THREE SIGMA

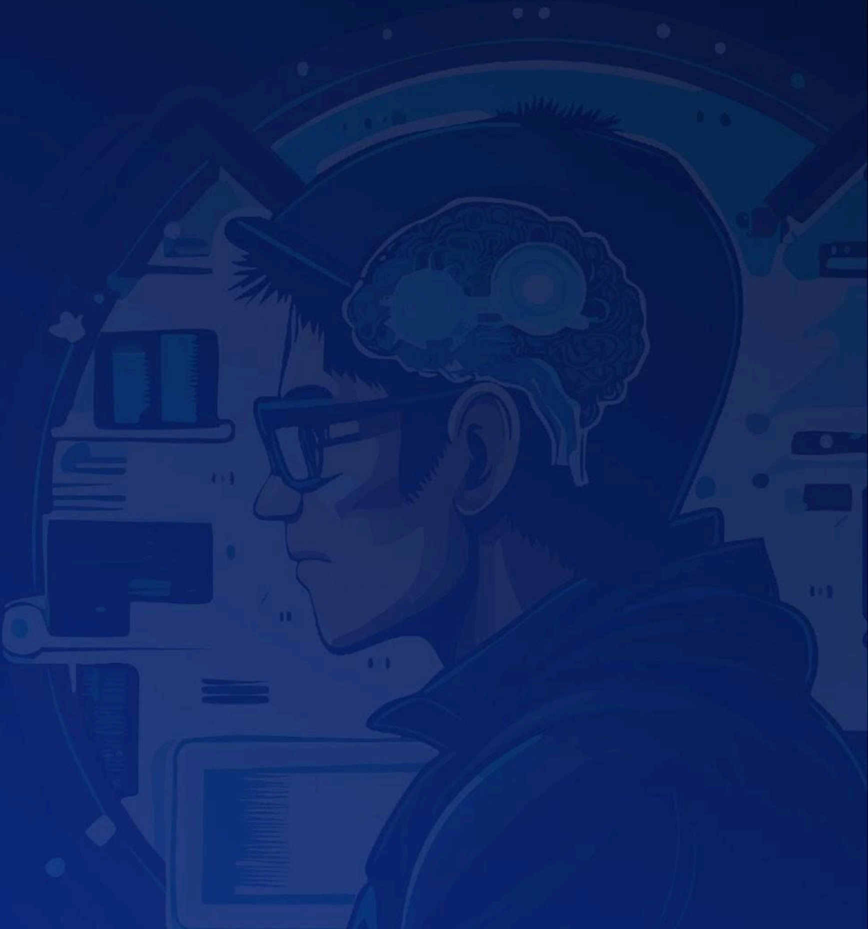
Deepbook Margin

Security Review

Disclaimer

Security Review

Deepbook Margin



Disclaimer

Table of Contents

Security Review

Deepbook Margin

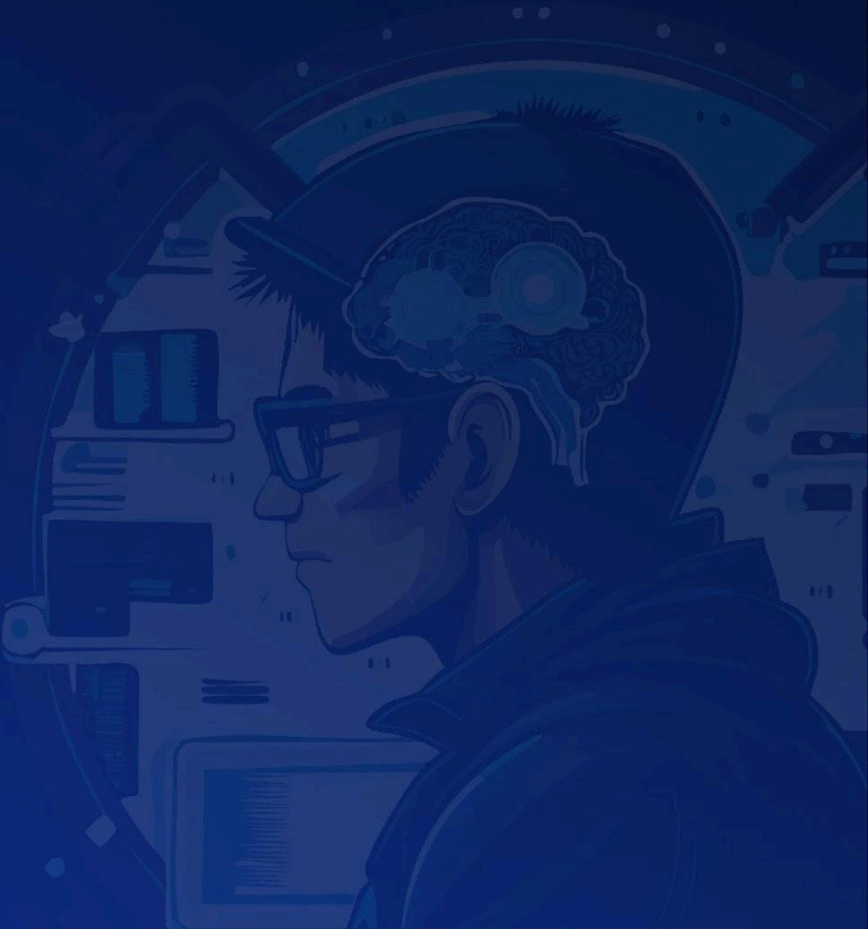
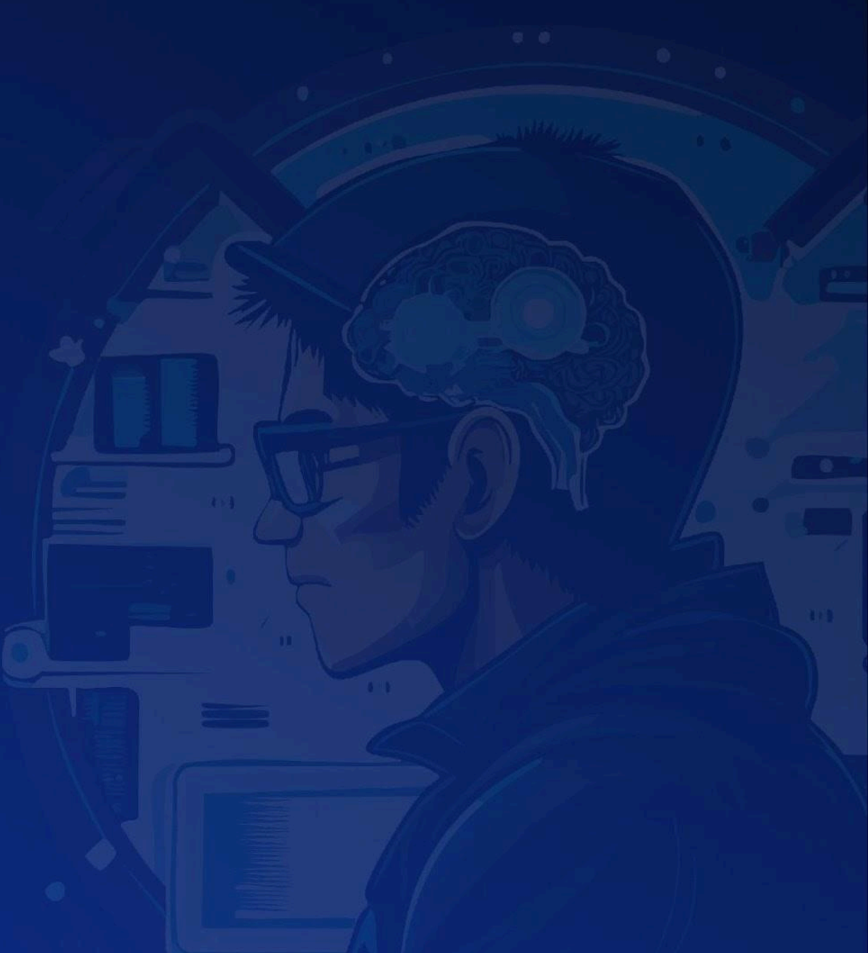


Table of Contents

Summary

Security Review

Deepbook Margin



Summary

Three Sigma audited Deepbook in a 13.2 person week engagement. The audit was conducted from October 6 2025 to November 5 2025.

Protocol Description

DeepBook Margin extends the trading capabilities of DeepBookV3 by enabling leveraged trading positions. With margin trading, users can borrow funds to increase their buying power.

DeepBook Margin provides the following capabilities:

- Leveraged positions: Trade with borrowed funds to increase position sizes beyond available capital

- Risk management: Built-in liquidation mechanisms to protect lenders and maintain system solvency

