Sec3™

Security Assessment Report

Marginfi V2 PRs 411, 424 and 427

December 01, 2025

Summary

The Sec3 team was engaged to conduct a thorough security analysis of the Marginfi V2 PRs 411, 424 and 427.

The artifact of the audit was the source code of the following programs, excluding tests, in https://github.com/mrgnlabs/marginfi-v2.

The initial audit focused on the following versions and revealed 11 issues or questions.

#	Task	Type	Commit
P1	PR411: Multi-point curves	Solana	`d82019e` → `117fc5f`
P2	PR424: Fixed oracle setup	Solana	`36a5b12` → `f2096c0`
P3	PR427: Forced deleverage	Solana	`da56c55` → `dfc4b03`
P4	Out of scope	Solana	`dfc4b03`

This report provides a detailed description of the findings and their respective resolutions.

Table of Contents

Result Overview

Issue	Impact	Status
PR411: MULTI-POINT CURVES		
`[P1-L-01]` Missing validation in interest rate configuration	Low	Open
`[P1-I-01]` Rounded `util` 0 with non-zero `rate` breaks validation	Info	Open
`[P1-I-02]` Legacy max rate above `10` is clamped to `10`	Info	Open
`[P1-I-03]` Inconsistent comments	Info	Open
PR424: FIXED ORACLE SETUP		
`[P2-L-01]` Allowing a fixed price of zero introduces potential vulnerabilities	Low	Open
`[P2-I-01]` Inconsistent error value	Info	Open
`[P2-I-02]` Remove outdated functions for calculating remaining accounts	Info	Open
PR427: FORCED DELEVERAGE		
`[P3-M-01]` Dust liabilities may block `TOKENLESS_REPAYMENTS_COMPLETE`	Medium	Open
`[P3-L-01]` Deposits allowed when `TOKENLESS_REPAYMENTS_ALLOWED` is `true`	Low	Open
`[P3-I-01]` Deleverage withdrawal limits bypass by changing the authority	Info	Open
OUT OF SCOPE		
`[P4-L-01]` Incorrect flag check disables program fees	Low	Open

Findings in Detail

PR411: MULTI-POINT CURVES

[P1-L-01] Missing validation in interest rate configuration

> *Identified in commit `76dd728`.*

The `lending_pool_configure_bank_interest_only` instruction is used to modify the interest rate configuration. It calls `bank.config.interest_rate_config.update` to apply the changes.

```
/* programs/marginfi/src/instructions/marginfi_group/configure_bank_lite.rs */
pub fn lending_pool_configure_bank_interest_only(
    ctx: Context<LendingPoolConfigureBankInterestOnly>,
    interest_rate_config: InterestRateConfigOpt,
) -> MarginfiResult {
    let mut bank = ctx.accounts.bank.load_mut()?;

    // If settings are frozen, interest rates can't update.
    if bank.get_flag(FREEZE_SETTINGS) {
        msg!("WARN: Bank settings frozen, did nothing.");
    } else {
        bank.config
            .interest_rate_config
            .update(&interest_rate_config);
        msg!("Bank configured!");
    }

    Ok(())
}

/* programs/marginfi/src/state/interest_rate.rs */
fn update(&mut self, ir_config: &InterestRateConfigOpt) {
    set_if_some!(self.zero_util_rate, ir_config.zero_util_rate);
    set_if_some!(self.hundred_util_rate, ir_config.hundred_util_rate);
    set_if_some!(self.points, ir_config.points);
    // Note: If we ever support another curve type, this will become configurable.
    self.curve_type = INTEREST_CURVE_SEVEN_POINT;
}
```

However, this function does not call `validate` to verify the new configuration (such as the ordering requirements for `points`). This omission can lead to incorrect interest rate calculations later.

It is recommended to add a `bank.config.interest_rate_config.validate` check within `lending_pool_configure_bank_interest_only`.

Resolution

To be completed after reviewing the 2nd version with fixes for the reported issues.

PR411: MULTI-POINT CURVES

[P1-I-01] Rounded `util` 0 with non-zero `rate` breaks validation

Identified in commit `117fc5f`.

In `migrate_curve.rs`, the `migrate_curve` function migrates the legacy interest curve to seven seven-point curve. Specifically, it builds `RatePoint { util: centi_to_u32(util), rate: milli_to_u32(rate) }`.

However, if `centi_to_u32` rounds a very small optimal utilization rate down to `0`, while the corresponding plateau rate is non-zero, the resulting point has `util == 0` and `rate > 0`.

