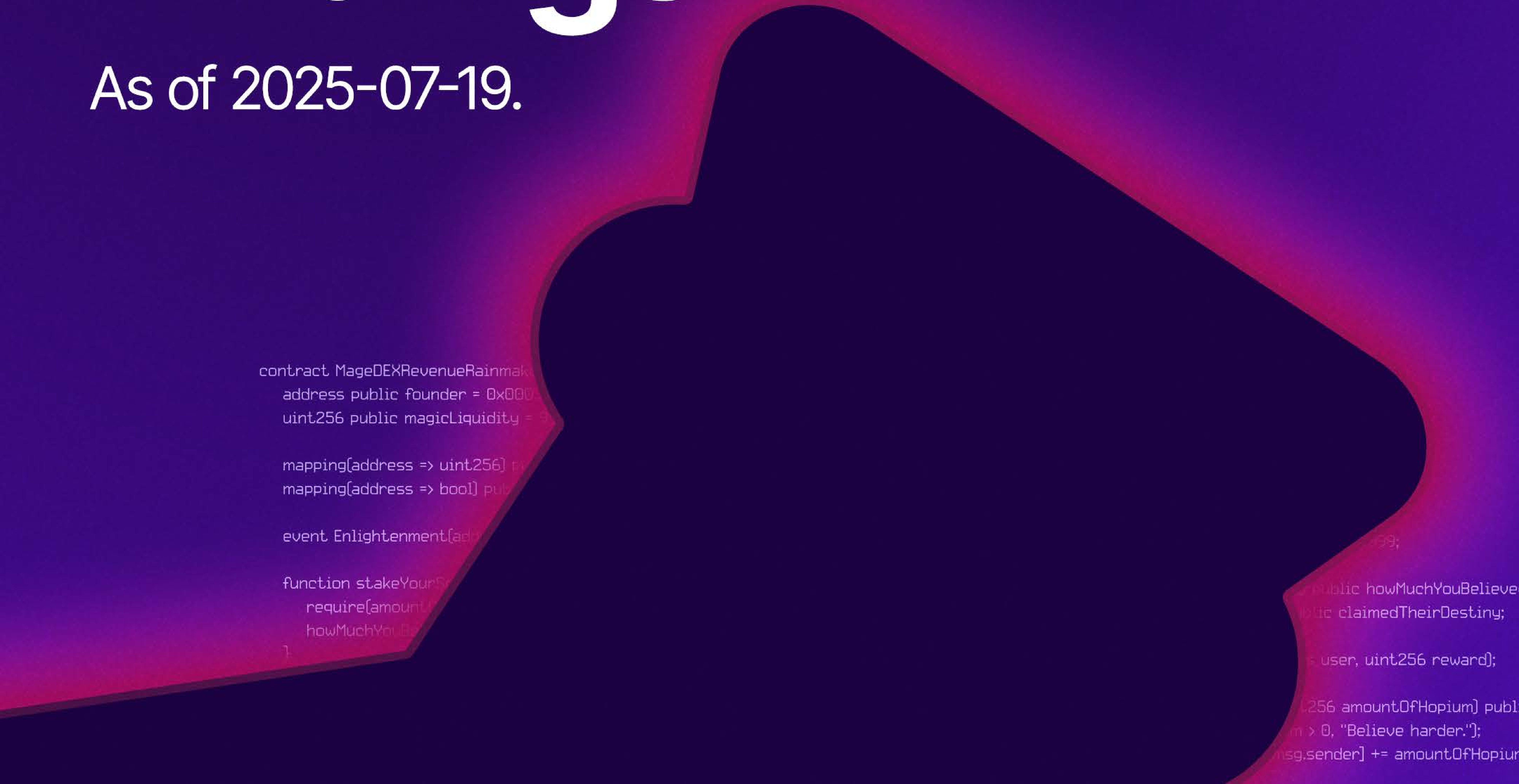
Preliminary Security Audit Report of Mage Labs: Findings

