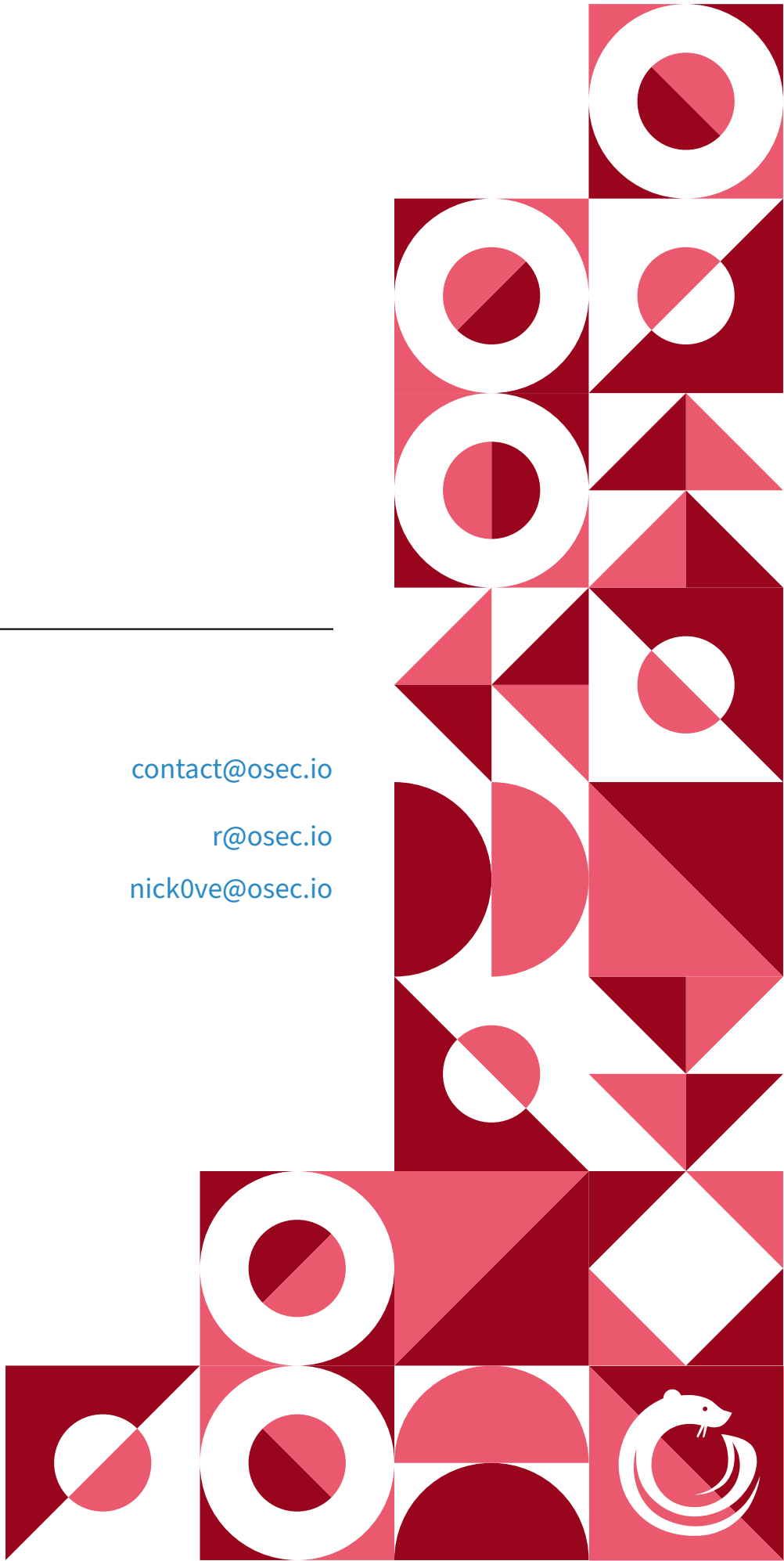


marginfi v2

Audit

Presented by:

