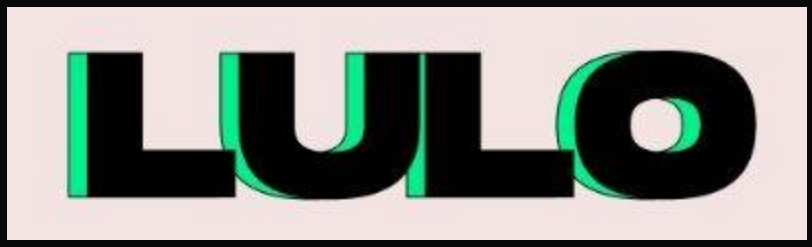
Security Assessment Report



Lulo

January 2025

Prepared for Lulo

certora

LULO

Table of content

certora

LULO

Project Summary

Project Scope

Project Name	Repository (link)	Audited Commits	Platform
Lulo	https://github.com/lulo-labs/flexlend/tree/feat/v2-audit	4b0e0a7 - initial 2b4ec2a - latest including fixes	Solana

Project Overview

This document describes the discovered findings during the audit of **Lulo smart contracts** using manual code review. The work was undertaken from **18 November 2024** to **23 January 2025.**

The following contract list is included in our scope:

```
/programs/flexlend/src/instructions/pool/*
/programs/flexlend/src/state/pool/*
/programs/flexlend/src/lib.rs
/programs/flexlend/src/state/user.rs
/programs/flexlend/src/utils/{ price.rs, math.rs, checks.rs}
```

The Certora team performed a manual audit of all the Solana contracts listed above**.** During the manual audit, the Certora team discovered bugs in the Solana contracts code, as listed on the following page.

Findings Summary

The table below summarizes the findings of the review, including type and severity details.

Severity	Discovered	Confirmed	Fixed
Critical	7	7	7
High	4	4	4
Medium	8	8	8
Low	0	0	0
Informational	9	8	7
Total	28	28	26

Severity Matrix

Impact	High	Medium	High	Critical
	Medium	Low	Medium	High
	Low	Low	Low	Medium
		Low	Medium	High
		Likelihood		

Detailed Findings

Critical Severity Issues

C-01 The `complete_regular_withdraw` instruction does not update oracles but requires oracle info is latest

Severity: **Critical**	Impact: **High**	Likelihood: **High**
Files: instructions/pool/complete_regular_withdraw.rs	Status: Fixed	

Description: The `complete_regular_withdraw` instruction requires that the oracle information of the `input_mint` is not stale without calling the `Pool::update_oracles` function. As a result, user's withdraws might fail.

```
Unset
pub fn complete_regular_withdraw<'a, 'b, 'c, 'info>(
    ctx: Context<'a, 'b, 'c, 'info, CompleteWithdraw<'info>>,
    withdrawal_id: u16,
) -> Result<()> {
    [...]
    pool.error_if_stale_refresh(pool.allocations[allocation_index], current_ts)?;
```

A snippet of the `complete_regular_withdraw` instruction handler in [complete_regular_withdraw.rs#L76]

Exploit Scenario:

1. Alice deposits `10k` USDC into the regular pool
2. After a while, Alice initiates withdrawal of total `11k` USDC from the regular pool.
3. After `cooldown_seconds` time has elapsed, oracle information became stale.

4. Alice uses the protocol frontend to complete the withdrawal. Frontend calls the `complete_regular_withdraw` instruction.
    - withdrawal fails as the oracle information is stale.

Alice has to wait for some operation to happen in the protocol which updates the `USDC` token oracle information.

Recommendations: Update the `complete_regular_withdraw` instruction handler to call the `Pool::update_oracles` function before calling the `Pool::error_if_stale_refresh` function.

Customer's response: Fixed in commit 2b4ec2a.

Fix Review: Fix confirmed.

C-02 Missing call to `Pool::error_if_stale_refresh` function in `initiate_regular_withdraw` instruction handler

Severity: **Critical**	Impact: **High**	Likelihood: **High**
Files: instructions/pool/initiate_regular_withdraw.rs	Status: Fixed	

Description: The `initiate_regular_withdraw` instruction does not call the `Pool::error_if_stale_refresh` function. As a result, the instruction succeeds even if the pool is not refereshed and price information is stale. This allows users to skip the price safety checks and initiate withdrawal.

Recommendations: Add a call to `Pool::error_if_stale_refresh` in the `initiate_regular_withdraw` instruction handler.

Customer's response: Fixed in commit 2b4ec2a.

Fix Review: Fix confirmed.

C-03 Referrers can claim assets by claiming fees from a different allocation

Severity: **Critical**	Impact: **High**	Likelihood: **High**
Files: /src/state/user.rs	Status: Fixed	

Description: The referral shares and the amounts are tracked separetely for each allocation. However, a user referral shares for all allocations is tracked in a single value `UserAccount::referral_supply`. As a result, the program cannot distinguish referral shares based on the allocation allowing a referrer to get referral shares from one allocation and claim referral fees from a different allocation with better exchange rate.

```
Unset
#[account]
#[derive(Debug, Default, InitSpace)]
pub struct UserAccount {
    [...]
    // up to 5 mints
    pub deposited_allocations: [u64; POOL_MAX_ALLOCATIONS],

    /// referred_amount
    pub referred_amount: u64,
    pub referral_supply: u64,
    pub referral_fee_unclaimed: u64,
    [...]
}
```

Definition of the `UserAccount` struct in state/user.rs

Exploit Scenario:

1. The Lulo protocol supports USDC token

   - `referral_exchange_rate` for USDC is `0.9`.
   - `USDC::accumulated_referral_fees == 1000 USDC`
2. Lulo protocol adds support for USDT. The `referral_exchange_rate` starts at `1`.
3. Eve, an attacker, refers EveX and EveX deposits `1000` USDT into the Lulo protocol
   - `Eve::referred_amount = 1000` (6 decimals)
   - `referral_supply = 1000`
   - `referral_fee_unclaimed = 0`
4. Eve calls `claim_referral_fee` instruction using the USDC mint
   - `UserAccount::calculate_claimable_referral_rewards` computes the claimable amount
     i. `referred_c_amount = 0.9 * 1000 = 900`
     ii. `claimable_c_amount = referral_supply - referred_c_amount = 1000 - 900 = 100`
     iii. `claimable_amount = 100 / 0.9 = 111 USDC`
   - Eve receives `111` USDC as referral fees
5. EveX withdraws `1000` USDT

Eve can deposit into an allocation with lowest `referral_exchange_rate` to get `referral_shares` and withdraw the accumulated referral fees from rest of the allocations.

Recommendations: Update the `UserAccount` to store `referral_supply` for each allocation separately.

Customer's response: Fixed in commit 2b4ec2a.

Fix Review: Fix confirmed.

C-04 Attacker can make themselves the referrer of existing user accounts

Severity: **Critical**	Impact: **High**	Likelihood: **High**
Files: instructions/pool/deposit_pool.rs	Status: Fixed	

Description: Because the `deposit_pool` instruction does not validate that the given `owner` account is the actual owner of the provided `user_account`, an attacker can call `deposit_pool` for a user account with unintialized referrer and set `referrer` to their own address. Attacker can repeat this for other user accounts and gain from referrer fees.

The `deposit_pool` instruction ensures a certain `owner` account signed the transaction. However, it does not check the address of the `owner` is the provided `UserAccount::owner` allowing anyone to perform deposits on behalf of other user's `UserAccount`'s.

```
Unset
#[derive(Accounts)]
pub struct DepositPool<'info> {
    #[account(mut)]
    pub owner: Signer<'info>,

    [...]
    /// CHECK: user_account can be initialized in the instruction
    #[account(mut)]
    pub user_account: UncheckedAccount<'info>,

    [...]

    #[account(mut)]
    pub referrer_user_account: Option<Account<'info, UserAccount>>,
```

Definition of Accounts context for `deposit_pool` instruction.

The `deposit_pool` instruction allows to set the `referrer` address for the `UserAccount`

```
Unset
pub fn deposit_pool(ctx: Context<DepositPool>, amount: u64, pool_type: PoolType) ->
Result<()> {
    [...]

    // if referrer is set once then a referrer account must be passed for all future calls
    if let Some(referrer) = &mut ctx.accounts.referrer_user_account {
        if user.referrer.eq(&Pubkey::default()) {
            user.referrer = referrer.key();
        }
        [...]
}
```

A snippet of the `deposit_pool` function in [deposit_pool.rs]

If the `referrer` is not initialized for a `UserAccount`, an attacker can deposit a minimal amount and provide their own account as the referrer. The `deposit_pool` function will set the `referrer` address to the attacker account.

As a result, all future deposits of this user must pass the attacker's account as referrer. The attacker earns from the referrer fees for the user. Attacker can repeat this for all users with uninitialized `referrer`.

Exploit Scenario:

1. Bob, an honest user, creates a user account without a referrer.
2. Eve, an attacker, finds accounts with uninitialized `referrer` including Bob's account.
3. Eve calls the `deposit_pool` instruction using Bob's user account with 1 USD amount and referrer Eve's user account.
    - The `deposit_pool` sets the Bob's `UserAccount::referrer` to Eve's user account.
4. Bob uses front-end to perform deposits and front-end uses Eve's account as referrer for the deposit call.
    - Eve account is updated with referral balances and starts accuring fees

Eve repeats this for all other accounts and earns referrer from fees.

Recommendations: Add following anchor constraint to the `user_account` account:

```
Unset
    #[account(
        mut,
        seeds = [
            b"flexlend",
            owner.key().as_ref(),
        ],
        bump,
    )]
    pub user_account: UncheckedAccount<'info>,,
```

The constraint ensures that the `user_account` is a valid PDA and the provided `owner` is the owner of the `user_account`.

Customer's response: Fixed in commit 2b4ec2a.

Fix Review: Fix confirmed.