As of 2025-07-19.



```
contract MageDEXRevenueRainmak
    address public founder = 0x000
    uint256 public magicLiquidity =

    mapping(address => uint256)
    mapping(address => bool) pub

    event Enlightenment(ad

    function stakeYourS
        require(amount
        howMuchYouBe
    }
```

```
99;

 public howMuchYouBelieve
lic claimedTheirDestiny;

 user, uint256 reward);

256 amountOfHopium) publ
m > 0, "Believe harder.");
sg.sender] += amountOfHopium
```

Closing Accounts

Improperly closing accounts without marking them as closed, enabling reuse/exploitation.

//programs/magelabs/src/instructions/close.rs:14, 23-26

`account_to_close` is not actually closed. Only the lamports are moved.

1. No zeroing out the account data.

2. Not marking the account as closed using CLOSED_ACCOUNT_DISCRIMINATOR

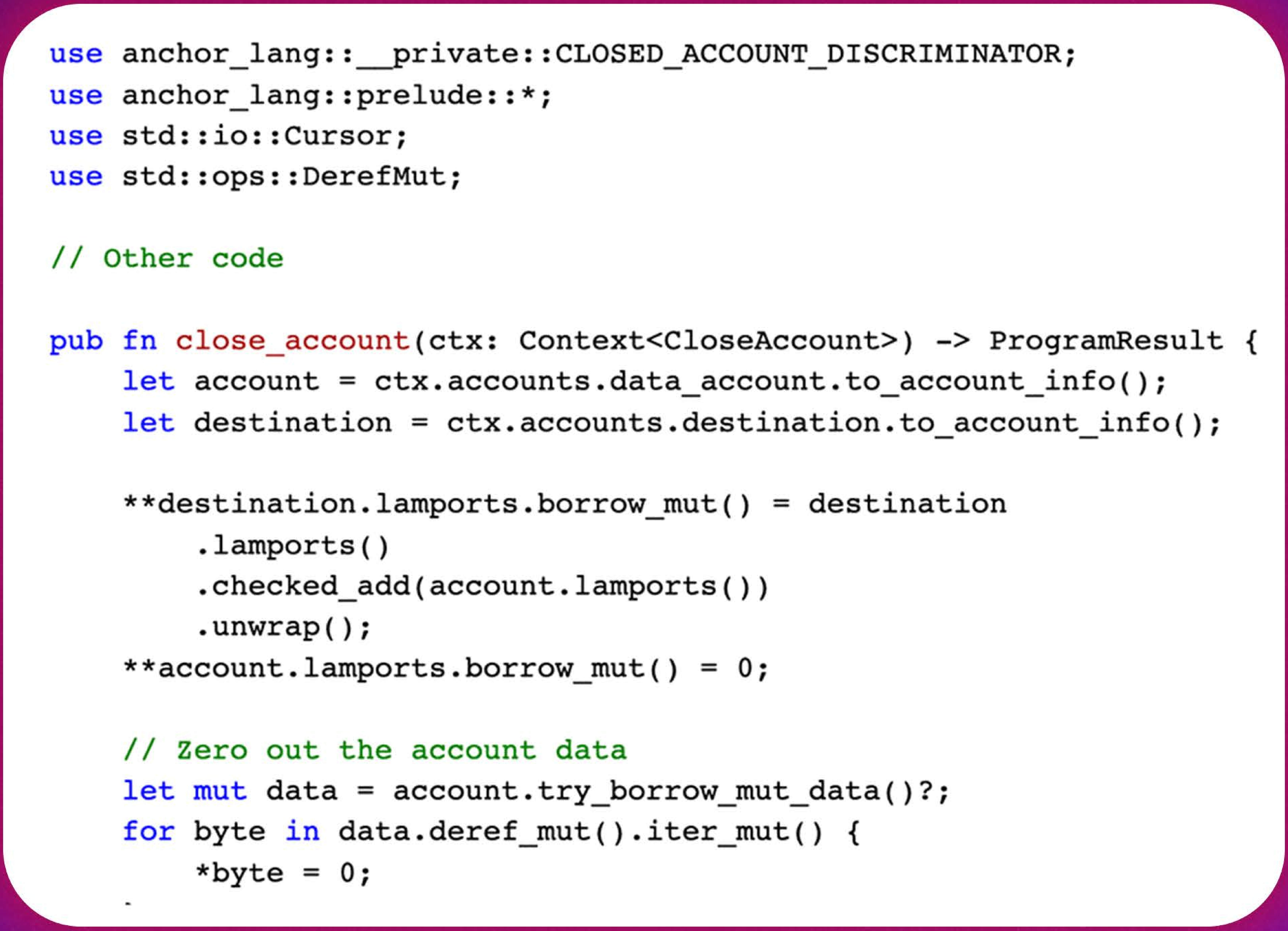
3. No constraint on account_to_close

4. No force_defund implemented: if users accidentally send tokens to the closed account_to_close, there is no way to withdraw.