This violates the requirements of `validate_seven_point`, which mandates that padding points (`util == 0`) must have a rate of `0`. As a result, `bank.config.validate()` fails after migration, causing the migration to revert.

```rust
/* programs/marginfi/src/instructions/marginfi_group/migrate_curve.rs */
let util: I80F48 = irc.optimal_utilization_rate.into();
let rate: I80F48 = irc.plateau_interest_rate.into();
let point = RatePoint {
    util: centi_to_u32(util),
    rate: milli_to_u32(rate),
};

/* programs/marginfi/src/state/interest_rate.rs */
for (i, p) in self.points.iter().enumerate() {
    if p.util == 0 {
        // Padding: must be (0,0); once seen, all following must be padding as well.
        if p.rate != 0 {
            msg!("Expected padding (zero rate) at {:?}", i);
            return err!(MarginfiError::InvalidConfig);
        }
        seen_padding = true;
    }
```

Test case:

```rust
fn tiny_util_to_zero_test() {
    use marginfi_type_crate::types::{centi_to_u32, make_points, milli_to_u32, InterestRateConfig, RatePoint,
    ↪  INTEREST_CURVE_SEVEN_POINT};

    let tiny_util: I80F48 = I80F48::from_num(1e-10);
    let util_u32: u32 = centi_to_u32(tiny_util);
    assert_eq!(util_u32, 0, "tiny non-zero util should quantize to zero");
    let mut irc_invalid = InterestRateConfig {
        zero_util_rate: 0,
        hundred_util_rate: milli_to_u32(I80F48::from_num(0.20)),
        points: make_points(&[RatePoint::new(
            util_u32,
            milli_to_u32(I80F48::from_num(0.05)),
        )]),
        curve_type: INTEREST_CURVE_SEVEN_POINT,
        ..Default::default()
    };
    assert!(irc_invalid.validate_seven_point().is_err());
}
```

Test result:

```
running 1 test
Expected padding (zero rate) at 0
test state::interest_rate::tests::tiny_util_to_zero_test ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 105 filtered out; finished in 0.00s
```

Although the probability of `optimal_utilization_rate` being small enough to quantize to `util == 0` is extremely low, it is recommended to include a safeguard that sets the rate to `0` whenever util becomes `0`. This ensures that the migration process does not fail and that the legacy curve can be successfully converted into the new curve format.

Resolution

To be completed after reviewing the 2nd version with fixes for the reported issues.

PR411: MULTI-POINT CURVES

[P1-I-02] Legacy max rate above 10 is clamped to 10

Identified in commit 117fc5f.

In migrate_curve.rs, the migrate_curve function derives the new hundred_util_rate from the legacy max_interest_rate via milli_to_u32.

The u32 encoding used for rates in the seven-point curve maps the range 0-10 linearly and clamps any value above this range.

Consequently, if the legacy max_interest_rate exceeds 10, the conversion silently saturates to the maximum representable value, corresponding to 10.

```
/* programs/marginfi/src/instructions/marginfi_group/migrate_curve.rs */
let hundred_rate: I80F48 = irc.max_interest_rate.into();
irc.hundred_util_rate = milli_to_u32(hundred_rate);

/* type-crate/src/types/interest_rate.rs */
/// Useful when converting an I80F48 (e.g. apr) into a percentage from 0-1000. Clamps to 1000% if
/// exceeding that amount. Clamps to zero for negative inputs.
pub fn milli_to_u32(value: I80F48) -> u32 {
    let max_percent: I80F48 = I80F48::from_num(10.0); // 1000%
    let clamped: I80F48 = value.min(max_percent).max(I80F48::ZERO);
    let ratio: I80F48 = clamped / max_percent;
    (ratio * I80F48::from_num(u32::MAX)).to_num::<u32>()
}
```

However, the original validate_legacy function does not impose an upper bound on max_interest_rate. As a result, this conversion caps the migrated maximum rate at 10 and discards information from legacy configurations that permit a higher max_interest_rate.

```
/* programs/marginfi/src/state/interest_rate.rs */
fn validate_legacy(&self) -> MarginfiResult {
    let optimal_ur: I80F48 = self.optimal_utilization_rate.into();
    let plateau_ir: I80F48 = self.plateau_interest_rate.into();
    let max_ir: I80F48 = self.max_interest_rate.into();

    check!(
        optimal_ur > I80F48::ZERO && optimal_ur < I80F48::ONE,
        MarginfiError::InvalidConfig
    );
    check!(plateau_ir > I80F48::ZERO, MarginfiError::InvalidConfig);
    check!(max_ir > I80F48::ZERO, MarginfiError::InvalidConfig);
    check!(plateau_ir < max_ir, MarginfiError::InvalidConfig);

    Ok(())
}
```

It is recommended to enforce a precondition that rejects legacy configurations with `max_interest_rate > 10` and returns a clear error, preventing migration from silently truncating values beyond the supported range.