C-05 Anchor account state overwrite issue allows attackers to arbitrage

Severity: **Critical**	Impact: **High**	Likelihood: **High**
Files: instructions/pool/deposit_pool.rs, instructions/pool/initiate_regular_withdraw.rs	Status: Fixed	

Description: Attacker can set referrer to their own account and cause state overwrite issue in user withdrawal instructions. As a result, changes performed to the user account are not recorded allowing attackers to withdraw more amount from an allocation than deposited and arbitage between stablecoins.

The `withdraw_protected_pool` instruction takes in two `UserAccount` type accounts, `user_account` and `referrer`.

Unset

```
#[derive(Accounts)]
pub struct WithdrawProtectedPool<'info> {
    [...]
    #[account(
        mut,
        seeds = [
            b"flexlend",
            owner.key().as_ref(),
        ],
        bump,
    )]
    pub user_account: Box<Account<'info, UserAccount>>,

    [...]
    #[account(mut, address = user_account.referrer)]
```

```
    pub referrer_user_account: Option<Account<'info, UserAccount>>,
}
```

Definition of the `WithdrawProtectedPool` accounts struct in [withdraw_protected_pool.rs#L233-L252]

The program allows an user to set their own account has their referrer. As a result, the `user_account` and the `referrer_user_account` can be same accounts.

```
Unset
user_account == referrer_user_account == user_account.referrer
```

This leads to state overwrite issues because of how Anchor `#[derive(Accounts)]` works.

Anchor `#[derive(Accounts)]` macro works by creating an in-memory copy of the account data, modifying the memory and then writing back into the account data:

For `Account<'a, T>` type accounts, Anchor deserializes the `AccountInfo::data` into `T`. For e.g, if account `X` is passed for `Account<'a, UserAccount>` then Anchor deserializes `X.data` into type `UserAccount` and keeps the copy in memory.

The deserialized in-memory copies of the accounts are passed to the instruction-handler. After the instruction handler finishes execution, Anchor serializes the accounts and writes back into the account data. The account data is persistent and hence the state changes are saved.

Consider a Solidity function which copies a struct state variable into memory, changes the values in memory and at the end of the function copies the memory values into the state variable. Anchor internally uses the same approach for `Account<'a, T>` type accounts.

This leads to a state overwrite issue when two of the passed-in accounts are same. When the second struct is written into the account data, it will overwrite any changes that are made to the first struct.

As a result, if the `user_account`, and `referrer_user_account` are the same accounts and because `referrer_user_account` is placed after `user_account` in the `Accounts` struct, changes performed to the `user_account` will be overwritten and are not recorded.

The `withdraw_protected_pool` instruction updates `UserAccount::deposited_allocations[allocation_index]` of the `user_account`.

Unset

```
    /// updates the `deposited_allocations` array on the user for this withdrawal,
    /// this check prevents a user from withdrawing an asset that was not deposited
    pub fn withdraw_allocation(
        &mut self,
        allocation_index: usize,
        withdraw_supply: u64,
        withdraw_all: bool,
    ) -> Result<()> {
        let deposited_supply = self.deposited_allocations[allocation_index];

        if deposited_supply == 0 {
            msg!("asset {} not deposited", allocation_index);
            return Err(error!(Errors::InvalidPoolWithdraw));
        }
        [...]
        self.deposited_allocations[allocation_index] =
self.deposited_allocations[allocation_index]
            .checked_sub(withdraw_supply)
            .unwrap_or(0);

        Ok(())
    }
```

Definition of the `UserAccount::withdraw_allocation` function in [state/user.rs#L92-L98]

The `UserAccount::deposited_allocations` tracks user accounts deposits into each of the allocations, i.e each of deposit tokens. The `deposited_allocations` values are used to disallow user from withdrawing a token that was not deposited by them. This is used to prevent attackers from arbitraging between different stablecoins as the Lulo protocol considers all stablecoins to have equal value at all times.

Because the attacker can set referrer of the user account to itself, the attacker can cause the program to not record withdrawals from the allocations and hence withdraw more tokens than deposited from an allocation. As a result, attacker can arbitrage between the stablecoins.

This also leads to DoS for withdrawals of honest users. The last withdrawals from an allocation will fail because of lack of sufficient tokens in the allocation and honest users cannot withdraw from other allocations.

Exploit Scenario:

1. The Lulo protocol supports USDC and USDT tokens. Current market prices of USDC is 1.005 and USDT price is 0.995.
    - Pool exchange rate is `0.8`
    - USDC allocation exchange rate is `0.75`
    - USDT allocation exchange rate is `0.85`
2. Eve, an attacker, creates a new user account `A` without initializing the `referrer`.
3. Eve deposits 50k USDC into the protected pool. Receives 40k PUSD
    - `Eve PUSD = 40k`
    - `A::depoited_allocations[USDC_index] = 37.5k`
4. Eve deposits 100k USDT into the protected pool. Receives 80k PUSD
    - `PUSD = 120k`
    - `A::deposit_allocations[USDT_index] = 85k`
5. Eve deposits `1` USDC and sets referrer to `A`.
    - `user_account == user_account.referrer == referrer_user_account`
6. Eve calls `withdraw_protected_pool` to withdraw `50k` USDC
    - Pool burns 40k PUSD and transfers 50k USDC to Eve
    - For user_account, `A::deposited_allocations[USDC_index]` is decremented by `37.5k`
    - For referrer account,
        - `UserAccount::referral_fee_unclaimed` is incremented with claimable amount
        - `UserAccount::referral_supply = ...`
        - `UserAccount::referred_amount = 1`
    - Anchor serializes the `user_account` into `A.data` first
    - Anchor serializes the `referrer` into `A` account

- The serialization of `referrer` overwrites the changes performed to `user_account`. The `deposited_allocations[USDC_index]` is not decremented.
- `A::deposited_allocations[USDC_index] = 37.5k`

7. Eve calls `withdraw_protected_pool` to withdraw `50k` USDC
   - Pool burns 40k PUSD and transfers 50k USDC to Eve
   - `A::deposited_allocations[USDC_index]` is not updated and is still `37.5k`
8. Eve calls `withdraw_protected_pool` to withdraw `50k` USDC
   - Pool burns 40k PUSD and transfers 50k USDC to Eve
   - `A::deposited_allocations[USDC_index]` is not updated and is still `37.5k`

Eve deposited `50k` USDC and `100k` USDT at the start of the exploit. Eve received `150k` USDC after the exploit. Eve successfully swapped `100k` USDT for `100k` USDC and has arbitraged at zero cost.

The exploit can be performed in a single transaction and with smaller initial USDC deposit.

Recommendations: Ensure `user_account.referrer != user_account.key()` when initializing the `referrer` address..

Customer's response: Fixed in commit 2b4ec2a.

Fix Review: Fix confirmed.

C-06 Attackers can arbitrage using the `withdraw_all` feature

Severity: **Critical**	Impact: **High**	Likelihood: **High**
Files: /src/state/user.rs	Status: Fixed	

Description: The program allows withdrawing more assets from an allocation than was deposited when the user withdraws all the assets owned by the user account. As a result, an attacker can use the `withdraw_all` feature for arbitrage.

Unset

```
    /// updates the `deposited_allocations` array on the user for this withdrawal,
    /// this check prevents a user from withdrawing an asset that was not deposited
    pub fn withdraw_allocation(
        &mut self,
        allocation_index: usize,
        withdraw_supply: u64,
        withdraw_all: bool,
    ) -> Result<()> {
        let deposited_supply = self.deposited_allocations[allocation_index];
        [...]

        // TODO should still recheck this
        if withdraw_supply > deposited_supply {
            msg!(
                "user withdraw supply {} > deposited_supply {}, withdraw_all={}",
                withdraw_supply,
                deposited_supply,
                withdraw_all
            );

            require!(withdraw_all, Errors::InvalidPoolWithdraw);
        }
```

```
        self.deposited_allocations[allocation_index] =
self.deposited_allocations[allocation_index]
            .checked_sub(withdraw_supply)
            .unwrap_or(0);

        Ok(())
    }
```

Definition of the `UserAccount::withdraw_allocation` function in state/user.rs#L82-L91

Similar to the previous issue C-05, arbitrage causes DoS of withdrawals for honest users.

Exploit Scenario:

1. The Lulo protocol supports USDC and USDT tokens. Current market prices of USDC is 1.005 and USDT price is 0.995.
    - Pool exchange rate is `0.8`
    - USDC allocation exchange rate is `0.75`
    - USDT allocation exchange rate is `0.85`
2. Eve, an attacker, creates a new user account `A` without initializing the `referrer`.
3. Eve deposits 50k USDC into the protected pool. Receives 40k PUSD
    - Eve PUSD = 40k
    - `A::depoited_allocations[USDC_index] = 37.5k`
4. Eve deposits 100k USDT into the protected pool. Receives 80k PUSD
    - PUSD = 120k
    - `A::deposit_allocations[USDT_index] = 85k`
5. Eve calls `withdraw_protected_pool` with USDC as `input_mint` and `withdraw_all = true`
    - Pool burns `120k` PUSD and transfers 150k USDC to Eve
    - The `withdraw_allocation` function is called with
        - `withdraw_all = true`
        - `withdraw_supply = 200k`
        - The function succeeds as `withdraw_all = true`

Eve deposited `50k` USDC, `100k` USDT and received `150k` USDC. Eve swapped low value USDT for USDC at zero cost.

Recommendations: Perform the check even in the case of `withdraw_all = true` and change the implementation of arbitrage prevention mechanism by directly storing `c_tokens` (PUSD or LUSD) minted for each allocation in the user account. The withdrawal instructions can check that the user is not burning more c_tokens for an allocation than they deposited. This restricts users from using c_tokens minted for one allocation to withdraw from different allocations.

Customer's response: Fixed in commit 2b4ec2a.

Fix Review: Fix confirmed.