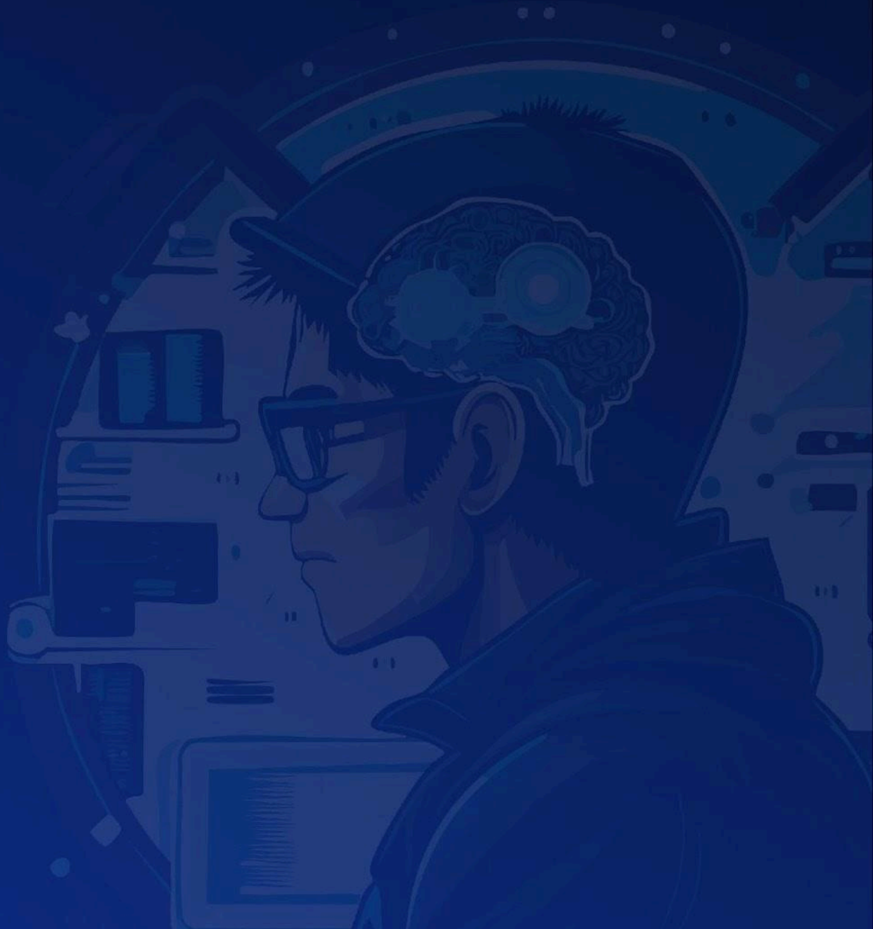
- Collateral flexibility: Support for multiple assets as collateral for isolated margin positions

- Interest accrual: Transparent interest rate calculations for borrowed funds

Scope

Security Review

Deepbook Margin



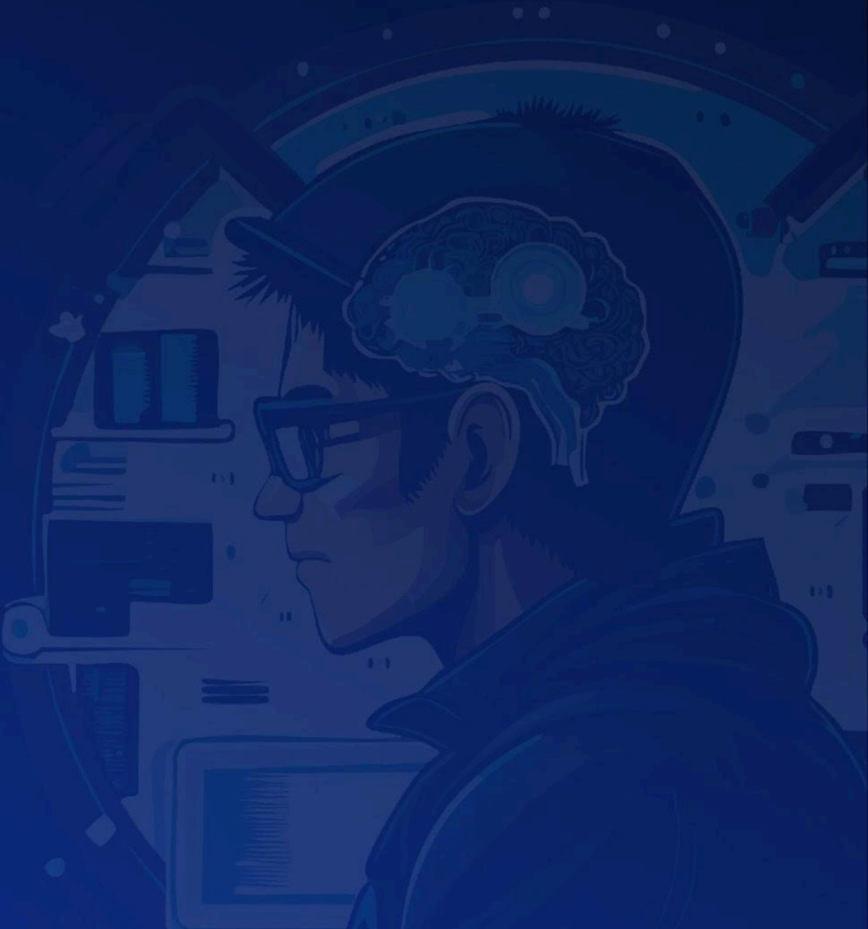
Scope

Pull Request: https://github.com/MystenLabs/deepbookv3/pull/494/files

Methodology

Security Review

Deepbook Margin



Methodology

To begin, we reasoned meticulously about the contract's business logic, checking security-critical features to ensure that there were no gaps in the business logic and/or inconsistencies between the aforementioned logic and the implementation. Second, we thoroughly examined the code for known security flaws and attack vectors. Finally, we discussed the most catastrophic situations with the team and reasoned backwards to ensure they are not reachable in any unintentional form.