Resolution

To be completed after reviewing the 2nd version with fixes for the reported issues.

PR411: MULTI-POINT CURVES

[P1-I-03] Inconsistent comments

Identified in commit `117fc5f`.

The function `centi_to_u32` is used to convert an `I80F48` utilization rate into a percentage over the range `0100`, clamping above to `100%` and below to `0`.

However, the inline comment beside `max_percent` incorrectly states `1000%` even though `max_percent` is `1.0`.

```
/* type-crate/src/types/interest_rate.rs */
/// Useful when converting an I80F48 (e.g. utilization rate) into a percentage from 0-100. Clamps to
/// 100% if exceeding that amount. Clamps to zero for negative inputs.
pub fn centi_to_u32(value: I80F48) -> u32 {
    let max_percent: I80F48 = I80F48::from_num(1.0); // 1000%
    let clamped: I80F48 = value.min(max_percent).max(I80F48::ZERO);
    let ratio: I80F48 = clamped / max_percent;
    (ratio * I80F48::from_num(u32::MAX)).to_num::<u32>()
}
```

Resolution

To be completed after reviewing the 2nd version with fixes for the reported issues.

PR424: FIXED ORACLE SETUP

[P2-L-01] Allowing a fixed price of zero introduces potential vulnerabilities

Identified in commit `ffdf6c6`.

The current configuration allows the fixed price to be set to zero. However, because the `RiskEngine` accepts a price of zero, a bank configured with `fixed_price = 0` may have its health erroneously overstated.

```
/* programs/marginfi/src/instructions/marginfi_group/set_fixed_oracle_price.rs */
bank.config.oracle_keys[0] = Pubkey::default();

let price_i80: I80F48 = price.into();
check!(
    price_i80 >= I80F48::ZERO,
    MarginfiError::FixedOraclePriceNegative
);

bank.config.fixed_price = price;
```

Three functions are directly affected:

- `lending_account_borrow`: Does not enforce `price > 0`. If the debt asset has a fixed price of zero, the account's health score is artificially inflated, allowing borrowing until the borrowing limit is reached.
- `lending_account_withdraw`: Outside of liquidation, it does not check `price > 0`. An inflated health score may allow a user to withdraw other collateral improperly.
- `lending_pool_handle_bankruptcy`: `check_account_bankrupt` compares asset and liability values based on equity. A zero-price liability makes the bank appear solvent, preventing the bankruptcy handling flow from executing.

It is recommended to disallow setting `fixed_price = 0` for any bank whose `borrow_limit` is non-zero.

Resolution

To be completed after reviewing the 2nd version with fixes for the reported issues.

PR424: FIXED ORACLE SETUP

[P2-I-01] Inconsistent error value

Identified in commit `ffdf6c6`.

In `lending_pool_set_fixed_oracle_price`, the `MarginfiError::StakedPythPushWrongAccountOwner` is returned when the bank's `asset_tag` is `ASSET_TAG_STAKED`.

```
/* programs/marginfi/src/instructions/marginfi_group/set_fixed_oracle_price.rs */
if bank.config.asset_tag == ASSET_TAG_STAKED {
    msg!("Staked banks cannot set a fixed price");
    return err!(MarginfiError::StakedPythPushWrongAccountOwner);
}
```

It is recommended to replace it with a more proper error.

Resolution

To be completed after reviewing the 2nd version with fixes for the reported issues.

PR424: FIXED ORACLE SETUP

[P2-I-02] Remove outdated functions for calculating remaining accounts

Identified in commit `ffdf6c6`.

With the introduction of `OracleSetup::Fixed`, determining the number of remaining accounts required for a token's oracle now depends on `config.oracle_setup` in the bank account. It is impossible to determine this relying solely on the `balance`.