C-07 Incorrect computation of referral fee share reduces fees earned by protocol

Severity: **Critical**	Impact: **High**	Likelihood: **High**
Files: state/pool/liquidity.rs	Status: Fixed	

Description: The flexlend program computes referral fee share, percentage of total deposits that are referred, as `total_referral_supply / deposited_supply`

```
Unset
    pub fn calculate_referral_fee_share(&self) -> Result<u32> {
        if self.total_referral_supply == 0 || self.deposited_supply == 0 {
            return Ok(0_u32);
        }

        let share = self.total_referral_supply as f64 / self.deposited_supply as f64;

        return Ok(decimal_to_bps(share));
    }
```

Definition of `Allocation::calculated_referral_fee_share` function in liquidity.rs#L429-L437

The `total_referral_supply` are total shares that represent total referred amount and the referral fees earned. The `deposited_supply` are total shares representing total deposited amount and the earnings.

The total shares is a function of the percentage yeild. The percentage yield for `referral fees` is significantly less than the user earnings. As a result, the `total_referral_supply` and the `deposited_supply` do not maintain the `total_referral_supply:deposited_supply` ratio as rewards are accured.

The deposited_supply increments slower than the total_referral_supply because of more percentage yield for deposits. As a result, the `referral_fee_share` will be more than the actual reffered amount to total deposits ratio.

The `referral_fee_share` is used to split the total fees into protocol and referral fees. Because `referral_fee_share` is more than it should be, the protocol receives fewer fees.

Unset

```
pub fn calculate_protocol_fees(
    [...]
    referral_share_bps: u32,
) -> Result<(u64, u64, u64)> {
    [...]
    // referral fees are represented as a % of total fees, but should
    // only be taken based on the ratio of total supply total referred supply
    let referral_fees = total_fees
        .checked_mul(referral_fee_bps.into())
        .ok_or(Errors::OverflowError)?
        .checked_div(BPS_SCALE.into())
        .ok_or(Errors::UnderflowError)?
        .checked_mul(referral_share_bps.into())
        .ok_or(Errors::OverflowError)?
        .checked_div(BPS_SCALE.into())
        .ok_or(Errors::UnderflowError)?;

    // @audit referral_fees = (total_fees * referral_fee) * referral_fee_share
    // More referral_fees result in less protocol fees.
    // TODO check
    let protocol_fees = total_fees - referral_fees;
```

A snippet of the `math::calculate_protocol_fees` function in [math.rs#L291-L302]

Exploit Scenario:

1. Alice is referred by Bob. Alice is the first USDC depositor of the protocol and deposits 2000 USDC
    - Allocation exchange rate = 1

    - `regular_amount = 2000`
    - `deposited_supply = 2000`
    - `total_referred_amount = 2000`
    - `total_referred_supply = 2000`
2. USDC allocation earns `100` USDC rewards
    - total_fees = 10% = 10 USDC
    - referral_fee_share = 2000/2000 = 100%
    - referral_fee = total_fee * referral_fee_bps * referral_fee_share = 10 * 50% * 100% = 5 USDC
    - protocol_fee = total_fee - referral_fee = 5 USDC
    - accumulated_amount = total_rewards - total_fees = 100 - 10 = 90
    - `regular_amount = 2000 + 90 = 2090`
    - `accumulated_referral_fee = 5`
    - `deposited_supply` and `total_referred_supply` are not changed
3. Alice deposits another 1000 USDC
    - Allocation exchange rate = 2000 / 2090
    - deposited_supply = 2000 + (2000/2090) * 1000 = 2956.937799043062
    - total_referred_supply = 2000 + (2000/2005) * 1000 = 2997.506234413965
    - `regular_amount = 2090 + 1000`
    - `total_referred_amount = 3000`

The `referral_fee_share` is `total_referred_supply / deposited_supply = 1.0137 = 10137 bps > 100%`.

4. USDC allocation earns `100` USDC rewards
    - total_fees = 10 USDC
    - referral_fee_share = 10137 bps = 101.37%
    - referral_fee = 10 * 50% * 101.37% = 5.0685 USDC
    - protocol_fee = 4.9315 USDC

Protocol should always receive `>= 50%` of the total fees but because of incorrect `referral_fee_share` computation, protocol received less fees. Overtime, the `referral_fee_share` increases and protocol will lose significant amount of fees.

Recommendations: Compute `referral_fee_share` as `total_referred_amount / (regular_amount + protected_amount)`.

Customer's response: Fixed in PR-1605.

Fix Review: Fix confirmed.

High Severity Issues

H-01 The `withdraw_protocol` function might fail when protocol weight is zero

Severity: **High**	Impact: **High**	Likelihood: **Medium**
Files: instructions/pool/withdraw_protocol.rs	Status: Fixed	

Description: The `withdraw_protocol` rounds-down the collataral shares and does not account for rounding-loss when the withdrawal amount is equal to `Protocol::balance_native`. As a result, the received amount from the protocol might be less than the withdrawal amount leading to error.

Unset

```
pub fn withdraw_protocol<'info, T: WithdrawProtocol<'info>>(
    ctx: Context<'_, '_, '_, 'info, T>,
    protocol: Protocol,
) -> Result<()> {
    [...]
    let amount = ctx.accounts.get_amount(protocol_index)?;

    let mut c_amount = ctx.accounts.liquidity_to_collateral(amount)?;

    [...]
    // TODO here I'm not sure about amount vs c_amount vs native amount
    // Add 1 as due to rounding to make sure we have enough funds withdrawn from the protocol
    if amount < protocol_config.balance_native {
        c_amount = c_amount
            .checked_add(1)
            .ok_or(error!(Errors::OverflowError))?;
    }
```

A snippet of the `withdraw_protocol` instruction handler in [withdraw_protocol.rs#L56-L60]

The computation of `c_amount` depends on the integrated protocol. Kamino protocol, and Solend protocols uses collateral shares and the `liquidity_to_collateral` function computes the shares for the withdrawal amount by rounding-down

```
Unset
    fn liquidity_to_collateral(&self, amount: u64) -> Result<u64> {
        let reserve = &self.reserve.load()?;
        let pool = &self.base_accounts.pool.load()?;

        validate_reserve(pool, reserve, self.base_accounts.input_mint.key())?;

        reserve
            .collateral_exchange_rate()?
            .liquidity_to_collateral(amount)
    }
```

Definition of `KaminoPoolWithdraw::liquidity_to_collateral` in [protocols/pool/kamino_pool.rs#L387-L396]

```
Unset
    pub fn liquidity_to_collateral(&self, liquidity_amount: u64) -> Result<u64> {
        Ok((self.0 * u128::from(liquidity_amount)).to_floor())
    }
```

Definition of `CollateralExchangeRate` in [protocols/pool/kamino_pool.rs#L638-L640]

Because the computation of `c_amount` rounds-down and `c_amount` is not incremented in the case of `amount == Protocol::balance_native`, the `amount_received` might be less than

the `amount`. As a result, the `withdraw_protocol` function fails the `require_eq!` condition preventing the withdrawal

```
Unset
    require!(
        is_gte_close(
            amount_received,
            amount - WITHDRAW_FEE_TOLERANCE,
            10 + WITHDRAW_FEE_TOLERANCE
        ),
    Errors::InvalidProtocolWithdraw
    );
```

snippet of the `withdraw_protocol` instruction handler in [withdraw_protocol.rs#L85-L92]

The `amount` is equal to `Protocol::balance_native` only in the case of `Protocol::weight == 0`.

Exploit Scenario:

1. The Lulo Protocol team decides to remove support for Kamino JLP protocol and sets the protocol weight to zero
2. The Lulo protocol Automation attempts to rebalance the pool and calls `withdraw_protocol`.
    - amount = Protocol::balance_native
3. For the current state of Kamino JLP and Lulo protocol, the value of `c_amount` returned by `KaminoPoolWithdraw::liquidity_to_collateral` is less than `Protocol::balance_native`.
4. The `amount_received` from Kamino protocol is less than `amount = balance_native` and call fails.

Automation cannot withdraw from Kamino JLP protocol

Recommendations: Review the implementation of integrated protocols, Kamino, and Solend protocols, and identify how the shares and amounts are related. Use correct rounding direction in the implementation of `liquidity_to_collateral` function, the value of `c_amount` is equal to requested amount and `c_amount` is less than owned shares. Additionally, consider removing the `if` branch incrementing the `c_amount`.

Customer's response: Fixed in PR-1584.

Fix Review: Fix confirmed.

H-02 Incorrect order of operations leads to incorrect `total_referred_amount` and `total_referral_supply values`

Severity: **High**	Impact: **Medium**	Likelihood: **High**
Files: state/pool/liquidity.rs	Status: Fixed	

Description: The `Allocation::update_referrer_on_withdraw` sets the `UserAccount::referral_supply` and `UserAccount::referred_amount` to `0` before subtracting them from the `total_referred_amount` and `total_referral_supply`

Unset

```
    pub fn update_referrer_on_withdraw(
        &mut self,
        referrer: &mut UserAccount,
        withdraw_amount: u64,
    ) -> Result<()> {
        [...]
        if withdraw_amount >= referrer.referred_amount {
            referrer.referral_supply = 0;
            referrer.referred_amount = 0;

            // @audit these two values should be updated before setting `referrer` values.
            self.total_referred_amount -= referrer.referred_amount;
            self.total_referral_supply -= referrer.referral_supply;
        } else {
```

A snippet of the `Allocation::update_referrer_on_withdraw` function in liquidity.rs#L575-L580

The `total_referred_amount` and the `total_referral_supply` values will be more than the correct values. As a result, referrers will receive less fees as part of the fees will be distributed to these extra shares.

Exploit Scenario:

1. Alice, a user of the Lulo protocol, is referred by Bob.
2. USDC referral exchange rate `0.95`
3. Alice deposits 50k USDC
    - `total_referred_amount += 50k USD`
    - `total_referral_supply += 47500`
4. Alice withdraws everything from the pool by setting `withdraw_all = true`
    - `total_referred_amount` and `total_referral_supply` are not updated
5. `refresh_tvl` is called and the `referral_fees` are added
    - Because `47500` shares are not subtracted, part of the `referral_fees` are distributed to these shares.

Referrers receive less fee and fees earned by the `47500` shares cannot be claimed by anyone.