Taxonomy

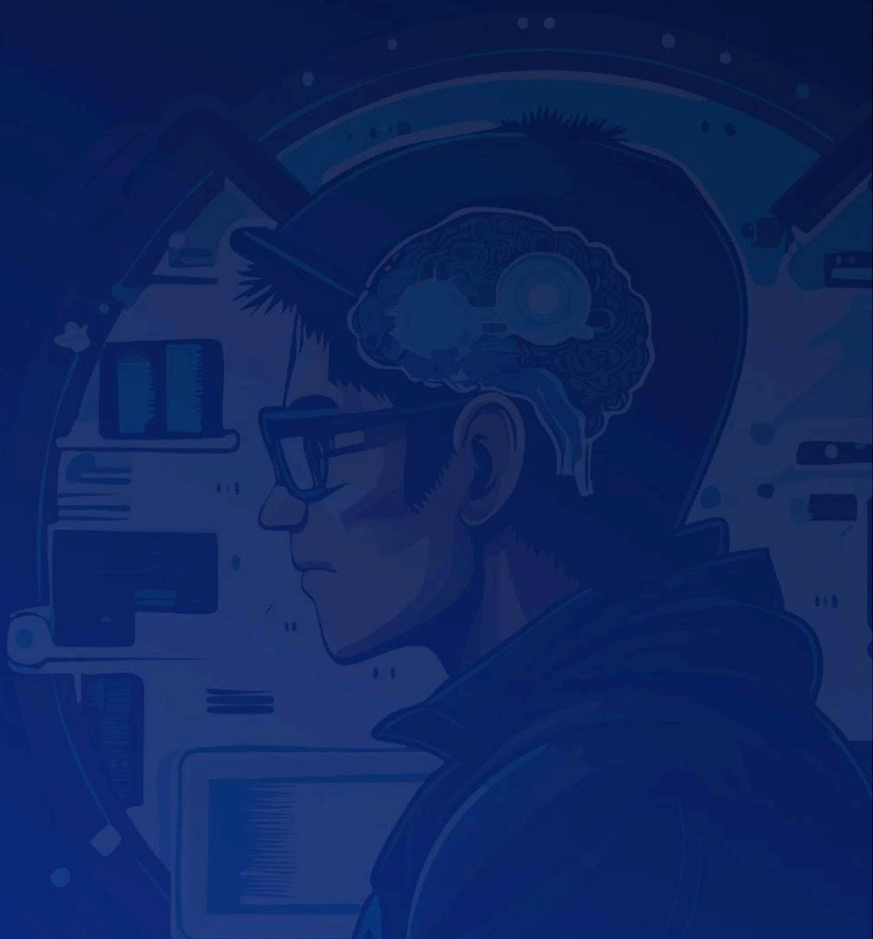
In this audit, we classify findings based on Immunefi's Vulnerability Severity Classification System (v2.3) as a guideline. The final classification considers both the potential impact of an issue, as defined in the referenced system, and its likelihood of being exploited. The following table summarizes the general expected classification according to impact and likelihood; however, each issue will be evaluated on a case-by-case basis and may not strictly follow it.

Impact / Likelihood	LOW	MEDIUM	HIGH
NONE	None		
LOW	Low		
MEDIUM	Low	Medium	Medium
HIGH	Medium	High	High
CRITICAL	High	Critical	Critical

Project Dashboard

Security Review

Deepbook Margin



Project Dashboard

Application Summary

Name	Deepbook
Repository	https://github.com/MystenLabs/deepbookv3
Commit	e9a703a
Language	Move
Platform	Sui

Engagement Summary

Timeline	October 6 2025 to November 5 2025
№ of Auditors	3
Review Time	13.2 person weeks

Vulnerability Summary

Issue Classification	Found	Addressed	Acknowledged
Critical	0	0	0
High	1	1	0
Medium	5	5	0
Low	2	2	0
None	0	0	0

