program_id: *token_program_id,
        accounts,
        data,
    })
}
```

```rust
/// Creates a `Revoke` instruction.
pub fn revoke(
    token_program_id: &Pubkey,
    source_pubkey: &Pubkey,
    owner_pubkey: &Pubkey,
    signer_pubkeys: &[&Pubkey],
) -> Result<Instruction, ProgramError> {
    check_program_account(token_program_id)?;
    let data = TokenInstruction::Revoke.pack();

    let mut accounts = Vec::with_capacity(2 + signer_pubkeys.len());
    accounts.push(AccountMeta::new(*source_pubkey, false));
    accounts.push(AccountMeta::new_readonly(
        *owner_pubkey,
        signer_pubkeys.is_empty(),
    ));
    for signer_pubkey in signer_pubkeys.iter() {
        accounts.push(AccountMeta::new_readonly(**signer_pubkey, true));
    }

    Ok(Instruction {
        program_id: *token_program_id,
        accounts,
        data,
    })
}

/// Creates a `SetAuthority` instruction.
pub fn set_authority(
    token_program_id: &Pubkey,
    owned_pubkey: &Pubkey,
    new_authority_pubkey: Option<&Pubkey>,
    authority_type: AuthorityType,
    owner_pubkey: &Pubkey,
    signer_pubkeys: &[&Pubkey],
) -> Result<Instruction, ProgramError> {
    check_program_account(token_program_id)?;
    let new_authority = new_authority_pubkey.cloned().into();
    let data = TokenInstruction::SetAuthority {
        authority_type,
        new_authority,
    }
    .pack();

    let mut accounts = Vec::with_capacity(3 + signer_pubkeys.len());
    accounts.push(AccountMeta::new(*owned_pubkey, false));
    accounts.push(AccountMeta::new_readonly(
        *owner_pubkey,
        signer_pubkeys.is_empty(),
    ));
    for signer_pubkey in signer_pubkeys.iter() {
        accounts.push(AccountMeta::new_readonly(**signer_pubkey, true));
    }

    Ok(Instruction {
        program_id: *token_program_id,
        accounts,
        data,
    })
}

/// Creates a `MintTo` instruction.
pub fn mint_to(
    token_program_id: &Pubkey,
    mint_pubkey: &Pubkey,
    account_pubkey: &Pubkey,
    owner_pubkey: &Pubkey,
    signer_pubkeys: &[&Pubkey],
    amount: u64,
) -> Result<Instruction, ProgramError> {
    check_program_account(token_program_id)?;
    let data = TokenInstruction::MintTo { amount }.pack();

    let mut accounts = Vec::with_capacity(3 + signer_pubkeys.len());
    accounts.push(AccountMeta::new(*mint_pubkey, false));
    accounts.push(AccountMeta::new(*account_pubkey, false));
    accounts.push(AccountMeta::new_readonly(
        *owner_pubkey,
        signer_pubkeys.is_empty(),
    ));
    for signer_pubkey in signer_pubkeys.iter() {
        accounts.push(AccountMeta::new_readonly(**signer_pubkey, true));
    }

    Ok(Instruction {
        program_id: *token_program_id,
        accounts,
        data,
    })
}

/// Creates a `Burn` instruction.
pub fn burn(
    token_program_id: &Pubkey,
    account_pubkey: &Pubkey,
    mint_pubkey: &Pubkey,
    authority_pubkey: &Pubkey,
    signer_pubkeys: &[&Pubkey],
    amount: u64,
) -> Result<Instruction, ProgramError> {
    check_program_account(token_program_id)?;
    let data = TokenInstruction::Burn { amount }.pack();

    let mut accounts = Vec::with_capacity(3 + signer_pubkeys.len());
    accounts.push(AccountMeta::new(*account_pubkey, false));
    accounts.push(AccountMeta::new(*mint_pubkey, false));
    accounts.push(AccountMeta::new_readonly(
        *authority_pubkey,
        signer_pubkeys.is_empty(),
    ));
    for signer_pubkey in signer_pubkeys.iter() {
        accounts.push(AccountMeta::new_readonly(**signer_pubkey, true));
    }

    Ok(Instruction {
        program_id: *token_program_id,
        accounts,
        data,
    })
}