OtterSec contact@osec.io

Robert Chen r@osec.io

Nicola Vella nick0ve@osec.io

Contents

01 Executive Summary **2**

Overview . . . 2

Key Findings . . . 2

02 Scope **3**

03 Findings **4**

04 Vulnerabilities **5**

OS-MRG-ADV-00 [crit] [resolved] | Inadequate Liquidation Access Control . . . 6

OS-MRG-ADV-01 [med] [resolved] | Improper Token Accounting . . . 9

OS-MRG-ADV-02 [low] [resolved] | Incorrect Fees Accounting . . . 11

OS-MRG-ADV-03 [low] [resolved] | Incorrect Order of Operations in Liquidation . . . 13

05 General Findings **14**

OS-MRG-SUG-00 | Unchecked Loss Socialization . . . 15

OS-MRG-SUG-01 | Tighter Liquidation Access Control . . . 16

OS-MRG-SUG-02 | Outdated Bank Interest . . . 17

OS-MRG-SUG-03 | Incorrect Bank Deserialization . . . 18

OS-MRG-SUG-04 | Weak Weight Validation . . . 19

Appendices

A Vulnerability Rating Scale **20**

B Procedure **21**

01 | Executive Summary

Overview

marginfi engaged OtterSec to perform an assessment of the `marginfi-v2` program. This assessment was conducted between January 17th and February 10th, 2023. For more information on our auditing methodology, see Appendix B.

All the issues were communicated to the team prior to the delivery of the report to speed up remediation. After delivering our audit report, we worked closely with the team to streamline patches and confirm remediation. We delivered final confirmation of the patches February 14th, 2023.

Key Findings

Over the course of this audit engagement, we produced 9 findings in total.

In particular, we found a loss of funds concern with protocol liquidations (OS-MRG-ADV-00), relying on unsafe liquidation logic and a subtle fixed point decimal precision issue. We also noted improper token accounting (OS-MRG-ADV-01) and a minor denial of service issue with liquidations (OS-MRG-ADV-03).

We also noted some suggestions in the way the protocol handles bank deserialization (OS-MRG-SUG-03), execution of operations on outdated bank states (OS-MRG-SUG-02), and potentially inaccurate weight validation (OS-MRG-SUG-04).

Overall, we commend the marginfi team for being responsive and knowledgeable throughout the audit.

02 | Scope

The source code was delivered to us in a git repository at github.com/mrgnlabs/marginfi-v2/. This audit was performed against commit `0f710a4`.

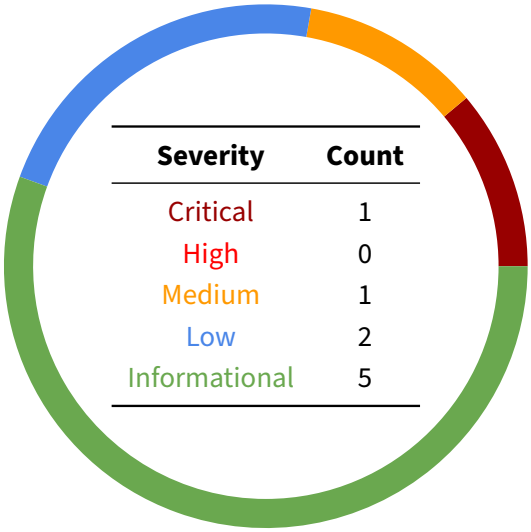
A brief description of the program is as follows.

Name	Description
marginfi-v2	Lending and borrowing protocol on Solana. Users can provide liquidity to the protocol in order to earn interest on their assets or borrow assets to trade on margin.

03 | Findings

Overall, we reported 9 findings.

We split the findings into **vulnerabilities** and **general findings**. Vulnerabilities have an immediate impact and should be remediated as soon as possible. General findings don't have an immediate impact but will help mitigate future vulnerabilities.