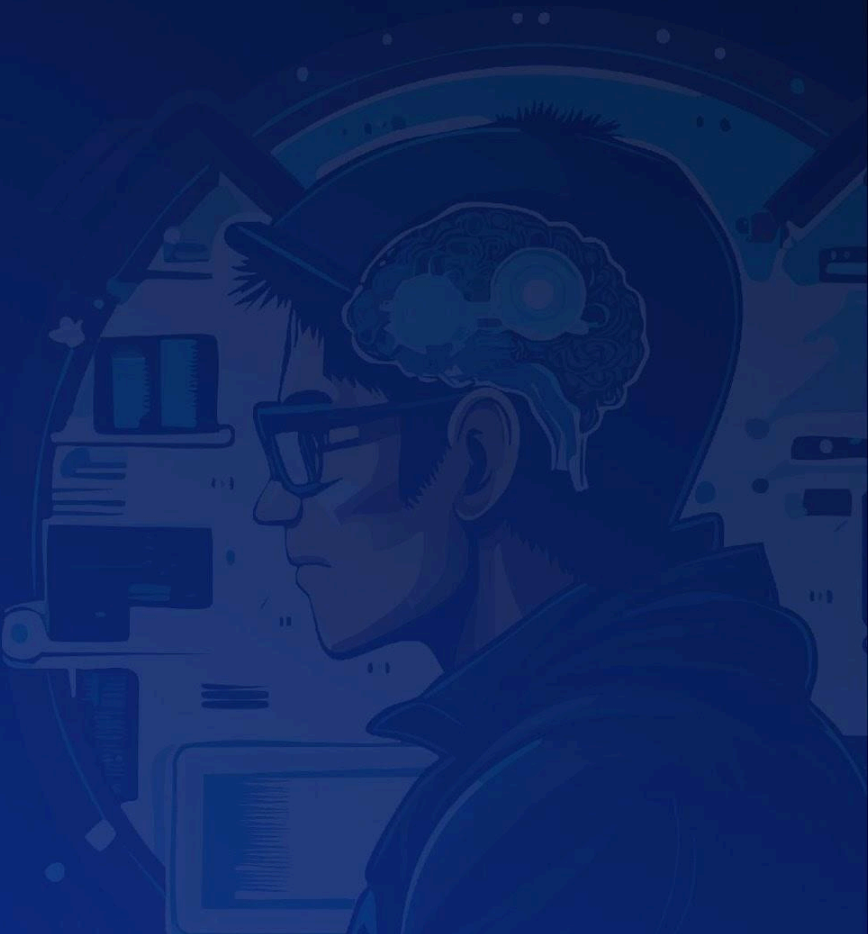
Category Breakdown

Suggestion	0
Documentation	0
Bug	8
Optimization	0
Good Code Practices	0

Risk Section

Security Review

Deepbook Margin



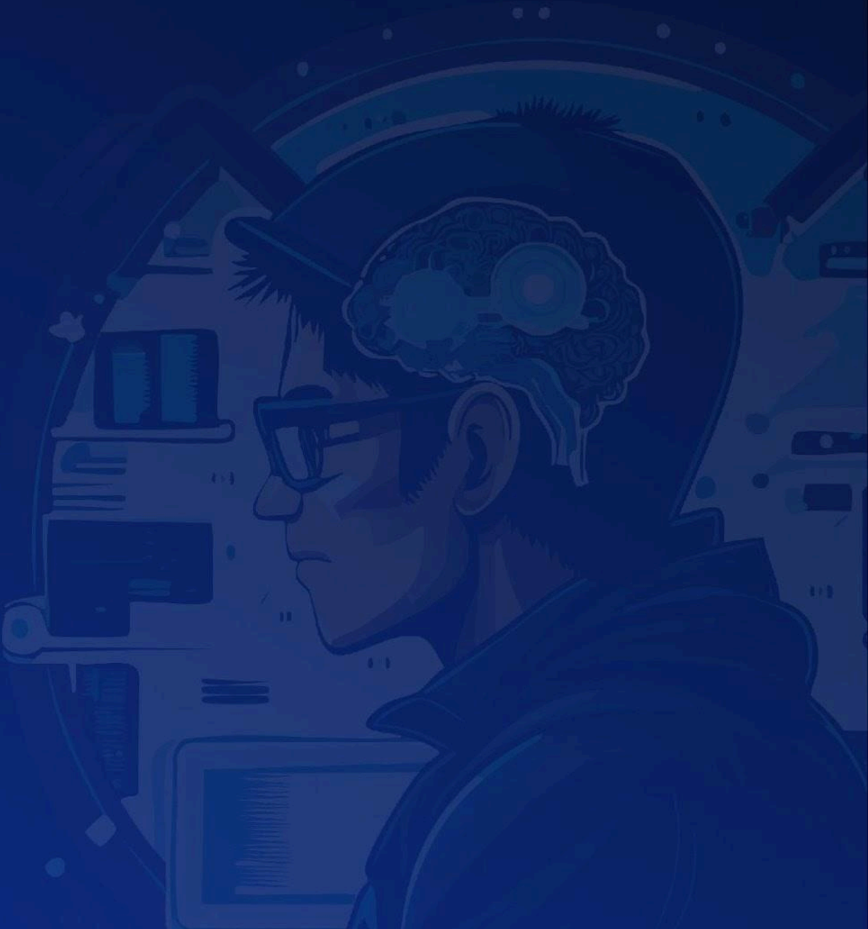
Risk Section

No risks were identified.

Findings

Security Review

Deepbook Margin



Findings

3S-Deepbook-H01

Default referral address lacks claim mechanism leading to permanent fee lock

Id	3S-Deepbook-H01
Classification	High
Impact	High
Likelihood	High
Category	Bug
Status	Addressed in #21dfd3.

Description

The **ReferralFees** module initializes a default referral address (**0x0**) during system setup through **default_referral_fees**. This default referral is used for all users who supply assets without specifying a referrer. The issue is that while the default referral accumulates shares and earns fees like any other referral, there is no mechanism to claim these fees, resulting in permanent value lock within the protocol.