Follow the good practice in the following example.

Or

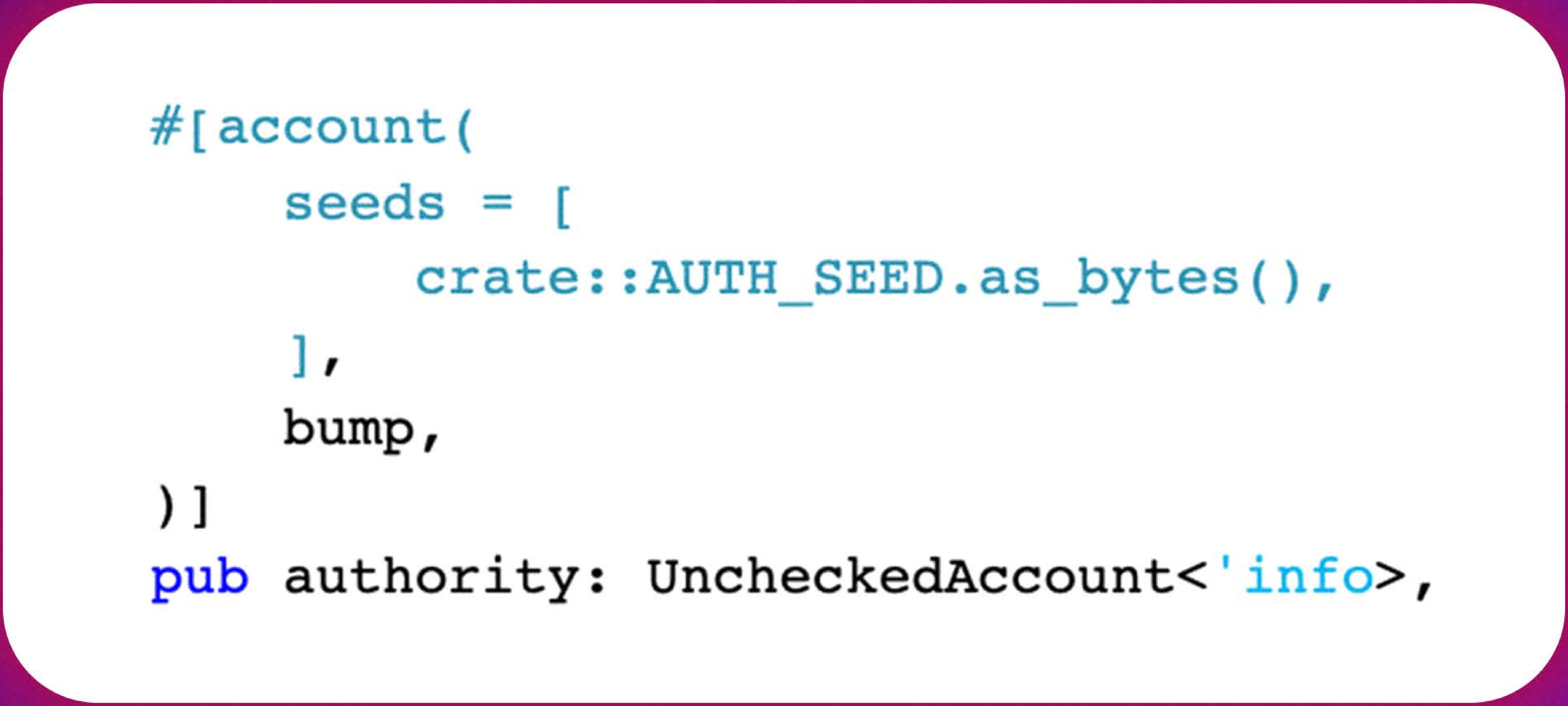
Add #[account(close = admin)] constraint, automating the secure closure of accounts by transferring lamports, zeroing data, and setting the closed account discriminator, all in one operation.



```rust
use anchor_lang::__private::CLOSED_ACCOUNT_DISCRIMINATOR;
use anchor_lang::prelude::*;
use std::io::Cursor;
use std::ops::DerefMut;

// Other code

pub fn close_account(ctx: Context<CloseAccount>) -> ProgramResult {
    let account = ctx.accounts.data_account.to_account_info();
    let destination = ctx.accounts.destination.to_account_info();

    **destination.lamports.borrow_mut() = destination
        .lamports()
        .checked_add(account.lamports())
        .unwrap();
    **account.lamports.borrow_mut() = 0;

    // Zero out the account data
    let mut data = account.try_borrow_mut_data()?;
    for byte in data.deref_mut().iter_mut() {
        *byte = 0;
```

