Untitled diff

Eliminare Salvare Condividere

Real-time diff
Unified diff
Collapse lines

HIGHLIGHT CHANGE
Parola Lettera

SYNTAX HIGHLIGHTING
Choose syntax

STRUMENTI
Convertire in minuscolo
Ordinare le righe
Sostituire le interruzioni di riga con spazi
Tagliare gli spazi bianchi

Confrontare e unire
Esportare su formato PDF

Editore

14 removals 123 lines Copia tutti

```
use borsh::{BorshDeserialize, BorshSerialize};
use solana_program::{
    account_info::{next_account_info, AccountInfo},
    entrypoint::ProgramResult,
    program::{invoke, invoke_signed},
    program_error::ProgramError,
    pubkey::Pubkey,
    rent::Rent,
    system_instruction,
    sysvar::Sysvar,
};

use crate::{Wallet, WalletInstruction, WALLET_LEN};

pub fn process_instruction(
    program_id: &Pubkey,
    accounts: &[AccountInfo],
    mut instruction_data: &[u8],
) -> ProgramResult {
    match WalletInstruction::deserialize(&mut instruction_data)? {
        WalletInstruction::Initialize => initialize(program_id, accounts),
        WalletInstruction::Deposit { amount } => deposit(program_id, accounts, amount),
        WalletInstruction::Withdraw { amount } => withdraw(program_id, accounts, amount),
    }
}

fn initialize(program_id: &Pubkey, accounts: &[AccountInfo]) -> ProgramResult {
    let account_info_iter = &mut accounts.iter();
    let wallet_info = next_account_info(account_info_iter)?;
    let vault_info = next_account_info(account_info_iter)?;
    let authority_info = next_account_info(account_info_iter)?;
    let rent_info = next_account_info(account_info_iter)?;
    let (wallet_address, wallet_seed) =
        Pubkey::find_program_address(&[&authority_info.key.to_bytes()], program_id);
    let (vault_address, vault_seed) = Pubkey::find_program_address(
        &[&authority_info.key.to_bytes(), &"VAULT".as_bytes()],
        program_id,
    );

    let rent = Rent::from_account_info(rent_info)?;

    assert_eq!(*wallet_info.key, wallet_address);
    assert!(wallet_info.data_is_empty());

    invoke_signed(
        &system_instruction::create_account(
            &authority_info.key,
            &wallet_address,
            rent.minimum_balance(WALLET_LEN as usize),
            WALLET_LEN,
            &program_id,
        ),
        &[authority_info.clone(), wallet_info.clone()],
        &[&[&authority_info.key.to_bytes(), &[wallet_seed]]],
    )?;

    invoke_signed(
        &system_instruction::create_account(
            &authority_info.key,
            &vault_address,
            rent.minimum_balance(0),
            0,
            &program_id,
        ),
        &[authority_info.clone(), vault_info.clone()],
        &[&[
            &authority_info.key.to_bytes(),
            &"VAULT".as_bytes(),
            &[vault_seed],
        ]],
    )?;

    let wallet = Wallet {
        authority: *authority_info.key,
        vault: vault_address,
    };

    wallet
        .serialize(&mut &mut (*wallet_info.data).borrow_mut()[..])
        .unwrap();

    Ok(())
}

fn deposit(_program_id: &Pubkey, accounts: &[AccountInfo], amount: u64) -> ProgramResult {
    let account_info_iter = &mut accounts.iter();
    let wallet_info = next_account_info(account_info_iter)?;
```