/// Creates a `CloseAccount` instruction.
pub fn close_account(
    token_program_id: &Pubkey,
    account_pubkey: &Pubkey,
    destination_pubkey: &Pubkey,
    owner_pubkey: &Pubkey,
    signer_pubkeys: &[&Pubkey],
) -> Result<Instruction, ProgramError> {
    check_program_account(token_program_id)?;
    let data = TokenInstruction::CloseAccount.pack();
    let mut accounts = Vec::with_capacity(3 + signer_pubkeys.len());
    accounts.push(AccountMeta::new(*account_pubkey, false));
    accounts.push(AccountMeta::new(*destination_pubkey, false));
    accounts.push(AccountMeta::new_readonly(
        *owner_pubkey,
        signer_pubkeys.is_empty(),
    ));
    for signer_pubkey in signer_pubkeys.iter() {
        accounts.push(AccountMeta::new_readonly(**signer_pubkey, true));
    }

    Ok(Instruction {
        program_id: *token_program_id,
        accounts,
        data,
    })
}

/// Creates a `FreezeAccount` instruction.
pub fn freeze_account(
    token_program_id: &Pubkey,
    account_pubkey: &Pubkey,
    mint_pubkey: &Pubkey,
    owner_pubkey: &Pubkey,
    signer_pubkeys: &[&Pubkey],
) -> Result<Instruction, ProgramError> {
    check_program_account(token_program_id)?;
    let data = TokenInstruction::FreezeAccount.pack();

    let mut accounts = Vec::with_capacity(3 + signer_pubkeys.len());
    accounts.push(AccountMeta::new(*account_pubkey, false));
    accounts.push(AccountMeta::new_readonly(*mint_pubkey, false));
    accounts.push(AccountMeta::new_readonly(
        *owner_pubkey,
        signer_pubkeys.is_empty(),
    ));
    for signer_pubkey in signer_pubkeys.iter() {
        accounts.push(AccountMeta::new_readonly(**signer_pubkey, true));
    }

    Ok(Instruction {
        program_id: *token_program_id,
        accounts,
        data,
    })
}

/// Creates a `ThawAccount` instruction.
pub fn thaw_account(
    token_program_id: &Pubkey,
    account_pubkey: &Pubkey,
    mint_pubkey: &Pubkey,
    owner_pubkey: &Pubkey,
    signer_pubkeys: &[&Pubkey],
) -> Result<Instruction, ProgramError> {
    check_program_account(token_program_id)?;
    let data = TokenInstruction::ThawAccount.pack();

    let mut accounts = Vec::with_capacity(3 + signer_pubkeys.len());
    accounts.push(AccountMeta::new(*account_pubkey, false));
    accounts.push(AccountMeta::new_readonly(*mint_pubkey, false));
    accounts.push(AccountMeta::new_readonly(
        *owner_pubkey,
        signer_pubkeys.is_empty(),
    ));
    for signer_pubkey in signer_pubkeys.iter() {
        accounts.push(AccountMeta::new_readonly(**signer_pubkey, true));
    }

    Ok(Instruction {
        program_id: *token_program_id,
        accounts,
        data,
    })
}

/// Creates a `TransferChecked` instruction.
#[allow(clippy::too_many_arguments)]
pub fn transfer_checked(
    token_program_id: &Pubkey,
    source_pubkey: &Pubkey,
    mint_pubkey: &Pubkey,
    destination_pubkey: &Pubkey,
    authority_pubkey: &Pubkey,
    signer_pubkeys: &[&Pubkey],
    amount: u64,
    decimals: u8,
) -> Result<Instruction, ProgramError> {
    check_program_account(token_program_id)?;
    let data = TokenInstruction::TransferChecked { amount, decimals }.pack();

    let mut accounts = Vec::with_capacity(4 + signer_pubkeys.len());
    accounts.push(AccountMeta::new(*source_pubkey, false));
    accounts.push(AccountMeta::new_readonly(*mint_pubkey, false));
    accounts.push(AccountMeta::new(*destination_pubkey, false));
    accounts.push(AccountMeta::new_readonly(
        *authority_pubkey,
        signer_pubkeys.is_empty(),
    ));
    for signer_pubkey in signer_pubkeys.iter() {
        accounts.push(AccountMeta::new_readonly(**signer_pubkey, true));
    }

