Capstone Project: On-Chain Program Architecture

1. Overview

This document outlines the architecture of the **Capstone** on-chain program, a comprehensive financial system built on the Solana blockchain. The program is designed as a single, monolithic smart contract that logically separates functionality into five core components: a Vault Program, a Spending Program, a Yield Program, a Liquidation Program, and a Payment Program. These components work in concert to manage user funds, provide spending capabilities based on collateral, generate yield, ensure platform safety through liquidations, and process merchant payments.

2. State Account Architecture

The program utilizes several Program Derived Addresses (PDAs) to manage user and system state securely. Each account type serves a distinct purpose.

- **`Vault`**: The primary account for each user, acting as their personal collateral vault.
  - `authority`: The user's public key, who owns the account.
  - `balance`: A `u64` representing the amount of native SOL held as collateral.
  - `bump`: The PDA bump seed.
- **`SpendingAccount`**: Tracks a user's spending power and history.
  - `authority`: The user's public key.
  - `spending_limit`: A `u64` representing the total amount the user is authorized to spend, derived from their vault collateral.
  - `amount_spent`: A `u64` tracking the user's current debt or total amount spent against their limit.
  - `bump`: The PDA bump seed.
- **`Treasury`**: A single, global PDA owned by the program that acts as a central pot for all user funds that are staked to earn yield. Its balance is tracked by the Solana runtime's native lamport count.
  - `bump`: The PDA bump seed.
- **`YieldAccount`**: A personal account for each user that tracks their individual contribution to the central `Treasury`.
  - `authority`: The user's public key.
  - `staked_amount`: A `u64` representing the amount of SOL the user has personally staked in the treasury.
  - `bump`: The PDA bump seed.
- **`MerchantAccount`**: Represents a merchant registered with the platform who can receive payments.
  - `authority`: The public key of the merchant's wallet that will receive payments.
  - `name`: A `String` for the merchant's display name.

    ○ `bump`: The PDA bump seed.

3. Detailed Instruction Flows

The program's logic is exposed through a series of instructions, grouped by their function.

3.1. Vault Program Flows

- **`initialize(ctx)`**: Creates a new `Vault` PDA for the calling user. Sets the `authority` to the user and initializes the `balance` to 0.
- **`deposit(ctx, amount)`**: Transfers a specified `amount` of SOL from the user's wallet into their `Vault` PDA, increasing the vault's `balance`.
- **`withdraw(ctx, amount)`**: Transfers a specified `amount` of SOL from the user's `Vault` PDA back to their wallet, decreasing the vault's `balance`.

3.2. Spending Program Flows

- **`initialize_spending_account(ctx)`**: Creates a new `SpendingAccount` PDA for the user, initializing `spending_limit` and `amount_spent` to 0.
- **`update_spending_limit(ctx)`**: Reads the user's `Vault` balance and calculates their spending power, updating the `spending_limit` in their `SpendingAccount`. (Current logic sets this to 50% of the vault balance).
- **`authorize_spend(ctx, amount)`**: A utility instruction that checks if a purchase `amount` is within the user's available spending limit and updates their `amount_spent` accordingly. This is the core logic used by `process_payment`.
- **`reset_spend_tracker(ctx)`**: A developer utility instruction to reset a user's `amount_spent` to 0 for testing purposes.

3.3. Yield Program Flows

- **`initialize_treasury(ctx)`**: A one-time instruction to create the global `Treasury` PDA.
- **`initialize_yield_account(ctx)`**: Creates a new `YieldAccount` PDA for a user to track their staked funds.
- **`stake(ctx, amount)`**: Moves a specified `amount` of SOL from a user's `Vault` into the central `Treasury`. It decreases the user's `Vault.balance` and increases their `YieldAccount.staked_amount`.
- **`unstake(ctx, amount)`**: Moves a specified `amount` of SOL from the central `Treasury` back to a user's `Vault`. It decreases the user's `YieldAccount.staked_amount` and increases their `Vault.balance`.

3.4. Liquidation Program Flows

- **`liquidate(ctx)`**: The core safety mechanism. This instruction reads the user's staked collateral value (using the Pyth price feed), compares it to their debt (`amount_spent`), and calculates their collateralization ratio. If the ratio falls below a safety threshold (e.g., 120%), it performs a partial liquidation by "selling" just enough staked SOL from the treasury to repay a portion of the debt and restore the user's position to a healthy ratio (e.g., 150%).

3.5. Payment Program Flows

- **`initialize_merchant_account(ctx, name)`**: Creates a new `MerchantAccount` PDA for a new merchant, setting their `name` and `authority`.
- **`process_payment(ctx, amount)`**: The final settlement instruction. It first authorizes the spend against the user's `SpendingAccount`, then executes a CPI to the System Program to transfer the `amount` in SOL from the user's wallet to the merchant's authority wallet.

4. External Integration Points

- **Price Oracles**: The `liquidate` instruction integrates directly with the **Pyth Network** by reading one of its on-chain price feed accounts to get a reliable, real-time SOL/USD price for collateral valuation.

5. Security and Technical Details

The program employs several key security patterns common in Solana development:

- **Program Derived Addresses (PDAs)** are used for all state accounts, ensuring the program itself is the sole owner and preventing unauthorized modifications.
- **`has_one` Constraints** are used extensively in `Accounts` structs to validate that the signer of a transaction is the legitimate owner of the accounts they are trying to modify.
- **`address` Constraints** are used in the `process_payment` instruction to guarantee that the payment is sent to the correct wallet address associated with the merchant account.
- **Manual Lamport Transfers** are used instead of CPIs for withdrawing from PDAs with data (`Vault`, `Treasury`), which is a critical security pattern to prevent certain classes of errors.