The **calculate_and_claim** function requires a **SupplyReferral** object with an owner who can authorize the claim. However, no **SupplyReferral** object is created for the default referral address during initialization, and the **0x0** address has no corresponding private key to create or control such an object.

```
// Default referral is added during initialization
public(package) fun default_referral_fees(ctx: &mut TxContext): ReferralFees {
    let default_id = margin_constants::default_referral(); // Returns 0x0
    let mut manager = ReferralFees { /* ... */ };
    manager.referrals.add(
        default_id,
        ReferralTracker {
            current_shares: 0,
            min_shares: 0,
        },
    );
    // No SupplyReferral object created for default_id!
    manager
}
// Claims require a SupplyReferral object with an owner
public(package) fun calculate_and_claim(
    self: &mut ReferralFees,
    referral: &mut SupplyReferral,  // Required but doesn't exist for 0x0
    ctx: &TxContext,
): u64 {
    assert!(ctx.sender() == referral.owner, ENotOwner);
    // ...
}
```

This design flaw affects a significant portion of the protocol's fee distribution. Users who don't use referral codes will have their referral fees allocated to the unclaimed default address, where they accumulate indefinitely. Over time, this could lock substantial value that should either be distributed to active participants or allocated to the protocol treasury.

Steps to Reproduce:

1. System initializes with default referral at address `0x0`

2. User A supplies 500,000 USDC without a referral code

3. User B supplies 300,000 USDC without a referral code

4. Both users are assigned to default referral, which now has 800,000 shares

5. Trading activity generates 10,000 USDC in total referral fees

6. Default referral's portion: `(800,000 / total_shares) * 10,000` USDC

7. These fees remain in the vault but are unclaimable as no one can create or control a `SupplyReferral` for address `0x0`

8. The locked fees continue accumulating indefinitely

Recommendation

Exclude the default referral from share distribution entirely and allocate those fees directly to the legit referrals or to protocol treasury.

Alternatively, Implement a mechanism to handle default referral fees appropriately:

```
+ // Add a treasury claim function for default referral fees
+ public fun claim_default_referral_fees(
+     self: &mut ReferralFees,
+     admin_cap: &MarginAdminCap,
+ ): u64 {
+     let default_address = margin_constants::default_referral();
+     let referral_tracker = self.referrals.borrow_mut(default_address);
+     let referred_shares = referral_tracker.min_shares;
+     let fees_per_share_delta = self.fees_per_share - 0; // Assuming tracking from genesis
+     let fees = math::mul(referred_shares, fees_per_share_delta);
+
+     referral_tracker.min_shares = referral_tracker.current_shares;
+     fees
+ }
```

3S-Deepbook-M01

Margin pool ID not reset after full liquidation

Id	3S-Deepbook-M01
Classification	Medium
Impact	Medium
Likelihood	Medium
Category	Bug
Status	Addressed in #d7930e.

Description

margin_manager.move#repay

margin_manager.move#liquidate

When a position is fully liquidated (all debt repaid), the **margin_pool_id** is not reset to **None**, leaving the account in an inconsistent state where **debt = 0** but the account remains linked to the old margin pool.

The bug is present in **margin_manager.move**, function **liquidate()** where no reset of **margin_pool_id** happens when both shares reach 0

```
// liquidate()
if (debt_is_base) {
    self.borrowed_base_shares = self.borrowed_base_shares - repay_shares;
} else {
    self.borrowed_quote_shares = self.borrowed_quote_shares - repay_shares;
};
// @audit-poc no reset of margin_pool_id when both shares reach 0
```

if you compare the code with **repay()** it correctly resets **self.margin_pool_id** on full debt repaying

```
// repay()
if (self.borrowed_base_shares == 0 && self.borrowed_quote_shares == 0) {
    self.margin_pool_id = option::none();  // @audit correctly resets
}
```

This issue will arise on every full liquidation where all borrowed shares are repaid.

When liquidator repays entire debt → shares become 0 → margin_pool_id remains set → user can only borrow from same pool again, not different pools.

One **concrete impact** of this is that user permanently locked to one margin pool & cannot borrow from alternative pools even with zero debt, effectively DoS'ing the margin account.

Recommendation

Add the same reset logic from **repay()** to **liquidate()**

```
// in liquidate():
if (self.borrowed_base_shares == 0 && self.borrowed_quote_shares == 0) {
    self.margin_pool_id = option::none();
}
```

3S-Deepbook-M02

Incorrect use of new_mean instead of old_mean in new variance calculation

Id	3S-Deepbook-M02
Classification	Medium
Impact	Medium
Likelihood	Medium
Category	Bug
Status	Addressed in #aee939.

Description:

The **ewma::update** function in the DeepBook protocol implements an Exponentially Weighted Moving Average (EWMA) algorithm to track gas price volatility and apply dynamic taker fees based on Z-score calculations. The function calculates both a smoothed mean and variance of gas prices over time, which are used to determine when gas prices are unusually high and warrant additional taker fees.

The EWMA variance update formula should use the previous mean value when calculating the difference for variance updates. However, the current implementation correctly uses mean_new (the new mean) in the difference calculation.

The correct EWMA variance formula is:

variance_new = (1 - α) × variance_old + α × (current_price - mean_old) ** 2

Where **mean_old** is the previous mean value, not the newly calculated mean. The variance calculation measures how much the current gas price deviates from the historical average, which requires using the old mean as the reference point.