Therefore, the following two functions are obsolete, and using them will return an incorrect number of accounts.

```
/* programs/marginfi/src/state/marginfi_account.rs */
/// 4 for `ASSET_TAG_STAKED` (bank, oracle, lst mint, lst pool), 2 for most others (bank, oracle), 3
/// for Kamino (bank, oracle, reserve), 1 for Fixed
fn get_remaining_accounts_per_balance(balance: &Balance) -> MarginfiResult<usize> {
    get_remaining_accounts_per_asset_tag(balance.bank_asset_tag)
}

fn get_remaining_accounts_len(&self) -> MarginfiResult<usize> {
    let mut total = 0usize;
    for balance in self
        .lending_account
        .balances
        .iter()
        .filter(|b| b.is_active())
    {
        let num_accounts = get_remaining_accounts_per_balance(balance)?;
        total += num_accounts;
    }
    Ok(total)
}
```

Currently, these outdated functions are not used anywhere in the program. To avoid confusion in future development, it is recommended to delete them.

Resolution

To be completed after reviewing the 2nd version with fixes for the reported issues.

PR427: FORCED DELEVERAGE

[P3-M-01] Dust liabilities may block TOKENLESS_REPAYMENTS_COMPLETE

Identified in commit `dfc4b03`.

In the `repay` process, the program sets `TOKENLESS_REPAYMENTS_COMPLETE` based on `total_liability_shares` compared against `ZERO_AMOUNT_THRESHOLD`. The `repay_all` path reduces per-balance liabilities until they are treated as zero if the amount is below the threshold, but the completion check compares the sum of shares.

```
/* programs/marginfi/src/instructions/marginfi_account/repay.rs */
// During deleverage, once the last repayment is complete, and the bank's debts have been fully
// discharged, the risk admin becomes empowered to purge the balances of lenders
let liabs: I80F48 = bank.total_liability_shares.into();
if bank.get_flag(TOKENLESS_REPAYMENTS_ALLOWED) && liabs.abs() < ZERO_AMOUNT_THRESHOLD {
    bank.update_flag(true, TOKENLESS_REPAYMENTS_COMPLETE);
}

/* programs/marginfi/src/state/marginfi_account.rs */
let total_liability_shares: I80F48 = balance.liability_shares.into();
let current_liability_amount = bank.get_liability_amount(total_liability_shares)?;
let current_asset_amount = bank.get_asset_amount(balance.asset_shares.into())?;
debug!("Repaying all: {}", current_liability_amount,);
  // note: repay_all rejects current_liability_amount smaller than ZERO_AMOUNT_THRESHOLD
check!(
    current_liability_amount.is_positive_with_tolerance(ZERO_AMOUNT_THRESHOLD),
    MarginfiError::NoLiabilityFound
);
```

Attackers can maintain their recorded `liability_shares` below `ZERO_AMOUNT_THRESHOLD` by specifying an amount via `repay`. When multiple marginfi accounts' `liability_shares` are below `ZERO_AMOUNT_THRESHOLD`, yet the sum of all balances `liability_shares` exceeds `ZERO_AMOUNT_THRESHOLD`, the bank cannot be set to `TOKENLESS_REPAYMENTS_COMPLETE`.

When the `risk_admin` executes the deleverage method and intends to complete repayment without performing an actual transfer, three conditions must be met:

- The instruction must be `repay_all`
- The bank must be in the `TOKENLESS_REPAYMENTS_ALLOWED` state
- The signer must be the `risk_admin`

```
/* programs/marginfi/src/instructions/marginfi_account/repay.rs */
if authority.key() == group.risk_admin
    && bank.get_flag(TOKENLESS_REPAYMENTS_ALLOWED)
    && repay_all
{
    // In some rare cases (e.g. super illiquid token sunset) we allow risk admin
    // to "repay" the debt with nothing. Hence we skip the actual transfer here.

    // repay_all must be enabled: this enables the risk admin to voluntarily pay when it wants,
    // but in general, once the risk admin is prepared to use this feature, there's no point in
    // not repaying the entire balance!

    // Note: Doing this means there will not be enough funds left for lenders to withdraw! This
    // state is irrecoverable. Lenders will be paid out on a first-come-first-served basis as
    // they withdraw. Remaining lenders will either absorb the loss - or more likely - be repaid
    // through some OTC claims portal using assets seized from borrowers
}
```

Since `repay_all` cannot be executed for accounts whose liabilities fall within the `ZERO_AMOUNT_THRESHOLD`, an attacker can intentionally leave residual liabilities with `liability_shares` below this threshold. To reduce the total liabilities below `ZERO_AMOUNT_THRESHOLD` and successfully set `TOKENLESS_REPAYMENTS_COMPLETE`, the `risk_admin` must then execute `repay` for each such account and transfer a small real amount of tokens to extinguish the remaining debt.