Recommendations: Reorder the operations: Subtract the values first and then assign them

Customer's response: Fixed in commit 2b4ec2a.

Fix Review: Fix confirmed.

H-03 `Allocation::update_referrer_on_withdraw` function updates `total_referred_amount` and `referred_amount` values incorrectly

Severity: **High**	Impact: **Medium**	Likelihood: **High**
Files: state/pool/liquidity.rs	Status: Fixed	

Description: The `UserAccount::referred_amount`, and the `Allocation::total_referred_amount` values track the amount deposited by the referred users. However, the `update_referrer_on_withdraw` funtion incorrectly subtracts the deposited amount and the claimed referrel fees from these values resulting in these values to be less than the actual values.

Unset

```
    pub fn update_referrer_on_withdraw(
        &mut self,
        referrer: &mut UserAccount,
        withdraw_amount: u64,
    ) -> Result<()> {
        // The `calculate_claimable_referral_rewards` should return all the referral fees
that
        // are owed to this user.
        // i.e share of total referral rewards accumulated from the referred amount time till
now.
        let (claimable_c_amount, claimable_amount) =
            referrer.calculate_claimable_referral_rewards(self)?;

        [...]

        if withdraw_amount >= referrer.referred_amount {
            [...]
        } else {
            [...]
```

```
        let withdraw_amount = withdraw_amount + claimable_amount;

        referrer.referred_amount = referrer
            .referred_amount
            .checked_sub(withdraw_amount)
            .unwrap_or(0);
        [...]
        self.total_referred_amount = self
            .total_referred_amount
            .checked_sub(withdraw_amount)
            .unwrap_or(0);
```

A snippet of the `Allocation::update_referrer_on_withdraw` function in liquidity.rs#L595-L610

Incorrect `referred_amount` results in lesser fees for the `referrer` and the incorrect `total_referred_amount` leads to decreased value for referral shares hence lesser fees for all referrers.

Exploit Scenario:

1. Bob refers Alice to the Lulo protocol
2. Alice deposits 100k USDC in the protocol
    - Bob::referred_amount = 100k USDC
3. After a while, Alice withdraws 75k USDC.
    - Bob earns 1k USDC referral fees
    - Bob::referred_amount = 100k - 75k - 1k = 24k USDC
    - Correct referred_amount = 100k - 75k = 25k USDC

    Bob receives less referral fees.

Recommendations: Update the function to only deduct the withdrawn amount from the `UserAccount::referred_amount` and the `Allocation::total_referred_amount` values.

Customer's response: Fixed in commit 2b4ec2a.

Fix Review: Fix confirmed.

H-04 Incorrect implementation of oracle update staleness check

Severity: **High**	Impact: **Medium**	Likelihood: **High**
Files: state/pool/pool.rs	Status: Fixed	

Description: The `Pool::error_if_stale_refresh` function incorrectly uses elapsed time of pool refresh for checking staleness of the oracle update. Instead, the function should use the elapsed time from the last oracle update

Unset

```
    /// Error if the pool (last_updated) hasn't been refreshed
    /// or if the oracle hasn't been refreshed
    pub fn error_if_stale_refresh(&self, allocation: Allocation, current_ts: u64) ->
Result<()> {
        let elapsed_time = current_ts.saturating_sub(self.last_updated);

        if elapsed_time > self.refresh_stale_max_seconds {
            [...]
        }

        let oracle_elapsed = current_ts.saturating_sub(allocation.oracle_last_updated);

        // @audit this condition should check `oracle_elapsed >
self.oracle_stale_max_seconds` not `elapsed_time`
        if elapsed_time > self.oracle_stale_max_seconds {
            msg!("stale oracle {}", oracle_elapsed);
            return Err(error!(Errors::OraclesNotRefreshed));
        }

        Ok(())
    }
```

Definition of `Pool::error_if_stale_refresh` function in [pool.rs]

The function is used in user deposit and withdraw operations to ensure user has provided correct oracle accounts and the safety checks on oracle prices are performed. Incorrect implementation of this function allows users to bypass the safety checks if the pool was refreshed sufficiently recently.

Exploit Scenario:

1. Eve, an attacker, has deposited 50k USDT into the protocol
2. USDT price fluctuates and is now more than `1.01` USD. Protocol should disallow all withdrawals as a safety mechanism.
3. Eve withdraws her USDC from the protocol because of the incorrect check.

Recommendations: Update the `if` condition checking the oracle update staleness to use `oracle_elapsed` value.

Customer's response: Fixed in commit 2b4ec2a.

Fix Review: Fix confirmed.

Medium Severity Issues

M-01 Smaller `refresh_stale_max_seconds` can DoS the protocol

Severity: **Medium**	Impact: **High**	Likelihood: **Low**
Files: programs/flexlend/src/instructions/pool/refresh_tvl.rs	Status: Fixed	

Description: If `Pool::refresh_stale_max_seconds` is less than 60 seconds, the pool will be in a stale state during the time window between `Pool::last_updated + Pool::refresh_stale_max_seconds` and `Pool::last_updated + 60`. This creates a period where user operations will fail due to the pool being considered stale before the mandatory 60-second refresh interval is complete

Unset

```
#[cfg(feature = "mainnet")]
const MIN_ELAPSED_TIME: u64 = 60;

pub fn refresh_tvl(ctx: Context<RefreshTvl>) -> Result<()> {
    let current_ts = Clock::get()?.unix_timestamp as u64;
    [...]
    let elapsed_time = current_ts
        .checked_sub(pool.last_updated)
        .ok_or(Errors::UnderflowError)?;

    // Shoud wait at least MIN_ELAPSED_TIME seconds to refresh again
    require!(
        pool.last_updated == 0 || elapsed_time >= MIN_ELAPSED_TIME,
        Errors::InvalidRefreshPoolTvl
    );
```

```
    [...]

    pool.last_updated = current_ts;
```

A snippet of the `refresh_tvl_instruction` handler in [refresh_tvl.rs#L29-L33] mandating 60 second refresh interval

Exploit Scenario:

1. Alice, the POOL_ADMIN of the Lulo protocol, sets the `Pool::refresh_stale_max_seconds` to 30 seconds
2. Automation refreshes the protocol at time `t`
3. Users can perform operations in `[t, t+30]` time interval.
4. Bob calls `deposit_pool` instruction at time `t + 45` seconds.
    a. - Call fails as the pool is considered stale and automation cannot refresh the pool until `t + 60`

All user operations fail during `[t + 30, t + 60]` period where `t` is the last refreshed time.

Recommendations: Ensure the `Pool::refresh_stale_max_seconds` is atleast `MIN_ELAPSED_TIME` before setting the value.

Customer's response: Fixed in PR-1609.

Fix Review: Fix confirmed.

M-02 Missing bounds for fee parameters

Severity: **Medium**	Impact: **High**	Likelihood: **Low**
Files: instructions/admin_parameters.rs	Status: Fixed	

Description: The setter instructions for admin parameters lack minimum and maximum bounds allowing for admin mistakenly or intentionally set the parameters to extreme values leads to issues.

The `update_protocol_fees`, `update_protected_interest_share`, and `update_coverage_float` instructions update BPS scaled parameters and does not ensure that arguments are less than maximum BPS value.

The `update_protocol_fees` updates the `Pool::protocol_fee_bps`, and the `Pool::referral_fee_bps` values. Lack of maximum protocol fee check and minimum referral fee check allows admin to set protocol fee to a large value and referral fee to smaller value to gain more in fees.

Similarly, the `update_coverage_float` updates the `Pool::coverage_float_bps`. If the admin mistakenly sets the value to a small value then it can DoS deposits into protected pools.

Exploit Scenario:

1. Alice, the Pool Admin of the Lulo protocol, mistakenly sets `coverage_float_bps` to `1000` (10%).
2. Bob calls `deposit_pool` instruction to 50k USDC deposit into protected pool. Call fails in `protected_deposit_allowed` function.

a. The protocol uses `coverage_float_bps` in `protected_deposit_allowed` to compute the available regular amount. Protocol only considers `10%` of regular amount.

Recommendations:

1. Ensure bps parameters are less than max bps value (`BPS_SCALE`).
2. Determine minimum and maximum values for each of the parameters. Ensure the input parameters statisfy these bounds.

Customer's response: Fixed in PR-1534.

Fix Review: Fix confirmed.

M-03 The `pool_refresh_kamino_alt` instruction lacks security check

Severity: **Medium**	Impact: **Low**	Likelihood: **High**
Files: flexlend/src/lib.rs	Status: Fixed	

Description: The `pool_refresh_kamino_alt` instruction lacks the `security_check` access control and can be called when the protocol is paused for security reasons.

```
Unset
pub fn pool_refresh_kamino_alt<'a, 'b, 'c, 'info>(
        ctx: Context<'_, '_, '_, 'info, KaminoPoolRefreshTVL<'info>>,
    ) -> Result<()> {
        instructions::pool::refresh_protocol_rewards(ctx, Protocol::KaminoAlt)
    }
```

Definition of the `flexlend::pool_refresh_kamino_alt` instruction in [flexlend/src/lib.rs#L532-L536]

Recommendations: Add `security_check` access control to the `pool_refresh_kamino_alt` instruction:

```
Unset
#[access_control(security_check(&ctx.accounts.base_accounts.pool))]
    pub fn pool_refresh_kamino_alt<'a, 'b, 'c, 'info>(
```

Customer's response: Fixed in commit 2b4ec2a.

Fix Review: Fix confirmed.