Missing Ownership Check

//programs/magelabs/src/instructions/deposit.rs, initialize_metadata.rs, intialize.rs, swap_base_input.rs, withdraw.rs, collect_fund_fee.rs, collect_protocol_fee.rs



```
#[account(
    seeds = [
        crate::AUTH_SEED.as_bytes(),
    ],
    bump,
)]
pub authority: UncheckedAccount<'info>,
```

Memory Size Calculation

It is discouraged to use std::mem::size_of to compute the account data len. It does not reflect the actual data size on-chain.

```
const _: () = {
    assert!(
        std::mem::size_of::<AmmConfig>() == 232,
        "AmmConfig size must be 232 bytes (based on raydium-cp-swap
amm config size)"
    );
    ()
};
```

Use INIT_SPACE instead since AmmConfig derives InitSpace.

```
AmmConfig::INIT_SPACE == 231,
```

Similar issue occurs in,
//programs/magelabs/src/states/pool.rs:118

Pubkey comparison

Use require_key_eq instead of require_eq.
Because require_eq only ensures two NON-PUBKEY values are equal.

```
require_eq!(
        ctx.accounts.authority.key(),
        pool_state.pool_creator,
        ErrorCode::Unauthorized
    );
```

Bad Practices

Not use get() for Sysvars
//programs/magelabs/src/instructions/lock_fee_anchor.rs

```
pub clock: Sysvar<'info, Clock>,
...
pool_state.anchor_lock_time = ctx.accounts.clock.unix_timestamp;
```

No need to pass in clock as an account.
Use Clock:get() instead.
It improves efficiency and simplicity. Consistent with other instructions where Clock::get is used instead of passing clock.

Unhandled Panic

Pervasive in the code.
e.g.

//programs/magelabs/src/instructions/swap_base_input.rs

```
pool_state.swap_fees_token_0.checked_add(swap_fee).unwrap();
```

Redundant Code

//programs/magelabs/src/instructions/withdraw.rs:115, 137-141
Variables are not used. Other unused variables to be summarized here.

```
let _user = ctx.accounts.user.key();
```

Mage
Labs

Comments incomplete or ambiguous

e.g.
//programs/magelabs/src/instructions/swap_base_input.rs

The comment before swap_base_input()

This comment claims that you don't do anything with the swap_fee. However, the code explicitly calculates and stores all three fee components (swap_fee, creator_fee, and protocol_fee) in the pool_state.