This increases the operational complexity of the deleverage process for the `risk_admin` and creates situations in which the `risk_admin` is forced to make token transfers in order to perform the repayment.

It is recommended to drive the completion condition with an admin instruction when the aggregate amount is below a certain threshold.

Resolution

To be completed after reviewing the 2nd version with fixes for the reported issues.

PR427: FORCED DELEVERAGE

[P3-L-01] Deposits allowed when TOKENLESS_REPAYMENTS_ALLOWED is true

Identified in commit `dfc4b03`.

In `lending_account_deposit` function, deposits are blocked when a bank is `Paused` or `ReduceOnly` via `validate_bank_state`.

However, there is no explicit guard for the sunset mode flag `TOKENLESS_REPAYMENTS_ALLOWED`.

As a result, users can still deposit into a bank that is undergoing tokenless repayments, potentially trapping funds in a sunset process and exposing depositors to avoidable loss or illiquidity.

```
/* programs/marginfi/src/instructions/marginfi_account/deposit.rs */
150 | #[account(
151 |     mut,
152 |     has_one = group @ MarginfiError::InvalidGroup,
153 |     has_one = liquidity_vault @ MarginfiError::InvalidLiquidityVault,
154 |     constraint = is_marginfi_asset_tag(bank.load()?.config.asset_tag)
155 |         @ MarginfiError::WrongAssetTagForStandardInstructions
156 | )]
157 | pub bank: AccountLoader<'info, Bank>,
```

It is recommended to add an explicit check to reject deposits when `bank.get_flag(TOKENLESS_REPAYMENTS_ALLOWED)` is `true`.

Resolution

To be completed after reviewing the 2nd version with fixes for the reported issues.

PR427: FORCED DELEVERAGE

[P3-I-01] Deleverage withdrawal limits bypass by changing the authority

Identified in commit `dfc4b03`.

In the `start_deleverage` and `end_deleverage` flow, the withdraw limit is applied only when `authority == group.risk_admin`.

However, under receivership, the authority account may be any signer. As a result, the risk admin can switch the authority to an arbitrary signer within the same deleverage transaction, thereby bypassing the intended withdrawal limit.

Although `start_deleverage` explicitly sets `liquidation_record.liquidation_receiver` to the risk admin, this does not guarantee that `authority == group.risk_admin` during the execution of withdrawal functions.

This issue affects both the `lending_account_withdraw` and `kamino_withdraw` methods.

```
/* programs/marginfi/src/instructions/marginfi_account/withdraw.rs */
// Note: we only care about the withdraw limit in case of deleverage
if authority.key() == group.risk_admin {
    let withdrawn_equity = calc_value(
        I80F48::from_num(amount_pre_fee),
        price,
        bank.mint_decimals,
        None,
    )?;
    group.update_withdrawn_equity(withdrawn_equity, clock.unix_timestamp)?;
}

/* programs/marginfi/src/instructions/kamino/withdraw.rs */
if ctx.accounts.authority.key() == group.risk_admin {
    let withdrawn_equity = calc_value(
        I80F48::from_num(collateral_amount),
        price,
        bank.mint_decimals,
        None,
    )?;
    group.update_withdrawn_equity(withdrawn_equity, clock.unix_timestamp)?;
}
```

It is recommended to enforce a check ensuring `authority == liquidation_record.liquidation_receiver` for deleverage operations.

Resolution

To be completed after reviewing the 2nd version with fixes for the reported issues.

OUT OF SCOPE

[P4-L-01] Incorrect flag check disables program fees

Identified in commit `dfc4b03`.

The issue was introduced in PR312, where a new `ARENA_GROUP` flag was added to `group_flags`.