    Ok(Instruction {
        program_id: *token_program_id,
        accounts,
        data,
    })
}

/// Creates an `ApproveChecked` instruction.
#[allow(clippy::too_many_arguments)]
pub fn approve_checked(
    token_program_id: &Pubkey,
    source_pubkey: &Pubkey,
    mint_pubkey: &Pubkey,
    delegate_pubkey: &Pubkey,
    owner_pubkey: &Pubkey,
    signer_pubkeys: &[&Pubkey],
    amount: u64,
    decimals: u8,
) -> Result<Instruction, ProgramError> {
    check_program_account(token_program_id)?;
    let data = TokenInstruction::ApproveChecked { amount, decimals }.pack();

    let mut accounts = Vec::with_capacity(4 + signer_pubkeys.len());
    accounts.push(AccountMeta::new(*source_pubkey, false));
    accounts.push(AccountMeta::new_readonly(*mint_pubkey, false));
    accounts.push(AccountMeta::new_readonly(*delegate_pubkey, false));
    accounts.push(AccountMeta::new_readonly(
        *owner_pubkey,
        signer_pubkeys.is_empty(),
    ));
    for signer_pubkey in signer_pubkeys.iter() {
        accounts.push(AccountMeta::new_readonly(**signer_pubkey, true));
    }

    Ok(Instruction {
        program_id: *token_program_id,
        accounts,
        data,
    })
}

/// Creates a `MintToChecked` instruction.
pub fn mint_to_checked(
    token_program_id: &Pubkey,
    mint_pubkey: &Pubkey,
    account_pubkey: &Pubkey,
    owner_pubkey: &Pubkey,
    signer_pubkeys: &[&Pubkey],
    amount: u64,
    decimals: u8,
) -> Result<Instruction, ProgramError> {
    check_program_account(token_program_id)?;
    let data = TokenInstruction::MintToChecked { amount, decimals }.pack();

    let mut accounts = Vec::with_capacity(3 + signer_pubkeys.len());
    accounts.push(AccountMeta::new(*mint_pubkey, false));
    accounts.push(AccountMeta::new(*account_pubkey, false));
    accounts.push(AccountMeta::new_readonly(
        *owner_pubkey,
        signer_pubkeys.is_empty(),
    ));
    for signer_pubkey in signer_pubkeys.iter() {
        accounts.push(AccountMeta::new_readonly(**signer_pubkey, true));
    }

    Ok(Instruction {
        program_id: *token_program_id,
        accounts,
        data,
    })
}

/// Creates a `BurnChecked` instruction.
pub fn burn_checked(
    token_program_id: &Pubkey,
    account_pubkey: &Pubkey,
    mint_pubkey: &Pubkey,
    authority_pubkey: &Pubkey,
    signer_pubkeys: &[&Pubkey],
    amount: u64,
    decimals: u8,
) -> Result<Instruction, ProgramError> {
    check_program_account(token_program_id)?;
    let data = TokenInstruction::BurnChecked { amount, decimals }.pack();

    let mut accounts = Vec::with_capacity(3 + signer_pubkeys.len());
    accounts.push(AccountMeta::new(*account_pubkey, false));
    accounts.push(AccountMeta::new(*mint_pubkey, false));
    accounts.push(AccountMeta::new_readonly(
        *authority_pubkey,
        signer_pubkeys.is_empty(),
    ));
    for signer_pubkey in signer_pubkeys.iter() {
        accounts.push(AccountMeta::new_readonly(**signer_pubkey, true));
    }

    Ok(Instruction {
        program_id: *token_program_id,
        accounts,
        data,
    })
}

/// Creates a `SyncNative` instruction
pub fn sync_native(
    token_program_id: &Pubkey,
    account_pubkey: &Pubkey,
) -> Result<Instruction, ProgramError> {
    check_program_account(token_program_id)?;

    Ok(Instruction {
        program_id: *token_program_id,
        accounts: vec![AccountMeta::new(*account_pubkey, false)],
        data: TokenInstruction::SyncNative.pack(),
    })
}