Severity	Count
Critical	1
High	0
Medium	1
Low	2
Informational	5

04 | Vulnerabilities

Here, we present a technical analysis of the vulnerabilities we identified during our audit. These vulnerabilities have *immediate* security implications, and we recommend remediation as soon as possible.

Rating criteria can be found in Appendix A.

ID	Severity	Status	Description
OS-MRG-ADV-00	Critical	Resolved	Inadequate liquidation access control allows liquidators to steal shares from liquidatees.
OS-MRG-ADV-01	Medium	Resolved	Improper token accounting allows for utilization ratio manipulation.
OS-MRG-ADV-02	Low	Resolved	Unchecked truncation in `accrue_interest` might result in incorrect fees accounting.
OS-MRG-ADV-03	Low	Resolved	The current order of operations in the liquidation process might result in a DOS.

OS-MRG-ADV-00 [crit] [resolved] | Inadequate Liquidation Access Control

Description

The liquidation of accounts in unhealthy positions is implemented in the LendingAccountLiquidate instruction. The purpose of this instruction is to enable a liquidator to purchase the collateral tokens of insolvent liquidatees at a discounted price, by providing liability tokens to cover the liquidatee's debt.

The main issue is that access control on the liquidatee's account health is only enforced **after** the liquidation logic is completed, through the `RiskEngine::check_post_liquidation_account_health` function. As stated in the comments of this function, the check relies on the strong assumption that **liquidation always reduces risk**.

Two checks are performed:

1. That the `liability_shares` of the liquidatee's liability token are not zero after liquidation.
2. That the liquidatee's account remains in an unhealthy state after liquidation.

src/state/marginfi_account.rs RUST

```
    pub fn check_post_liquidation_account_health(&self, bank_pk: &Pubkey)
    ↪   -> MarginfiResult {
        ...
        check!(
            liability_bank_balance.balance.liability_shares.value != 0,
            MarginfiError::AccountIllegalPostLiquidationState
        );
        ...
        check!(
            total_weighted_assets <= total_weighted_liabilities,
            MarginfiError::AccountIllegalPostLiquidationState
        );
        ...
    }
```

The first check relies on another strong assumption: A bank balance might only be in one of the following states:

- `deposit_shares > 0 && liability_shares == 0`
- `deposit_shares == 0 && liability_shares == 0`
- `deposit_shares == 0 && liability_shares > 0`

This is mostly true due to the implementation of deposit and withdraw operations:

1. When depositing tokens in a bank balance, the liability_shares are repaid before the deposit_shares field is increased.
2. When borrowing tokens from a bank balance, the deposit_shares are used until they reach zero, before increasing the liability_shares.

In theory, a bank balance should never reach a state where both deposit_shares and liability_shares are greater than zero.

In practice, this is not true, due to precision errors in deposit and withdraw operations, when calculating the value of shares corresponding to the deposited/withdrawn tokens.

For example, those precision errors might result in a bank balance in the following state:

```
liquidatee balance: {
    deposit_shares: 144684.000000000000004,
    liability_shares: 0.000000000000004
}
```

Meaning that the first check is not enough to prevent the liquidator to steal deposit shares from the liquidatee's balance.

Proof of Concept

1. Suppose that the liquidator executes the LendingAccountLiquidate instruction with AssetBank == LiabBank
2. Suppose that the liquidatee balance for that bank is {deposit_shares > 0, liability_shares: 4×10^{-15}} and that the liquidatee's account is in an unhealthy state.
3. The liquidation logic is executed as follows:
   (a) $0.975 \cdot asset\_quantity$ tokens are borrowed from the liquidator balance
   (b) $asset\_quantity$ tokens are deposited in the liquidatee balance
   (c) $asset\_quantity$ tokens are withdrawn from the liquidatee balance
   (d) $0.950 \cdot asset\_quantity$ tokens are repaid to the liquidator balance
4. Liquidatee account is still in an unhealthy state and the liability_shares are not zero, this means that the post liquidation health validation does not fail.

In the end this resulted in a loss of $0.05 \cdot asset\_quantity$ tokens for the liquidatee and a gain of $0.025 \cdot asset\_quantity$ for the liquidator.