In `get_group_bank_config`, the check for enabled program fees uses a direct equality operator (`==`) instead of a bitwise operation (`&`) to isolate the `PROGRAM_FEES_ENABLED` bit. Consequently, any group with the `ARENA_GROUP` flag set is incorrectly treated as having program fees disabled. The correct logic should match the implementation found in `program_fees_enabled`.

```
/* programs/marginfi/src/state/marginfi_group.rs */
pub const PROGRAM_FEES_ENABLED: u64 = 1;
pub const ARENA_GROUP: u64 = 2;

fn get_group_bank_config(&self) -> GroupBankConfig {
    GroupBankConfig {
        program_fees: self.group_flags == PROGRAM_FEES_ENABLED,
    }
}

fn set_program_fee_enabled(&mut self, fee_enabled: bool) {
    if fee_enabled {
        self.group_flags |= PROGRAM_FEES_ENABLED;
    } else {
        self.group_flags &= !PROGRAM_FEES_ENABLED;
    }
}

fn set_arena_group(&mut self, is_arena: bool) -> MarginfiResult {
    if is_arena {
        self.group_flags |= ARENA_GROUP;
    } else {
        self.group_flags &= !ARENA_GROUP;
    }
}

/// True if program fees are enabled
fn program_fees_enabled(&self) -> bool {
    (self.group_flags & PROGRAM_FEES_ENABLED) != 0
}
```

However, `program_fees_enabled`, which is the correct one, is not currently used in production code, but `get_group_bank_config` is called by `accrue_interest` to create the interest rate calculator. This value is subsequently used in the calculation flow: `calc_interest_rate_accrual_state_changes -> calc_interest_rate -> get_fees`. The incorrect `add_program_fees` value results in the group failing to collect protocol fees.

```
/* programs/marginfi/src/state/interest_rate.rs */
fn create_interest_rate_calculator(&self, group: &MarginfiGroup) -> InterestRateCalc {
    let group_bank_config = &group.get_group_bank_config();
    InterestRateCalc {
        add_program_fees: group_bank_config.program_fees,
    }
}

/* programs/marginfi/src/state/bank.rs */
fn accrue_interest(
) -> MarginfiResult<()> {
    let ir_calc = self
        .config
        .interest_rate_config
        .create_interest_rate_calculator(group);

    let InterestRateStateChanges {
        new_asset_share_value: asset_share_value,
        new_liability_share_value: liability_share_value,
        insurance_fees_collected,
        group_fees_collected,
        protocol_fees_collected,
    } = calc_interest_rate_accrual_state_changes(
        time_delta,
        total_assets,
        total_liabilities,
        &ir_calc,
        self.asset_share_value.into(),
        self.liability_share_value.into(),
    )
    .ok_or_else(math_error!())?;
}

/* programs/marginfi/src/state/interest_rate.rs */
pub fn calc_interest_rate_accrual_state_changes(
) -> Option<InterestRateStateChanges> {
    let interest_rates = interest_rate_calc.calc_interest_rate(utilization_rate)?;
}

/* programs/marginfi/src/state/interest_rate.rs */
pub fn calc_interest_rate(&self, utilization_ratio: I80F48) -> Option<ComputedInterestRates> {
    let Fees {
        insurance_fee_rate,
        insurance_fee_fixed,
        group_fee_rate,
        group_fee_fixed,
        protocol_fee_rate,
        protocol_fee_fixed,
    } = self.get_fees();
}

/* programs/marginfi/src/state/interest_rate.rs */
pub fn get_fees(&self) -> Fees {
    let (protocol_fee_rate, protocol_fee_fixed) = if self.add_program_fees {
        (self.program_fee_rate, self.program_fee_fixed)
    } else {
        (I80F48::ZERO, I80F48::ZERO)
    };

    Fees {
            insurance_fee_rate: self.insurance_rate_fee,
            insurance_fee_fixed: self.insurance_fixed_fee,
            group_fee_rate: self.protocol_rate_fee,
            group_fee_fixed: self.protocol_fixed_fee,
            protocol_fee_rate,
            protocol_fee_fixed,
        }
}
```

Although a recent PR indicates that arena functionality is going to be sunset, it is still recommended to fix the logic in `get_group_bank_config`. This ensures the function returns the correct result even if new bits are added to `group_flags` in the future.

Resolution

To be completed after reviewing the 2nd version with fixes for the reported issues.

Appendix: Methodology and Scope of Work

Assisted by the Sec3 Scanner developed in-house, the manual audit particularly focused on the following work items:

- Check common security issues.
- Check program logic implementation against available design specifications.
- Check poor coding practices and unsafe behavior.
- The soundness of the economics design and algorithm is out of scope of this work

DISCLAIMER

ABOUT

The Sec3 audit team comprises a group of computer science professors, researchers, and industry veterans with extensive experience in smart contract security, program analysis, testing, and formal verification. We are also building automated security tools that incorporate static analysis, penetration testing, and formal verification.

At Sec3, we identify and eliminate security vulnerabilities through the most rigorous process and aided by the most advanced analysis tools.

For more information, check out our website and follow us on twitter.

Sec3™