M-04 `KaminoPoolWithdraw::cpi_withdraw` incorrectly passes pool input token account for collateral account

Severity: **Medium**	Impact: **Low**	Likelihood: **High**
Files: protocols/pool/kamino_pool.rs	Status: Fixed	

Description: The `placeholder_user_destination_collateral` token should be the collateral token account. However, the `cpi_withdraw` function incorrectly sends the `pool_input_token_account` for collateral token account

Unset

```
    fn cpi_withdraw(&self, amount: u64, _remaining_accounts: &[AccountInfo<'info>]) ->
Result<()> {
        [...]
        kamino_cpi::cpi::withdraw_obligation_collateral_and_redeem_reserve_collateral(
            CpiContext::new_with_signer(
                self.kamino_program.to_account_info(),
                WithdrawObligationCollateralAndRedeemReserveCollateral {
                    [...]
                    user_destination_liquidity: self
                        .base_accounts
                        .pool_input_token_account
                        .to_account_info(),
                    placeholder_user_destination_collateral: self
                        .base_accounts
                        .pool_input_token_account
                        .to_account_info(),
```

A snippet of the `KaminoPoolWithdraw::cpi_withdraw` function in [kamino_pool.rs#L424-L427]

The `withdraw_obligation_collateral_and_redeem_reserve_collateral` instruction takes `placeholder_user_destination_collateral` account as an optional account and does not appear to use the account.

Recommendations: Use `KaminoPoolWithdraw::collateral_token_account` for the `placeholder_user_destination_collateral` account.

Customer's response: Fixed in PR-1547.

Fix Review: Fix confirmed.

M-05 The `check_protocol_weights` might prevent admin from setting protocol weight to max possible value

Severity: **Medium**	Impact: **Medium**	Likelihood: **Medium**
Files: utils/checks.rs	Status: Fixed	

Description: The `checks::check_protocol_weights` counts non-active protocols as parents. As a result, `protocol.weight` of protocol with non-active protocol cannot be set to the maximum value

Unset

```
pub fn check_protocol_weights(protocols: Vec<ProtocolConfig>, max_exposure: u32) ->
Result<u32> {
    check_exposure(max_exposure)?;

    let mut sum: u32 = 0;
    for protocol in protocols.iter() {
        let parents: u32 = protocols
            .iter()
            .filter(|p| p.parent_id() == protocol.parent_id())
            .count()
            .try_into()
            .unwrap();

        if protocol.weight > (max_exposure / parents) {
            msg!(
                "err: {} weight {} > max_exposure {} / {} parents",
                protocol,
                protocol.weight,
                max_exposure,
                parents
            );
```

```
            return Err(error!(Errors::MaxExposureExceeded));
        }
```

Exploit Scenario:

1. The Lulo protocol supports 3 pools of the Kamino protocol
2. `max_exposure` is `25%`. Individual protocol weight of Kamino protocols should be `< 25/3 %`.
3. The Lulo protocol removes the support for one of the Kamino protocol
    - sets the protocol weight to `0`
4. The admin should be able to set the protocol weight for the remaining two Kamino protocols upto `25/2 %`.
    - Because the protocol considers inactive protocol as a parent, it restricts the protocol weight to be less than `(25/3)%`

Recommendations: Update the `check_protocol_weights` to only consider active protocols when counting the parents. Additionally, consider reformulating check to compute the combined weight of protocols with same parent_id and ensuring combined weight is less than the max exposure.

Customer's response: Fixed in commit 2b4ec2a.

Fix Review: Fix confirmed.

M-06 Protocol weight might become more than the max exposure

Severity: **Medium**	Impact: **Medium**	Likelihood: **Medium**
Files: flexlend/src/lib.rs	Status: Fixed	

Description: The protocol computes the `max_exposure` used to check the protocol weights as

```
Unset
x = regular_total_liquidity * LUSD_ME + protected_total_liquidity * PUSD_ME
y = regular_total_liquidity + protected_total_liquidity

max_exposure = x / y
```

The protocol ensures that protocol weight is less than max_exposure computed using the pool state when the `set_protocol_weights` instruction is called.

Because `max_exposure` depends on the pool state and pool state changes based on user deposits, and withdraws, new `max_exposure` could be less than the previous `max_exposure`. As a result, the current `protocol.weight` could be more than the `max_exposure` violating the invariant.

Also, the `regular_withdraw_allowed` and the `protected_deposit_allowed` functions do not consider the ratio between the regular tvl and protected tvl when `max_exposure` depends on this ratio. Further investigation is required to determine if and how this impacts the protocol.

regular_withdraw_allowed(amount):

- `updated_regular = Pool::regular_available_amount - amount`

- `if updated_regular == 0 then return Pool::protected_liquidity_amount == 0`
- `minimum_regular = protected_liquidity_amount / ((1 - LUSD_ME) / PUSD_ME)`
- `return updated_regular >= minimum_regular`

protected_deposit_allowed(amount):

- `coverage_ratio = (1 - LUSD_ME) / PUSD_ME`
- `regular_available_amount * coverage_float >= (protected_liquidity_amount + amount) / coverage_ratio`

Exploit Scenario:

1. Assume LUSD_ME = 25% and PUSD_ME = 20%
    - regular_amount = 200
    - protected_amount = 50
2. Admin calls `set_protocol_weights` function and sets `Marginifi::protocol weight` to 24%
    - `max_exposure = 200 * 0.25 + 50 * 20% / 250 => 0.24 = 24%`
3. Alice withdraws 100 tokens from regular pool
    - regular_amount = 100
    - max_exposure = 25 + 10 / 150 => 0.23333 = 23.3%
4. Marginifi has `protocol_weight 24% > max_exposure 23.3%` hence violating the invariant.

Recommendations: Ensuring `PUSD_ME == LUSD_ME` removes the dependence on `regular_tvl:protected_tvl` ratio hence removing the dependence of `max_exposure` on user operations. Further investigation is required to determine other solutions that do not require `PUSD_ME == LUSD_ME`

Customer's response: Fixed in commit 2b4ec2a.

Fix Review: Fix confirmed.

The program checks the current actual weight of each protocol against the `max_exposure` in the `deposit_protocol` instruction ensuring that none of protocols have assets more than the `max_exposure` percentage of total deposits.

The Protocol team should be aware that this check might lead some of the rebalancing attempts to fail under extreme conditions. Rebalancing involves withdrawing from a protocol and depositing in a different protocol. After withdrawal from a protocol, the current weight of the rest of the protocols will increase. If a protocol's weight is near the `max_exposure`, the withdrawal might cause that protocol's weight to cross `max_exposure`. As a result, `deposit_protocol` instruction might fail.

M-07 Attacker can grief `refresh_tvl` operation by causing `calculate_accumulated_split` function to fail

Severity: **Medium**	Impact: **High**	Likelihood: **Low**
Files: utils/math.rs	Status: Fixed	

Description: The `calculate_accumulated_split` function splits the total rewards into protected rewards and the regular rewards. The function computes the values using the `Fraction` type and rounds the values using the `Fraction::to_round()` function

Unset

```
pub fn calculate_accumulated_split(
    [...]
) -> Result<(u64, u64)> {
    [...]
    let regular_liquidity_amount = Fraction::from_num(regular_liquidity_amount);
    let protected_liquidity_amount = Fraction::from_num(protected_liquidity_amount);
    let available_liquidity = Fraction::from_num(available_liquidity);
    let accumulated_net = Fraction::from_num(accumulated_net);

    [...]
    let protected_rewards = accumulated_net
        .checked_mul(protected_liquidity_amount)
        .ok_or(error!(Errors::OverflowError))?
        .checked_div(available_liquidity)
        .ok_or(error!(Errors::OverflowError))?
        .to_round();

    // (regular_liqudity_amount / available_liquidity) * accumulated_net
    let regular_rewards = accumulated_net
        .checked_mul(regular_liquidity_amount)
        .ok_or(error!(Errors::OverflowError))?
        .checked_div(available_liquidity)
```

```
        .ok_or(error!(Errors::OverflowError))?
        .to_round();
```

The function computes the `protected_rewards` and `regular_rewards` using `Fraction` type and `rounds` the value:

```
Unset
if fractional_part < 0.5: floor(result)
if fractional_part >= 0.5: ceil(result)
```

The function has the following assert condition:

```
Unset
    [...]
    require_eq!(
        // @audit rounding errors will cause DoS
        // if both ratios result in `0.5` exactly then this will fail and lead to DoS
        // ex: total rewards is 10, protected is 3.5 before rounding and regular will be 6.5
before rounding.
        // After rounding: protected_rewards will be 4 and regular_rewards will 7. 4 + 7 = 11
!= 10 will cause
        // this assertion to fail.
        protected_rewards + regular_rewards,
        accumulated_net,
        Errors::UnderflowError,
    );
```

A snippet of the `math::calculate_accumulated_split` function in [utils/math.rs#L333-L358]

The `require_eq!` condition fails if the `fractional_part` of the `protected_rewards` is exactly `0.5`. Both the `protected_rewards` and `regular_rewards` will have `0.5` as `fractional_part` and the `to_round` function will round-up both the values. As a result, sum of these values will be `1` more than the `accumulated_net` failing the assert condition.

Because the function is used in the `refresh_tvl` instruction, attackers can grief calls to `refresh_tvl` by forcing the `calculate_accumulated_split` function to fail.

The `protected_liquidity_amount`, `regular_liquidity_amount`, and `available_liquidity` can be incremented or decremented by depositing and withdrawing from the protocol respectively. The exact values can be computed by using the following constraints:

1. (protected_liquidity_amount * accumulated_net) % available_liquidity != 0
    - Ensures the multiple is not divisible and the result has a fractional part
2. (protected_liquidity_amount * accumulated_net * 10) % available_liquidity == 0
    - Ensures the divison result only has only one decimal part `0.1, 0.2, ...`
3. (protected_liquidity_amount * accumulated_net) has factor `5`, even after divison
    - Ensures the decimal part is exactly `0.5`

By fixing the `accumulated_net`, attacker can factor the values and bruteforce the value to deposit/withdraw to grief the protocol.

The `math::calculate_interest_split_share` function is also vulnerable to this issue. The function also rounds `protected_rewards_share`, and `regular_rewards_share` values using `to_round` function and asserts there's no loss from rounding.