Remediation

The way the checks are performed should be refactored to:

1. Explicitly check the liquidatee's account health both before and after executing the liquidation logic, to prevent the liquidatee from going into a worse state.
2. Fix the comparison with zero in the check [1] to account for precision errors, ideally by using a threshold value instead of a strict equality.

Patch

Resolved in 9b915a0.

OS-MRG-ADV-01 [med] [resolved] | Improper Token Accounting

Description

The utilization ratio is calculated as the total amount of borrowed tokens divided by total deposited tokens. This ratio should not exceed 1.

In practice, when a borrow operation is performed, there is no check to ensure that the total amount of tokens borrowed does not exceed the total amount of tokens deposited. This is because the assumption is made that the total amount of tokens deposited in the bank is the same as the amount of tokens deposited in the `liquidity_vault` token account.

src/instructions/marginfi_account.rs RUST

```
    pub fn bank_withdraw(ctx: Context<BankWithdraw>, amount: u64)
        ...
        bank_account.account_borrow(I80F48::from_num(amount))?;
        bank_account.withdraw_spl_transfer(
            amount,
            Transfer {
                from: bank_liquidity_vault.to_account_info(),
                to: destination_token_account.to_account_info(),
                authority:
    ↪   bank_liquidity_vault_authority.to_account_info(),
            },
        ...
```

This assumption does not hold true in practice, as anyone could directly transfer tokens to the `liquidity_vault` token account, opening up the possibility to manipulate the utilization ratio, by borrowing more tokens than the total amount of tokens deposited.

Proof of Concept

Following is a proof of concept that demonstrates how the utilization ratio could exceed one:

- Initially the bank state is { `deposit_share_value`: 1.0, `liability_share_value`: 1.0, `deposit_shares`: 100, `liability_shares`: 0 }
- `bank_liquidity_vault` Token Account has 100 tokens
- Attacker sends 100 tokens to the the bank_liquidity_vault Token Account
- `bank_liquidity_vault` now has 200 tokens but the `deposit_shares` are still 100
- Attacker borrows 200 tokens from the bank

In the end this would result in an utilization ratio of $200 \div 100 = 2$ which is greater than 1.

Remediation

Explicitly check that the total value of tokens borrowed does not exceed the total value of tokens deposited in the bank, every time those values are updated.

Patch

Resolved in e13d1cf by adding the following check throughout the codebase:

src/state/marginfi_group.rs DIFF

```
+   pub fn check_utilization_ratio(&self) -> MarginfiResult {
+       let total_deposits =
    ↪  self.get_deposit_amount(self.total_deposit_shares.into())?;
+       let total_liabilities =
    ↪  self.get_liability_amount(self.total_liability_shares.into())?;
+
+       check!(
+           total_deposits >= total_liabilities,
+           crate::prelude::MarginfiError::IllegalUtilizationRatio
+       );
+
+       Ok(())
+   }
```

OS-MRG-ADV-02 [low] [resolved] | Incorrect Fees Accounting

Description

There are various usages of the function I80F48::to_num(). This function does not check for truncation, even when overflow-checks are enabled in release build. The most impactful usage of this unsafe function is in `accrue_interest`.

When collecting fees after interest accrual is performed, the fee amount to be transferred is truncated from a potentially 80-bit number to a 64-bit number.

The issue occurs in the `accrue_interest` function:

src/state/marginfi_group.rs RUST

```
    pub fn accrue_interest(&mut self, clock: &Clock) ->
    ↪  MarginfiResult<(u64, u64)> {
        ...
        fees_collected = fees_collected
            .checked_add(self.fee_transfer_remainder.into())
            .ok_or_else(math_error!())?;
        ...
        insurance_collected = insurance_collected
            .checked_add(self.insurance_transfer_remainder.into())
            .ok_or_else(math_error!())?;
        ...
        Ok((fees_collected.to_num(), insurance_collected.to_num()))
    }
```

Both `fees_collected` and `insurance_collected` are `I80F48` numbers. Directly converting them to 64-bit numbers using `I80F48::to_num` will truncate the numbers to 64 bits, which might result in a small amount of fees being transferred to the corresponding Vaults.

