Constraint-based Security

The security of the Anchor program is based on its simplicity and use of explicit account constraints. A trusted front-end, and our MPC signer through Para, increase security for our program by making sure signature requests are valid only at our application's endpoint.

The user that is signed-in via Para, such as through an email or phone number, is the "Owner" account. The Owner creates a Merchant account using the Owner's pubkey and the Merchant's name as seeds, thereby cryptographically linking them. An Owner can create Multiple Merchant accounts as long as they have unique names (for that Owner).

For the Owner's and Merchant's callable instructions we have an optional fee_payer that we operate, probably through a hidden Para server wallet that is available to sign and pay all fees. If supplied in the transaction by the front-end, then the fee_payer will pay even for account initialization and rent fees.

The Owner's pubkey is stored in the Merchant PDA and used as a check on the Owner account input, though it's technically redundant, as the Owner's key is mandatory in the Merchant account's derivation.

The USDC mint is expressly declared and constrained. This could be changed to an array of acceptable mints, such as USDC, USDT, USDPY, etc.. All derived USDC ATA's have their authority pubkey's expressly defined and constrained as well, such as the Owner, and HOUSE.

The Merchant PDA is the authority of its own USDC ATA, and therefore must sign any instruction to transfer coins out of that account. The Merchant PDA can also only sign transactions and do things defined within the program itself, further enhancing security of this account and its funds.

The Owner must sign for every transaction in which it is involved. The front-end must therefore be trusted, and by using Para's MPC infrastructure we gain a bit of layering in terms of the security of our signer and the transactions they are signing.

An explicit AUTH, operated by us, lets us control which Merchant accounts we are willing to pay transaction fees for, through the use of the is_active boolean. If you are a known entity, you will be active TRUE, if not, you will be FALSE. You must now pay your gas fees, but can still operate.

An Owner can update any of its Merchant's refund limits, thereby reducing this attack surface at leaast slightly. It is technically an open attack surface on the Merchant's USDC ATA, as the original_tx_sig input is novel, and coming from the front-end. This is why we must have our front-end be incredibly secure, and the environment/endpoint for which our Para API keys are willing to sign for be constrained to only our

```
Create Merchant Context

#[instruction(name: String)]

    #[account(mut)]
    pub fee_payer: Option<Signer<'info>>,

    #[account(mut, constraint = !name.trim().is_empty() @ CustomError::InvalidMerchantName)]
    pub owner: Signer<'info>,

    #[account(init, payer = fee_payer.as_ref().unwrap_or(&owner), seeds = [b"merchant", name.as_str().as_bytes(), owner.key().as_ref()], space = Merchant::LEN, bump)]
    pub merchant: Box<Account<'info, Merchant>>,

    #[account(constraint = usdc_mint.key() == Pubkey::from_str(USDC_DEVNET_MINT).unwrap())]
    pub usdc_mint: Box<InterfaceAccount<'info, Mint>>,

    #[account(
        init, payer = fee_payer.as_ref().unwrap_or(&owner), associated_token::mint = usdc_mint, associated_token::authority = merchant
    )]
    pub merchant_usdc_ata: Box<InterfaceAccount<'info, TokenAccount>>,

... other accounts
```

```
Withdraw USDC Context

... other accounts

    #[account(mut, constraint = amount > 0 @ CustomError::ZeroAmountWithdrawal)]
    pub owner: Signer<'info>,

    #[account(
        seeds = [b"merchant", merchant.entity_name.as_str().as_bytes(), owner.key().as_ref()],
        bump = merchant.merchant_bump,
        constraint = owner.key() == merchant.owner @ CustomError::NotMerchantOwner
    )]
    pub merchant: Box<Account<'info, Merchant>>,

... other accounts
```

```
Novel Program Accounts

#[account]
pub struct Merchant {
    pub owner: Pubkey,
    pub entity_name: String,
    pub total_withdrawn: u64,
    pub total_refunded: u64,
    pub is_active: bool,
    pub refund_limit: u64,
    pub merchant_bump: u8,
}

#[account]
pub struct RefundRecord {
    pub amount: u64,
    pub original_tx_sig: String,
    pub bump: u8,
}
```

```
Refund Context notes

#[instruction(original_tx_sig: String, amount: u64)]

... other accounts

    #[account(mut,
        constraint = amount > 0 @ CustomError::ZeroAmountWithdrawal)]
    pub owner: Signer<'info>,

    #[account(
        init, payer = fee_payer.as_ref().unwrap_or(&owner), seeds = [b"refund", original_tx_sig.as_bytes()], space = RefundRecord::LEN, bump)]
    pub refund_record: Box<Account<'info, RefundRecord>>,

... other accounts
```

```
Set Merchant Status context

    #[account(mut, constraint = auth.key() == Pubkey::from_str(AUTH).unwrap() @ CustomError::UnauthorizedStatusChange)]
    pub auth: Signer<'info>,

    #[account(mut)]
    pub merchant: Box<Account<'info, Merchant>>,

... other accounts
```

```
Update Refund limit context

    #[account(mut)]
    pub fee_payer: Option<Signer<'info>>,

    #[account(mut)]
    pub owner: Signer<'info>,

    #[account(
        mut,
        seeds = [b"merchant", merchant.entity_name.as_str().as_bytes(), owner.key().as_ref()],
        bump = merchant.merchant_bump,
        constraint = owner.key() == merchant.owner @ CustomError::NotMerchantOwner
    )]
    pub merchant: Box<Account<'info, Merchant>>,

... other accounts
```