/// Utility function that checks index is between MIN_SIGNERS and MAX_SIGNERS
pub fn is_valid_signer_index(index: usize) -> bool {
    (MIN_SIGNERS..=MAX_SIGNERS).contains(&index)
}
```

Analysis of audit results

Re-Entrancy

- **Description:**
  One of the features of program is the ability to call and utilise code of other external program. These external calls can be hijacked by attackers whereby they force the program to execute further code , including calls back into itself. Thus the code execution "re-enters" the program.
- **Detection results:**
  ```
  PASSED!
  ```
- **Security suggestion:**
  no.

Arithmetic Over/Under Flows

- **Description:**
  The Virtual Machine specifies fixed-size data types for integers. This means that an integer variable, only has a certain range of numbers it can represent. A uint8 for example, can only store numbers in the range [0,255]. Trying to store 256 into a uint8 will result in 0. If care is not taken, variables can be exploited if user input is unchecked and calculations are performed which result in numbers that lie outside the range of the data type that stores them.
- **Detection results:**
  ```
  PASSED!
  ```

- **Security suggestion:**
  no.

Replay Attacks

- **Description:**
  A replay attack occurs when a cybercriminal eavesdrops on a secure network communication, intercepts it, and then fraudulently delays or resends it to misdirect the receiver into doing what the hacker wants.
- **Detection results:**

```
PASSED!
```

- **Security suggestion:** no.

Denial Of Service (DOS)

- **Description:**
  This category is very broad, but fundamentally consists of attacks where users can leave the program inoperable for a small period of time, or in some cases, permanently. This can trap Blockchain Currency in these program forever, as was the case with the Second Parity MultiSig hack
- **Detection results:**

```
PASSED!
```

- **Security suggestion:**
  no.

Race Conditions / Front Running

- **Description:**
  The combination of external calls to other program and the multi-user nature of the underlying blockchain gives rise to a variety of potential pitfalls whereby users race code execution to obtain unexpected states. Re-Entrancy is one example of such a race condition. In this section we will talk more generally about different kinds of race conditions that can occur on the blockchain. There is a variety of good posts on this subject, a few are: Wiki - Safety, DASP - Front-Running and the Consensus - program Best Practices.
- **Detection results:**

```
PASSED!
```

- **Security suggestion:**
  no.

Permission restrictions

- **Description:**
  program managers who can control liquidity or pledge pools, etc., or impose unreasonable restrictions on other users.
- **Detection results:**

```
PASSED!
```

- **Security suggestion:**
  no.

Variable declaration and scope

- **Description:**
  Variable bindings have a scope, and are constrained to live in a block.
- **Detection results:**

```
PASSED!
```

- **Security suggestion:** no.

External program Referencing

- **Description:**
  One of the benefits of the global computer is the ability to re-use code and interact with program already deployed on the network. As a result, a large number of program reference external program and in general operation use external message calls to interact with these program. These external message calls can mask malicious actors intentions in some non-obvious ways, which we will discuss.
- **Detection results:**

```
PASSED!
```

- **Security suggestion:**
  no.

Forged account

- **Description:**
  Forged account checked.
- **Detection results:**

```
PASSED!
```

- **Security suggestion:**
  no.

Unchecked CALL Return Values

- **Description:**
  There a number of ways of performing external calls. Sending Blockchain Currency to external accounts is commonly performed via the transfer() method. However, the send() function can also be used and, for more versatile external calls, the CALL opcode can be directly employed. The call() and send() functions return a boolean indicating if the call succeeded or failed. Thus these functions have a simple caveat, in that the transaction that executes these functions will not revert if the external call (intialised by call() or send()) fails, rather the call() or send() will simply return false. A common pitfall arises when the return value is not checked, rather the developer expects a revert to occur.
- **Detection results:**

```
PASSED!
```

- **Security suggestion:**
  no.

Floating Points and Numerical Precision

- **Description:**
  Rust has IEEE 754 single precision (32-bit) and double precision (64-bit) types. This can lead to errors/vulnerabilities if not implemented correctly.
- **Detection results:**

```
PASSED!
```

- **Security suggestion:**
  no.

Unsolved TODO comments

- **Description:**
  Check for Unsolved TODO comments
- **Detection results:**

```
PASSED!
```

- **Security suggestion:**
  no.

Business logic defect

- **Description:**
  We do business logic checks where necessary。
- **Detection results:**

```
PASSED!
```

- **Security suggestion:**
  no.

armors.io

contact@armors.io