Exploit Scenario:

- Protocol has the following state:
    - protected_liquidity = 2536601 USDC
    - available_liqudity = 431223860 USDC
    - accumulated_net = 322575
- Eve, an attacker, front-runs the call to `refresh_tvl` and deposits `10` USDC in the protected pool
    - protected_liquidity = 2536611 USDC
    - available_liqudity = 431223870 USDC
    - accumulated_net = 322575
- The `refresh_tvl` instruction fails with `Underflow` error.

POC to show the function panics for the above state..

```
Unset
    #[test]
    fn test_accumulated_split_rounding() {
        let protected = 2536611 * 10u64.pow(6);
        let available = 431223870 * 10u64.pow(6);
        let accumulated_net = 322575;
        let regular = available - protected;

        // protected_rewards = 1897.5 and regular_rewards = 320677.5
        // both will be rounded up and the function will result in underflow error.
        let _ = calculate_accumulated_split(regular, protected, available, accumulated_net,
0).unwrap();
    }
```

Results of POC:

```
Unset
running 1 test
test utils::math::test::test_accumulated_split_rounding ... FAILED

failures:

---- utils::math::test::test_accumulated_split_rounding stdout ----
called `Result::unwrap()` on an `Err` value: AnchorError(AnchorError { error_name:
"UnderflowError", error_code_number: 6006, error_msg: "UnderflowError", error_origin:
Some(Source(Source { filename: "programs/flexlend/src/utils/math.rs", line: 392 })),
compared_values: Some(Values(("322576", "322575"))) })
```

Recommendations: Update the `calculate_accumulated_split` function to compute `regular_rewards` as the following and remove the `require_eq!` condition

```
Unset
let regular_rewards = accumulated_net - protected_rewards;
```

Similarly, update the `calculate_interest_split_share` function to compute the `regular_rewards_share` as the following:

```
Unset
let regular_rewards_share = (protected_rewards + regular_rewards) - protected_rewards_share
```

Customer's response: Fixed in PR-1608.

Fix Review: Fix confirmed.

M-08 Oracle updates fail because of incorrect assignment leading to DoS

Severity: **Medium**	Impact: **High**	Likelihood: **Low**
Files: utils/price.rs	Status: Fixed	

Description: Incorrectly assigning `write_authority` to `OraclePrice::address` instead of oracle account's key in `get_pyth_pull_oracle` causes oracle validation checks to fail, preventing deposits, withdrawals, and pool refresh operations.

The `price::get_pyth_pull_oracle` function incorrectly initializes the `OraclePrice::address` value to the `write_authority` of the `PriceUpdateV2` account

```
Unset
/// just returns the data, don't validate or do any safety checks,
/// uses ema_price
pub fn get_pyth_pull_oracle(account: &PriceUpdateV2) -> Result<OraclePrice> {
    require!(
        account.verification_level.gte(VerificationLevel::Full),
        Errors::InvalidPythPullOracle
    );
    let oracle = account.price_message;

    Ok(OraclePrice {
        // @audit address should be `PriceUpdateV2` account's key not `write_authority`
        address: account.write_authority,
        ema_price: oracle.ema_price as u64,
        timestamp: oracle.publish_time as u64,
        exponent: oracle.exponent.abs() as u32,
        conf: oracle.conf,
    })
}
```

Definition of `price::get_pyth_pull_oracle` in [price.rs]

The `Allocation::update_oracle_price` function requires that the `OraclePrice::address` is equal to the `Allocation::oracle` value

```
Unset
    pub fn update_oracle_price(
        &mut self,
        oracle: OraclePrice,
        oracle_stale_max: u64,
        current_ts: u64,
    ) -> Result<()> {
        require_keys_eq!(self.oracle, oracle.address);
        [...]
    }
```

A snippet of `Allocation::update_oracle_price` function in [liquidity.rs]

However, the `Allocation::oracle` value is set to the `PriceUpdateV2` account's key.

```
Unset
pub fn add_allocation(ctx: Context<AddAllocation>, deposit_limit: u64) -> Result<()> {
    let mut pool = ctx.accounts.pool.load_mut()?;

    let oracle = get_pyth_pull_oracle(&ctx.accounts.input_mint_pyth_oracle)?;

    let index = pool.add_allocation(
        ctx.accounts.input_mint.key(),
        // @audit-info Allocation::oracle is set to oracle account key
        ctx.accounts.input_mint_pyth_oracle.key(),
        ctx.accounts.input_mint.decimals,
        deposit_limit,
    )?;
```

A snippet of `add_allocation` function in [add_allocation.rs]

As a result, the check in the `Allocation::update_oracle_price` function fails for `PriceUpdateV2` accounts with `write_authority` different from the account's key.

The deposit, withdraw operations and the pool `refresh_tvl` operations attempts to update the oracles using the `Pool::update_oracles` function

```
Unset
    /// Update oracles for all active allocations
    pub fn update_oracles(&mut self, accounts: Vec<AccountInfo>, current_ts: u64) ->
Result<()> {
        for allocation in self.allocations.iter_mut() {
            if allocation.is_active() == false {
                continue;
            }

            let account = accounts
                .iter()
                // @audit allocation.oracle should be equal to account.key
                .find(|account| account.key == &allocation.oracle);

            if account.is_none() {
                continue;
            }

            let pyth_price = deserialize_pyth_pull_oracle(account.unwrap())?;

            // @audit `get_pyth_pull_oracle` incorrectly sets `oracle.address` to
`write_authority`
            let oracle = get_pyth_pull_oracle(&pyth_price)?;

            // @audit `update_oracle_price` requires `oracle.address == allocation.oracle`.
            // i.e Function requires `allocation.oracle == account.key` and
`allocation.oracle == write_authority`.
            allocation.update_oracle_price(oracle, self.oracle_stale_max_seconds,
current_ts)?;
        }

        Ok(())
    }
```

Definition of `Pool::update_oracles` in [pool.rs]

The `Pool::update_oracles` fails because of incorrect implementation of the `get_pyth_pull_oracle` function leading to DoS of user deposit, withdraw and pool refresh operations.

Exploit Scenario:

1. Alice, the `POOL_ADMIN` of the Lulo protocol, adds an allocation for USDC token and provides `PriceUpdateV2` account maintained by Lulo team.
    - The `Allocation::oracle` value is set to feed account address.
    - The `write_authority` is different from account address.
2. Bob, a user of Lulo protocol, calls `deposit_pool` instruction to deposit assets immediately after allocation is added
    - Bob skips the oracle update by not providing the oracle accounts and staleness check passes as the oracle is updated in `add_allocation` call.
3. Bob calls `withdraw_protected_pool` to withdraw deposited assets
    - Bob has to provide oracle accounts and oracle update fails disallowing Bob from withdrawing funds.

Recommendations: Update the `get_pyth_pull_oracle` function to set the `OraclePrice::address` to the oracle account's key.

Customer's response: Fixed in commit 2b4ec2a.

Fix Review: Fix confirmed.

Informational Issues

I-01 Incorrect implementation of `math::adjust_by_bps` function

File: utils/math.rs

Description: The `math::adjust_by_bps` function incorrectly returns `100%` of the amount when the caller requests to adjust by `0%`

```
C/C++
pub fn adjust_by_bps(amount: u64, factor_bps: u16) -> Result<u64> {
    if factor_bps == 0_u16 {
        return Ok(amount);
    }

    amount
        .checked_mul(factor_bps.into())
        .ok_or(error!(Errors::OverflowError))?
        .checked_div(BPS_SCALE.into())
        .ok_or(error!(Errors::UnderflowError))
}
```

The function is only used to adjust the regular amount with `Pool::coverage_float_bps` which is expected to be non-zero and the incorrect implementation is beneficial in this case when the `coverage_float_bps` is mistakenly set to zero.

Recommendation: Remove the `if` condition handling the `0%` case.

Customer's response: Fixed in PR-1548.

Fix Review: Fix confirmed.

I-02 Requirement of unused accounts and writable unmodified accounts

Description: Some of the flexlend program instructions require accounts to be writable when it is not necessary and also take more accounts than needed, increasing the complexity for integrators.

The following accounts are not required to be writable:
- `AddAllocation::admin`
- `AddProtocol::admin`
- `Generic::admin`
- `ClaimReferralFees::owner`
- `WithdrawProtectedPool::owner`
- `WithdrawRegularPool::owner`
- `CompleteWithdraw::owner`
- `SetProtocolWeights::admin`
- `RefreshTvl::fee_payer`

The following accounts are not used:
- `ClaimReferralFees::system_program`, `ClaimReferralFees::rent`
- `WithdrawRegularPool::owner_input_token_account`, `WithdrawRegularPool::pool_input_token_account`
- `CompleteWithdraw::flex_user_lp_token_account`

Recommendation: Remove the `mut` constraints for unmodified accounts and remove the unused accounts from the `Accounts` structs.

Customer's response: Fixed in commit 2b4ec2a.

Fix Review: Fix confirmed.

I-03 Users might not be able to update oracles in `withdraw_protected_pool` instruction

File: instructions/pool/withdraw_protected_pool.rs

Description: The Lulo protocol updates the oracle information by finding the oracle accounts in `Context::remaining_accounts`. However, the `withdraw_protected_pool` forwads the same `remaining_accounts` in the call to `DriftV2::withdraw` instruction. The `DriftV2::withdraw` instruction might fail in the presence of additional accounts in `remaining_accounts` prohibiting the `withdraw_protected_pool` instruction caller from including oracle accounts.

```
Unset
pub fn withdraw_protected_pool<'a, 'b, 'c, 'info>(
    ctx: Context<'a, 'b, 'c, 'info, WithdrawProtectedPool<'info>>,
    amount: u64,
    withdraw_all: bool,
) -> Result<()> {
    [...]
    let amount_received = drift_pool::execute_drift_withdrawal(
        [...]
        ctx.remaining_accounts.to_vec(),
    )?;
    [...]
    {
        // @audit this might not work
        ctx.accounts
            .pool
            .load_mut()?
            .update_oracles(ctx.remaining_accounts.to_vec(), current_ts)?;
    };
```

A snippet of the `withdraw_protected_pool` instruction handler in [withdraw_protected_pool.rs#L49-L66]

```
Unset
    pub fn update_oracles(&mut self, accounts: Vec<AccountInfo>, current_ts: u64) ->
Result<()> {
        for allocation in self.allocations.iter_mut() {
```

```
            if allocation.is_active() == false {
                continue;
            }

            let account = accounts
                .iter()
                .find(|account| account.key == &allocation.oracle);

            if account.is_none() {
                continue;
            }
```

A snippet of the `Pool::update_oracles` function in [pool.rs#L240-L252]

The `DriftV2::withdraw` function requires the `remaining_accounts` to have the following structure:

```
Unset
oracle_1, ..., oracle_i,
spot_market_1, ..., spot_market_j,
perp_market_1, ..., perp_market_k,
Optional<mint>
```

The `remaining_accounts` should not contain any other accounts in between. However, the `remaining_accounts` can contain additional accounts at the end after `mint` if and only if `mint` is not `None`.