//programs/magelabs/src/instructions/swap_base_output.rs
There is no comment before swap_base_output()

2025-07-20

Verification of programs/magelabs/src/curve

//programs/magelabs/src/curve/calculator.rs:232

check_curve_value_from_swap.

Pass the verification
0 < source_token_amount < bound
0 < swap_source_amount < bound,
0 < swap_destination_amount < bound
where bound == u64::MAX

//programs/magelabs/src/curve/calculator.rs:273
check_pool_value_from_deposit

Pass the verification
0 < lp_token_amount < bound
lp_token_amount <= lp_token_supply < bound
0 < swap_token_0_amount < bound
0 < swap_token_1_amount < bound
lp_token_amount * swap_token_0_amount >= lp_token_supply
lp_token_amount * swap_token_1_amount >= lp_token_supply
where bound == 50

//programs/magelabs/src/curve/calculator.rs:320
check_pool_value_from_withdraw

Pass the verification

0 < lp_token_amount < bound
lp_token_amount <= lp_token_supply < bound
0 < swap_token_0_amount < bound
0 < swap_token_1_amount < bound
lp_token_amount * swap_token_0_amount >= lp_token_supply
lp_token_amount * swap_token_1_amount >= lp_token_supply
where bound == 10

2025-07-23 Meeting

Reimplementation of Close

```
//programs/magelabs/src/instructions/close.rs
use crate::error::ErrorCode;
use anchor_lang::prelude::*;

#[derive(Accounts)]
pub struct Close<'info> {
    #[account(
        mut,
        address = crate::admin::ID @ ErrorCode::SignerIsNotAdmin
    )]
    pub admin: Signer<'info>,

    /// CHECK: The account to close. The account must be owned by
this program.
    /// Never close those accounts that are essential to the protocol's
integrity.
    ///
    /// If you know the specific type of account to be closed,
    /// it’s better to use a strongly-typed constraint like:
    /// #[account(mut, close = admin)]
    /// pub account_to_close: Account<'info, MyData>
    ///
    /// This approach ensures type safety and helps avoid accidental
closure
    /// of critical accounts. Ideally, only close PDAs created by this
program,
    /// and never those that are essential to the protocol's integrity.
    ///
    /// However, in this case, we do not know the exact account type at
runtime.
    /// Therefore, although not recommended, we use
`UncheckedAccount`.
    /// To mitigate risk, we explicitly add an `owner = <program_id>`
constraint
    /// to ensure the account is owned by this program.
    ///
    /// Be cautious: this means *any* account owned by the program
    /// can potentially be closed by the admin, which carries significant
risk.
    #[account(mut, owner = crate::ID)]
    pub account_to_close: UncheckedAccount<'info>,

    pub system_program: Program<'info, System>,
}
```

```
pub fn close(ctx: Context<Close>) → Result<()> {
    let dest_starting_lamports = ctx.accounts.admin.lamports();
    let source_account_lamports =
ctx.accounts.account_to_close.lamports();

    **ctx.accounts.admin.try_borrow_mut_lamports()? =
dest_starting_lamports
        .checked_add(source_account_lamports)
        .ok_or(ErrorCode::ArithmeticError)?;  // replace unwrap with error
code.
    **ctx.accounts.account_to_close.try_borrow_mut_lamports()? = 0;

    // Zero out the account data
    let mut data =
ctx.accounts.account_to_close.try_borrow_mut_data()?;
    for byte in data.iter_mut() {
        *byte = 0;
    }

    // Note:
    // CLOSED_ACCOUNT_DISCRIMINATOR was removed from 0.30.
    // Instead, data is zeroed out, including the discriminator.
    // Thus the account is marked as deallocated, from the Solana
runtime's point of view (by zeroing data).
    // I have also checked the generated code from #[close=...], it also
just zeros out the data.

    Ok(())
}
```