Using **mean_new** instead of **self.mean** in the variance calculation creates a systematic bias in each update. When the new mean is used for the difference calculation, the resulting difference becomes smaller because the new mean is already closer to the current gas price than the old mean. This smaller difference leads to a smaller diff_squared value, which directly results in a lower **variance_new** calculation.

The lower variance has a cascading effect on the system's penalty mechanism. Since the Z-score is calculated as the ratio of the price difference to the square root of variance, a lower variance results in a higher Z-score. Higher Z-scores mean that the system perceives gas price volatility as more extreme than it actually is, triggering taker penalties more frequently and at lower price thresholds than justified by actual market conditions.

Recommendation:

The variance calculation in update function should use the self.mean (the old mean) in the variance calculation instead of mean_new

```
- let diff = if (gas_price > mean_new) {
-     gas_price - mean_new
- } else {
-     mean_new - gas_price
- };
+ let diff = if (gas_price > self.mean) {
+     gas_price - self.mean
+ } else {
+     self.mean - gas_price
+ };
```

3S-Deepbook-M03

Rounding Down in Share Calculation Enables Free Token Extraction

Id	3S-Deepbook-M03
Classification	Medium
Impact	Medium
Likelihood	Medium
Category	Bug
Status	Addressed in #630065.

Description:

margin_state.move#L94

The **margin_state::increase_borrow()** functions in the DeepBook margin protocol contain a critical rounding down vulnerability. The **margin_state::increase_borrow()** function should use rounding up for share calculations when users borrow assets. Currently, the function uses **math::div(amount, ratio)** which rounds down, but in standard DeFi applications, borrowing operations should round up shares to favour the protocol.

The margin state module manages the total supply and borrow of margin pools using a share-based accounting system. Shares represent constant amounts that are used to calculate actual amounts after interest and protocol fees are applied. The system maintains two key ratios: **supply_ratio = total_supply / supply_shares** and **borrow_ratio = total_borrow / borrow_shares**.

```
public(package) fun increase_borrow(
    self: &mut State,
    config: &ProtocolConfig,
    amount: u64,
    clock: &Clock,
): (u64, u64) {
    let protocol_fees = self.update(config, clock);
    let ratio = self.borrow_ratio();
>>  let shares = math::div(amount, ratio);  // Rounds down
    self.borrow_shares = self.borrow_shares + shares;
    self.total_borrow = self.total_borrow + amount;
    (shares, protocol_fees)
}
```

In standard DeFi applications the supply shares should round down and borrow shares should round up to favour the protocol

---

Recommendation:

Handle the rounding correctly to ensure the protocol is properly protected against rounding-related accounting issues.

3S-Deepbook-M04

Referral **min_shares** initialization at zero leads to permanent loss of first period fees

Id	3S-Deepbook-M04
Classification	Medium
Impact	Medium
Likelihood	Medium
Category	Bug
Status	Addressed in #07f628.

Description

The **ReferralFees** module implements a fee distribution system where referrals earn commission based on their **min_shares** tracked in **ReferralTracker**. The module intends to prevent gaming by tracking the minimum shares held during a period to calculate fair fee distribution. However, when a new referral is created through **mint_supply_referral** or during system initialization with **default_referral_fees**, the **ReferralTracker** is initialized with both **current_shares** and **min_shares** set to 0.

The critical flaw is that **min_shares** never increases from this initial zero value. The **increase_shares** function only updates **current_shares**, while **min_shares** can only decrease through the **decrease_shares** function or be reset during a claim. This means referrals earn nothing until they perform their first claim transaction, which unintuively sets **min_shares** to **current_shares** for future periods.

```
// In referral_fees module
public struct ReferralTracker has store {
    current_shares: u64,
    min_shares: u64,  // Initialized to 0, never increases
}
// increase_shares only updates current_shares
public(package) fun increase_shares(
    self: &mut ReferralFees,
    referral: Option<address>,
    shares: u64,
) {
    let referral_address = referral.destroy_with_default(margin_constants::default_referral());
    let referral_tracker = self.referrals.borrow_mut(referral_address);
    referral_tracker.current_shares = referral_tracker.current_shares + shares;
    // min_shares remains 0!
    self.total_shares = self.total_shares + shares;
}
```

Steps to Reproduce:

1. Alice creates a new referral through **mint_supply_referral**, receiving a **SupplyReferral** object

2. Bob supplies 100,000 USDC using Alice's referral address, generating 100,000 shares

3. Alice's **ReferralTracker** shows **current_shares: 100,000** but **min_shares: 0**

4. Significant fees accrue over time, increasing **fees_per_share** from 0 to 0.1

5. Alice calls **calculate_and_claim** expecting to receive fees

6. The calculation yields: **fees = min_shares * fees_per_share_delta = 0 * 0.1 = 0**

7. Alice receives 0 USDC despite referring 100,000 USDC in deposits

---

Recommendation

Modify referral rewarding mechanism to account fees accured before claim.

3S-Deepbook-M05

Missing Referral Owner Validation in update_referral_multiplier leads to Unauthorized Referral Manipulation

Id	3S-Deepbook-M05
Classification	Medium
Impact	Medium
Likelihood	Medium
Category	Bug
Status	Addressed in #e21495.