Because the `mint` account is not significant for the call in the current **implementation** of Drift protocol, the caller of the `withdraw_protected_pool` can pass the `mint` account. As a result, the caller of the `withdraw_protected_pool` instruction can include the oracle accounts at the end and update the oracles.

However, if the `Drift` protocol implementation were to change and the new implementation further restricts `remaining_accounts`, users might not be able to update oracles in the `withdraw_protected_pool` instruction.

Recommendation: Update the implementation of `Pool::update_oracles` to follow the similar pattern to the Drift protocol. Require that all the oracle accounts are included at the start of the `remaining_accounts` ensuring that the accounts for the Lulo protocol are separated from the accounts passed to the integrating protocol.

```
Unset
update_oracles(self, remaining_accounts_peekable_iter, current_ts):

    while let Some(account_info) = account_info_iter.peek() {
        if account_info.owner != PYTH_RECEIVER_PROGRAM_ID {
            break;
        }
        // find the allocation with `allocation.oracle == account.key`. Break if not found
        ...
    }
```

Customer's response: Not confirmed. No fix is needed for this one, the remaining accounts are configured on the SDK side.

I-04 The `initiate_regular_withdraw` and `withdraw_protected_pool` round-down the shares burned

File: instructions/pool/initiate_regular_withdraw.rs
File: instructions/pool/withdraw_protected_pool.rs

Description: The `initiate_regular_withdraw` and the `withdraw_protected_pool` instructions use `Pool::convert_to_c_tokens` function to compute the amount of shares to burn for the withdrawal amount. The `convert_to_c_tokens` function rounds-down the shares resulting in the value of shares burned to be less than the withdrawal amount hence benefitting the user instead of the protocol.

```
Unset
    pub fn convert_to_c_tokens(
        &self,
        _mint: Pubkey,
        pool_type: PoolType,
        amount: u64,
        current_ts: u64,
    ) -> Result<u64> {
        let c_amount = self
            .get_exchange_rate(pool_type, current_ts)?
            .liquidity_to_collateral(amount);

        msg!("amount {}, c_amount {}", amount, c_amount);

        Ok(c_amount)
    }
```

Definition of the `Pool::convert_to_c_tokens` function in [pool.rs#L217-L231]

```
Unset
    pub fn liquidity_to_collateral(&self, liquidity_amount: u64) -> u64 {
        (self.0 * u128::from(liquidity_amount)).to_floor()
    }
```

Definition of the `ExchangeRate::liquidity_to_collateral` function in liquidity.rs#L636-L638

Recommendation: Add a function to Pool that rounds-up the amount of shares and update the `initiate_regular_withdraw`, and the `withdraw_protected_pool` instruction handlers to use the new function for computing the burned shares.

Customer's response: Fixed in commit 2b4ec2a.

Fix Review: Fix confirmed.

I-05 Users earn more by setting themselves as a referrer

Description: The `referral_fees` come from the protocol revenue and users earnings are same irrespective of whether the `referrer` for their account is initialized. As a result, a user can set referrer to an account owned by them and earn additional referral fees.

Exploit Scenario:

1. Eve, an attacker, creates two user accounts `A` and `B` controlled by different private keys.
2. Eve sets `B` as the referrer of `A`.
3. Eve uses account `A` to interact with the protocol and `B` receives the referral fees for `A`.
4. Eve receives earnings from deposits in `A`.
5. Eve also earns referral fees from `B`, Protocol receives lesser revenue.

Recommendations: Consider only supporting registered referrers or mandate user KYC ensuring a user can have only one account. Investigate benefits of these approaches compared to loss from this issue.

Customer's response: Confirmed. Risk accepted.

I-06 Protocol::try_into panics if protocol name is more than 8 bytes

File: src/protocols/mod.rs

Description: The `Protocol::TryInto<[u8; 8]>` implementation attempts to copy slice of size `10` bytes into a slice of size `8` bytes if length of the protocol name is more than 8 bytes. The `copy_from_slice` function panics if src and dst slices are of different sizes. As a result, the `try_into` function will panic if the protocol name is more than 8 bytes.

```
Unset
fn try_into(self) -> Result<[u8; 8], Self::Error> {
        let str = match self {
            Protocol::Drift => "drift",
            [...]
        };

        let mut protocol = [0u8; 8];
        // protocol.copy_from_slice(str.as_bytes());

        // Convert input string to bytes
        let str_bytes = str.as_bytes();
        let len = str_bytes.len();

        if len > protocol.len() {
            // Truncate the input string if it's longer than `protocol.length`
            protocol.copy_from_slice(&str_bytes[..10]);
        }
```

Implementation of `Protocol::TryInto<[u8;8]>` in protocols/mod.rs#L66-L86

All the current supported protocol names are less than 8 bytes resulting in no impact for the current implementation.

Recommendation: Change the flexlend/src/protocols/mod.rs#L85 line to

```
Unset
 protocol.copy_from_slice(&str_bytes[..8]);
```

Customer's response: Fixed in commit 2b4ec2a.

Fix Review: Fix confirmed.

I-07 Missing program ownership check on user_account in deposit_pool instruction

File: instructions/pool/deposit_pool.rs

Description: The `deposit_pool` instruction defines `user_account` as an `UncheckedAccount` and does not perform explicit checks on the address, and the program ownership of the `user_account` allowing attackers to pass in any account for `user_account`. However, the `deposit_pool` instruction anchor deserializes and modifies the `user_account` implicitly validating the account type and the program ownership

```
Unset
pub fn deposit_pool(ctx: Context<DepositPool>, amount: u64, pool_type: PoolType) ->
Result<()> {
    [...]
    // manually deserialize UserAccount
    let user =
        &mut UserAccount::try_deserialize(&mut
&ctx.accounts.user_account.data.borrow_mut()[..])?;

    pool.deposit_user_liquidity(user, pool_type, mint, amount)?;
    [...]
    // write user_account changes
    user.try_serialize(&mut *ctx.accounts.user_account.data.borrow_mut())?;
    [...]
    Ok(())
}
```

A snippet of the `deposit_pool` instruction handler in deposit_pool.rs#L119-L123

The modifications performed to the `user_account` by the `deposit_pool` can be made to be null and hence removing the implicit program ownership check. However, this requires depositing very small amounts. The current implementation of the program has a minimum deposit of 1 USD making this bug unexploitable.

Recommendation: Add explicit PDA address check on the `user_account`.

Customer's response: Fixed in commit 2b4ec2a.

Fix Review: Fix confirmed.

I-08 Users can bypass minimum transfer check by making post-withdrawal balance of less than 1 USD

Description: The Lulo protocol prevents deposits and withdraws of value less than 1 USD as a safety measure for rounding issues. However, it does not ensure that the user cannot have less than 1 USD value in the account after withdrawal. As a result, the user can perform first withdrawal to have remaining balance of less than 1 USD and then make a withdrawal of the small amount using `withdraw_all = true`.

Exploit Scenario:

1. Bob, a user of the Lulo protocol, deposits 10k USDC and gets 9k PUSD
2. Bob withdraws 9999.998888 USD
    - Bob account has a balance of `0.001112` < 1 USD
3. Bob calls the `withdraw_protected_pool` instruction with `withdraw_all = true`
    - Protocol performs a withdrawal of `0.001112` < 1 USD

    Bob was able to perform a withdrawal of value < 1 USD

Recommendation: . Update user withdrawal instructions to either reject withdrawals that result in a remaining balance of less than 1 USD or consider such withdrawals as `withdraw_all = true`. Additionally, document the behavior for users and integrators.

Customer's response: Fixed in commit 2b4ec2a.

Fix Review: Fix confirmed.

Program withdraws total `max_withdrawable` if requested amount is close to `max_withdrawable` with a tolerance of $1 USD.

I-09 Incorrect design of the arbitrage prevention mechanism

Description: Pool prevents users from arbitraging between stablecoins by tracking user deposits into each allocation and only allowing withdrawals from the deposited allocation(s). Each allocation has an allocation-specific exchange rate representing the user deposits of the allocation token and the rewards earned by that allocation. When a user withdraws a certain amount in an input token, the user must have the input token allocation's shares of the same value.

This leads to issues when the user withdrawal amount is more than the value of allocation shares. Its possible as user's earnings are tracked at the Pool level and the amount a user earns is average of earnings by all allocations. Because Pool APY is average of all allocations, Pool APY might be more than the some of the individual allocations. For such allocations, user withdrawal amount will be more than the value of allocation specific shares.

For example, Pool supports USDC and USDT tokens and USDC has 10% APY, USDT has 8% APY. User who deposits 1000 USDT will earn 9% and should be able to withdraw `1000 + 9% of 1000 = 1090`. However, the USDT allocation earnings are only `8%`. As a result, the user's USDT allocation shares will only be worth `1000 + 8% of 1000 = 1080`. The user will not be able to withdraw `1085` USDT.

The flexlend program solves this issue by allowing users to bypass the check when `withdraw_all = true`. However, this approach allows attackers to arbitrage as detailed in issue C-06 (Attackers can arbitrage using the withdraw_all feature). Moreover, this design also forces users with deposits into multiple allocations to withdraw everything even when the user wishes to only withdraw everything from one token.