12 additions 122 lines Copia tutti

```
use borsh::{BorshDeserialize, BorshSerialize};
use solana_program::{
    account_info::{next_account_info, AccountInfo},
    entrypoint::ProgramResult,
    program::{invoke, invoke_signed},
    program_error::ProgramError,
    pubkey::Pubkey,
    rent::Rent,
    system_instruction,
    sysvar::Sysvar,
};

use crate::{Wallet, WalletInstruction, WALLET_LEN};

pub fn process_instruction(
    program_id: &Pubkey,
    accounts: &[AccountInfo],
    mut instruction_data: &[u8],
) -> ProgramResult {
    match WalletInstruction::deserialize(&mut instruction_data)? {
        WalletInstruction::Initialize => initialize(program_id, accounts),
        WalletInstruction::Deposit { amount } => deposit(program_id, accounts, amount),
        WalletInstruction::Withdraw { amount } => withdraw(program_id, accounts, amount),
    }
}

fn initialize(program_id: &Pubkey, accounts: &[AccountInfo]) -> ProgramResult {
    let account_info_iter = &mut accounts.iter();
    let wallet_info = next_account_info(account_info_iter)?;
    let vault_info = next_account_info(account_info_iter)?;
    let authority_info = next_account_info(account_info_iter)?;
    let rent_info = next_account_info(account_info_iter)?;

    // Vulnerabilità: Uso di un bump seed hardcoded
    let bump_seed = 255;
    let wallet_seed = &[&authority_info.key.to_bytes()[..], &[bump_seed]];
    let vault_seed = &[&authority_info.key.to_bytes()[..], b"VAULT", &[bump_seed]];

    let (wallet_address, _wallet_bump) =
        Pubkey::find_program_address(wallet_seed, program_id);
    let (vault_address, _vault_bump) = Pubkey::find_program_address(vault_seed, program_id);

    let rent = Rent::from_account_info(rent_info)?;

    assert_eq!(*wallet_info.key, wallet_address);
    assert!(wallet_info.data_is_empty());

    invoke_signed(
        &system_instruction::create_account(
            &authority_info.key,
            &wallet_address,
            rent.minimum_balance(WALLET_LEN as usize),
            WALLET_LEN,
            &program_id,
        ),
        &[authority_info.clone(), wallet_info.clone()],
        &[wallet_seed],
    )?;

    invoke_signed(
        &system_instruction::create_account(
            &authority_info.key,
            &vault_address,
            rent.minimum_balance(0),
            0,
            &program_id,
        ),
        &[authority_info.clone(), vault_info.clone()],
        &[vault_seed],
    )?;

    let wallet = Wallet {
        authority: *authority_info.key,
        vault: vault_address,
    };

    wallet
        .serialize(&mut &mut (*wallet_info.data).borrow_mut()[..])
        .unwrap();

    Ok(())
}

fn deposit(_program_id: &Pubkey, accounts: &[AccountInfo], amount: u64) -> ProgramResult {
    let account_info_iter = &mut accounts.iter();
    let wallet_info = next_account_info(account_info_iter)?;
```

```
    let vault_info = next_account_info(account_info_iter)?;
    let source_info = next_account_info(account_info_iter)?;
    let wallet = Wallet::deserialize(&mut &(*wallet_info.data).borrow_mut()[..])?;

    assert_eq!(wallet.vault, *vault_info.key);

    invoke(
        &system_instruction::transfer(&source_info.key, &vault_info.key, amount),
        &[vault_info.clone(), source_info.clone()],
    )?;

    Ok(())
}

fn withdraw(_program_id: &Pubkey, accounts: &[AccountInfo], amount: u64) -> ProgramResult {
    let account_info_iter = &mut accounts.iter();
    let wallet_info = next_account_info(account_info_iter)?;
    let vault_info = next_account_info(account_info_iter)?;
    let authority_info = next_account_info(account_info_iter)?;
    let destination_info = next_account_info(account_info_iter)?;
    let wallet = Wallet::deserialize(&mut &(*wallet_info.data).borrow_mut()[..])?;

    assert!(authority_info.is_signer);
    assert_eq!(wallet.authority, *authority_info.key);
    assert_eq!(wallet.vault, *vault_info.key);

    if amount > **vault_info.lamports.borrow_mut() {
        return Err(ProgramError::InsufficientFunds);
    }

    **vault_info.lamports.borrow_mut() -= amount;
    **destination_info.lamports.borrow_mut() += amount;

    Ok(())
}
```

```
    let vault_info = next_account_info(account_info_iter)?;
    let source_info = next_account_info(account_info_iter)?;
    let wallet = Wallet::deserialize(&mut &(*wallet_info.data).borrow_mut()[..])?;

    assert_eq!(wallet.vault, *vault_info.key);

    invoke(
        &system_instruction::transfer(&source_info.key, &vault_info.key, amount),
        &[vault_info.clone(), source_info.clone()],
    )?;

    Ok(())
}

fn withdraw(_program_id: &Pubkey, accounts: &[AccountInfo], amount: u64) -> ProgramResult {
    let account_info_iter = &mut accounts.iter();
    let wallet_info = next_account_info(account_info_iter)?;
    let vault_info = next_account_info(account_info_iter)?;
    let authority_info = next_account_info(account_info_iter)?;
    let destination_info = next_account_info(account_info_iter)?;
    let wallet = Wallet::deserialize(&mut &(*wallet_info.data).borrow_mut()[..])?;

    assert!(authority_info.is_signer);
    assert_eq!(wallet.authority, *authority_info.key);
    assert_eq!(wallet.vault, *vault_info.key);

    if amount > **vault_info.lamports.borrow_mut() {
        return Err(ProgramError::InsufficientFunds);
    }

    **vault_info.lamports.borrow_mut() -= amount;
    **destination_info.lamports.borrow_mut() += amount;

    Ok(())
}
```

Diff salvati

Your saved diffs will appear here.

Testo originale Open file

```
    assert_eq!(wallet.authority, *authority_info.key);
    assert_eq!(wallet.vault, *vault_info.key);

    if amount > **vault_info.lamports.borrow_mut() {
        return Err(ProgramError::InsufficientFunds);
    }

    **vault_info.lamports.borrow_mut() -= amount;
    **destination_info.lamports.borrow_mut() += amount;

    Ok(())
}
```

Testo modificato Open file

```
    assert_eq!(wallet.authority, *authority_info.key);
    assert_eq!(wallet.vault, *vault_info.key);

    if amount > **vault_info.lamports.borrow_mut() {
        return Err(ProgramError::InsufficientFunds);
    }

    **vault_info.lamports.borrow_mut() -= amount;
    **destination_info.lamports.borrow_mut() += amount;

    Ok(())
}
```


Trovare la differenza