Description :

pool.move#L868

The **deepbook::pool::update_referral_multiplier()** function allows any user to modify referral reward multipliers for any **DeepBookReferral** object, including setting them to zero. This vulnerability exists because **DeepBookReferral** objects are shared objects that can be accessed by any user, but the function lacks proper ownership validation.

The **update_referral_multiplier** function is designed to allow referral owners to update their referral reward multipliers. However, since **DeepBookReferral** objects are shared objects (created with **transfer::share_object(referral)** in the **mint_referral** function), any user can pass any **DeepBookReferral** object to this function, regardless of ownership.

The vulnerability allows malicious actors to:

- Set other users' referral multipliers to zero, effectively disabling their referral rewards

- Manipulate referral multipliers to arbitrary values within the allowed range

- Disrupt the referral system by targeting specific referral objects

The function performs validation on the multiplier value (ensuring it's within bounds), but fails to verify that the caller is the actual owner of the **DeepBookReferral** object being modified.

Steps to Reproduce:

- User A creates a **DeepBookReferral** object through the **mint_referral** function, which becomes a shared object

- User B (malicious actor) calls **pool::update_referral_multiplier()** passing User A's **DeepBookReferral** object and sets the multiplier to 0

- The function executes successfully without ownership validation, setting User A's referral multiplier to 0

- User A's referral rewards are effectively disabled, as the multiplier of 0 results in no referral fees being distributed

---

Recommendation :

Add proper ownership validation at the beginning of the function to ensure only the referral owner can modify their referral rewards multiplier.

3S-Deepbook-L01

Interest rate prarameters apply retroactively

Id	3S-Deepbook-L01
Classification	Low
Impact	Low
Likelihood	Low
Category	Bug
Status	Addressed in #4e7fc9.

Description

The **update_interest_params()** function in margin_pool.move updates the interest configuration without first accruing interest at the current rates. This causes new interest rates to apply retroactively to the entire period since the last state update.

The function directly sets **self.config.set_interest_config(interest_config)** without calling **self.state.update(&self.config, clock)** first. When interest is next calculated (*during any borrow/repay/supply/withdraw*), the new rates apply to the entire elapsed period rather than just future accrual.

a simple **example scenario**

1. User borrows $10,000 at 5% APR on Day 1

2. Admin changes rate to 50% APR on Day 30

3. User repays on Day 31 → charged 50% for entire 30 days (*$411 instead of $41*)

One concrete impact of this is that it enables retroactive interest rate manipulation, allowing admins to overcharge borrowers or undercharge them at the expense of lenders.

Recommendation

Add state update before config change

```
public fun update_interest_params<Asset>(...) {
    registry.load_inner();
    assert!(margin_pool_cap.margin_pool_id() == self.id(), EInvalidMarginPoolCap);
    // Add this line to lock in current interest
    self.state.update(&self.config, clock);

    self.config.set_interest_config(interest_config);
    // ... rest of function
}
```

Note: The same issue exists in **update_margin_pool_config()** where **protocol_spread** changes retroactively affect protocol fee calculations.

3S-Deepbook-L02

Zero Fee Processing in increase_fees_accrued

Id	3S-Deepbook-L02
Classification	Low
Impact	Low
Likelihood	Low
Category	Bug
Status	Addressed in #1c0b7c.

Description: :

The **increase_fees_accrued** function in **deepbook_margin::protocol_fees** module is responsible for distributing accrued fees among protocol participants (referrals, maintainer, and protocol). The function calculates fee distribution by dividing referral fees by total shares and updates the fees per share accordingly.

protocol_fees.move#L98

The current implementation contains redundant logic that processes zero fees when **total_shares > 0**, which is unnecessary and wastes gas. The function performs fee calculations and emits events even when **fees_accrued** is zero, leading to inefficient execution.

Recommendation :

Implement an early return optimization to skip processing when **fees_accrued** is zero, reducing gas consumption and improving efficiency

```
public(package) fun increase_fees_accrued(self: &mut ProtocolFees, fees_accrued: u64) {
-    assert!(fees_accrued == 0 || self.total_shares > 0, EInvalidFeesAccrued);
+     if (fees_accrued == 0) return;
+     assert!(self.total_shares > 0, EInvalidFeesAccrued);
     let protocol_fees = fees_accrued / 4;
     let maintainer_fees = fees_accrued / 4;
     let referral_fees = fees_accrued - protocol_fees - maintainer_fees;

     if (self.total_shares > 0) {
```

```
    let fees_per_share_increase = math::div(referral_fees, self.total_shares);
    self.fees_per_share = self.fees_per_share + fees_per_share_increase;
    self.maintainer_fees = self.maintainer_fees + maintainer_fees;
    self.protocol_fees = self.protocol_fees + protocol_fees;
};
event::emit(ProtocolFeesIncreasedEvent {
    total_shares: self.total_shares,
    referral_fees,
    maintainer_fees,
    protocol_fees,
});
```