Exploit Scenario:

1. Pool supports USDS and USDT tokens. USDS allocation earns 10% APY and USDT's allocation earns 8% APY. Pool APY = 9%
2. Alice deposits 10k USDS and 10k USDT tokens
3. After a while, Alice pool shares are of value `20k + 9% of 20k = 21800 USD`

4. Alice intends to not hold any USDT tokens, and attempts to withdraw half of total value owned by Alice under the assumption she deposited same amount in both tokens hence should receive exactly half.
5. Alice attempts to withdraw `21800 / 2 = 10900` from USDT allocation.
    - Alice withdrawal fails, her USDT allocation shares have value of `10k + 8% of 10k = 10800` USDT

    Alice can only withdraw `10800` USD in USDT token or use `withdraw_all` and withdraw `21800` USD in USDT.

Recommendation: . Remove the use of allocation-specific exchange rate and track the PUSD, and the LUSD shares minted for each allocation separately in the user account.

Customer's response: Fixed in commit 2b4ec2a.

Fix Review: Fix confirmed.

Suggested Code Improvements

- Add a call to `Pool::verify_balances` in the `deposit_pool` and the `withdraw_protocol_pool` instructions. Calling `verify_balances` protects from unidentified issues by ensuring protocol does not process an user operation which results in an invalid state.

- Re-evaluate the need for `Allocation::_padding1` field. The `Allocation` struct defintion should maintain the required alignment without the `_padding1` field.

- Replace the expression below with an assignment to `0`.

```
Unset
        self.accumulated_protocol_fees = self
            .accumulated_protocol_fees
            .checked_sub(self.accumulated_protocol_fees)
            .ok_or(error!(Errors::OverflowError))?;
```

A snippet of the `Allocation::reset_protocol_fees` function in [liquidity.rs#L406-L409]

- Update the `Pool::seeds` function to use `Pool::pool_id` to compute the seeds. Using the `pool_id` makes it easy to allow for more pools by just removing the check from `initialize_pool` function.

```
Unset
    pub fn seeds(&self) -> [&[u8]; 3] {
        [&b"pool"[..], &[0], self.bump.as_ref()]
    }
```

Definition of the `Pool::seeds` function in [pool.rs#L102-L104]

- Use constant variables for seed string literals. Using constant variables protects from user-errors.

- Remove the redundant `pool.last_updated == 0` condition in the Figure A.13. The `last_updated` is initialized in the `initialize_pool` instruction and is always non-zero.

```
Unset
    // Shoud wait at least MIN_ELAPSED_TIME seconds to refresh again
    require!(
        pool.last_updated == 0 || elapsed_time >= MIN_ELAPSED_TIME,
        Errors::InvalidRefreshPoolTvl
    );
```

A snippet of the `refresh_tvl` instruction handler in [refresh_tvl.rs#L29-L33]

- Add a `fee_payer` account to `InitializePool` accounts. Introducing a `fee_payer` account allows fees to be paid from a throwaway account, eliminating the need for the `POOL_ADMIN` to sign calls to the metadata program. This approach improves security by avoiding the use of a privileged account for signing CPI calls, reducing impact from arbitrary-cpi issues.

- Fix the token program used for PUSD and LUSD lp tokens. Fixing the program simplifies the constraints for the `lp_token_program` account.

- Remove the unused `PendingWithdraw::withdraw_all` field

- Use `6` decimals for PUSD and LUSD tokens. Using `6` decimals provides better UX as the exchange rate for lp tokens starts at `1:1` and all input tokens have `6` decimals.

- Refactor the `initiate_regular_withdraw` instruction handler. Rewriting simplifies the code and removes the need to handle dust amounts. The `withdraw_protected_pool` instruction handler can be refactored similarly..

```
Unset
    let amount = if withdraw_all {
        pool.get_exchange_rate(PoolType::Regular, current_ts)?
            .collateral_to_liquidity(user_lp_token_amount)?
    } else {
        amount
    };
```

```
    let mut c_amount = pool.convert_to_c_tokens(mint, PoolType::Regular, amount,
current_ts)?;

    let exchange_rate = pool.get_exchange_rate(PoolType::Regular, current_ts)?;

    // withdraw dust
    if c_amount + 1 == user_lp_token_amount {
        c_amount += 1;
    }

    if withdraw_all {
        require_eq!(user_lp_token_amount, c_amount, Errors::InvalidPoolWithdraw);
    } else {
        require_gte!(user_lp_token_amount, c_amount, Errors::InvalidPoolWithdraw);
    }
```

A snippet of the `initiate_regular_withdraw` instruction handler in [initiate_regular_withdraw.rs#L42-L62]

```
Unset
    let (amount, c_amount) = if withdraw_all {
        let withdrawn_amount = pool.get_exchange_rate(PoolType::Regular, current_ts)?
                .collateral_to_liquidity(user_lp_token_amount)?;
        (withdrawn_amount, user_lp_token_amount)
    } else {
        let c_amount = pool.convert_to_c_tokens(mint, PoolType::Regular, amount,
current_ts)?;

        require_gte!(user_lp_token_amount, c_amount, Errors::InvalidPoolWithdraw);

        (amount, c_amount)
    };

    let exchange_rate = pool.get_exchange_rate(PoolType::Regular, current_ts)?;
```

- Remove the redundant condition with identical comparison `amount > deposited_drift`.

```
Unset
    pub fn add_pending_withdraw(&mut self, amount: u64) -> Result<u64> {
        // total pending withdraws can't be more than we have deposited into Drift
        let deposited_drift = self.get_protocol(Protocol::Drift)?.balance_native;

        if amount > deposited_drift || amount > deposited_drift {
```

A snippet of the `Allocation::add_pending_withdraw` function with the redundant condition in [liquidity.rs#L304]

- Remove the declaration of `mint` variable [here]. This is a no-op as the `mint` variable is redeclared.

- Remove the unused `owner_input_token_account`, and `pool_input_token_account` accounts for the `initiate_regular_withdraw` instruction.

- Re-evaluate the `require_eq` condition in the `Pool::verify_balances` function. The condition is always `true` irrespective of the underlying state and the implementation

```
Unset
    pub fn verify_balances(&self) -> Result<()> {
        require_eq!(
            self.regular_liquidity_amount()? - self.pending_regular_withdrawals()?
                + self.protected_liquidity_amount()?,
            self.available_liquidity()?
        );
```

A snippet of the `Pool::verify_balances` function with the tautological condition in [pool.rs#L502-L506]

```
Unset
    pub fn available_liquidity(&self) -> Result<u64> {
        Ok(self.protected_liquidity_amount()? + self.regular_available_amount()?)
    }
```

Definition of the `Pool::available_liquidity` function in [pool.rs#L295-L297]

```
Unset
    pub fn regular_available_amount(&self) -> Result<u64> {
        Ok(self.regular_liquidity_amount()? - self.pending_regular_withdrawals()?)
    }
```

Definition of the `Pool::regular_available_amount` function in [pool.rs#L282-L284]

- Update the `math::calculate_rewards` function to iterate over only active allocations (`existing_allocations`).

```
Unset
/// requires all protocols in all allocations are stale
pub fn calculate_rewards(
    pool: &Pool,
    current_ts: u64,
) -> Result<Vec<(usize, u64, u64, u64, u64, u64, u64)>> {
    let allocations = pool.allocations.iter();
```

- Remove the redundant `require_eq!` condition in the `math::calculate_protocol_fees` function:

```
Unset
pub fn calculate_protocol_fees(
    accumulated: u64,
    protocol_fee_bps: u16,
    referral_fee_bps: u16,
    referral_share_bps: u32,
) -> Result<(u64, u64, u64)> {
    // TODO check
```

```
    let protocol_fees = total_fees - referral_fees;

    require_eq!(total_fees, referral_fees + protocol_fees);
```

A snippet of the `math::calculate_protocol_fees` function in [utils/math.rs#L302-L304]

- Use Anchor `InitSpace` [macro] to compute size of the `#[account]` structs instead of calculating manually. Using the macro simplifies the code and makes it less error-prone.

```
Unset
#[account]
#[derive(Debug, Default)]
pub struct UserAccount {
    [...]
}
// 328
utils::size!(
    UserAccount,
    1 + 7 + 32 + 32 + 32 + (3 * 32) + (8 * POOL_MAX_ALLOCATIONS) + 8 + 8 + 32 + 8 + 8 + 8 +
(8 * 2)
);
```

Current implementation computing the size manually in [state/user.rs#L40-L43].

```
Unset
#[account]
#[derive(Debug, Default, InitSpace)]
pub struct UserAccount {
    [...]
}
// 328
utils::size!(
    UserAccount,
    UserAccount::INIT_SPACE
);
```

Modified version of Figure A.8 using the `InitSpace` macro.

- Use `Pool::seeds` to compute seeds consistently across the program.

- Initialize `pool_input_token_account` for `input_mint` in `AddAllocation` instruction to eliminate the need for `init_if_needed` account initialization in other instructions.

- Update the comment below to accurately reflect the 50% referral fee.

```
Unset
        self.referral_fee_bps = decimal_to_bps(0.50).try_into().unwrap(); // 10% protocol fee
```

Incorrect comment in [pool.rs#L147]

Disclaimer

Even though we hope this information is helpful, we provide no warranty of any kind, explicit or implied. The contents of this report should not be construed as a complete guarantee that the contract is secure in all dimensions. In no event shall Certora or any of its employees be liable for any claim, damages, or other liability, whether in an action of contract, tort, or otherwise, arising from, out of, or in connection with the results reported here.

About Certora

Certora is a Web3 security company that provides industry-leading formal verification tools and smart contract audits. Certora's flagship security product, Certora Prover, is a unique SaaS product that automatically locates even the most rare & hard-to-find bugs on your smart contracts or mathematically proves their absence. The Certora Prover plugs into your standard deployment pipeline. It is helpful for smart contract developers and security researchers during auditing and bug bounties.

Certora also provides services such as auditing, formal verification projects, and incident response.