Proof of Concept

The following is a proof of concept that demonstrates how the issue can be triggered:

1. Suppose that `fees_collected` is equal to 2^{64}
    - Note that even though the maximum token supply for a specific token is constrained to be less than 2^{64}, this is still possible, because `deposit_share_value` might be less than one.
2. `fees_collected` will be truncated to zero
3. Only zero tokens will be transferred to the `fee_vault` token account

Remediation

Use the `checked_to_num` safe function instead of `to_num`, to ensure that truncation never occurs. Explicitly check that the fees collected are less than 2^{64} before truncating to 64 bits, splitting the collection of fees in more than one transaction when necessary.

Patch

This has been fixed in 0fa4765.

More specifically, the interest accrual got separated by the fees collection logic.

To achieve this, the new instruction `LendingPoolCollectFees` has been introduced to collect fees, which correctly restricts the amount of fees that can be collected to be less than the available amount of tokens in the liquidity vault.

src/instructions/marginfi_group.rs RUST

```
pub fn lending_pool_collect_fees(ctx:
↪ Context<LendingPoolCollectFees>)
    ...
    let mut available_liquidity =
↪ I80F48::from_num(liquidity_vault.amount);
    let (insurance_fee_transfer_amount,
↪ new_outstanding_insurance_fees) = {
        let outstanding_fees =
↪ I80F48::from(bank.collected_insurance_fees_outstanding);
        let transfer_amount = min(outstanding_fees,
↪ available_liquidity).int();

        (transfer_amount.int(), outstanding_fees - transfer_amount)
    };
    ...
```

Unsafe usages of `to_num` were fixed in 366ffa2.

OS-MRG-ADV-03 [low] [resolved] | Incorrect Order of Operations in Liquidation

Description

The current order of operations in the liquidation process is not optimal and might result in unintended consequences. Specifically, reducing the balance of the liquidator first may cause the utilization ratio check to fail. The operations are currently performed in the following order:

1. Decrease the liability token balance of the liquidator to pay off the debt of the liquidatee
2. Increase the asset token balance of the liquidator in exchange for the liability tokens
3. Decrease the asset token balance of the liquidatee to pay off the collateral owed to the liquidator
4. Increase the liability token balance of the liquidatee, which represents the amount of debt paid off by the liquidator

Remediation

To ensure that unexpected behaviors do not occur, reorder the operations in the liquidation process so that the deposit step (2) for the liquidator's balance is performed before the borrow step (1). This will guarantee that the bank's utilization ratio check does not fail.

Patch

Resolved in d69716a.

05 | General Findings

Here, we present a discussion of general findings during our audit. While these findings do not present an immediate security impact, they represent antipatterns and could lead to security issues in the future.

ID	Description
OS-MRG-SUG-00	If the bank socializes a loss greater than the existing assets, the share value becomes negative.
OS-MRG-SUG-01	The liquidator and liquidatee should not be allowed to be the same account.
OS-MRG-SUG-02	The state of the bank with respect to interest accumulation is not enforced to be recent.
OS-MRG-SUG-03	The bank deserialization is implemented incorrectly, as the anchor's derived 8-byte long discriminator is not being skipped.
OS-MRG-SUG-04	Consider stricter validation of collateral and liability weights for liquidation logic soundness.

OS-MRG-SUG-00 | Unchecked Loss Socialization

Description

When an account goes bankrupt, the protocol socializes the loss to the other accounts. This loss socialization is done by decreasing the share value. If the loss is greater than the existing assets, the share value becomes negative.

RUST

```rust
/// Socialize a loss `loss_amount` among depositors,
/// the `total_deposit_shares` stays the same, but total value of
↪ deposits is
/// reduced by `loss_amount`;
pub fn socialize_loss(&mut self, loss_amount: I80F48) ->
↪ MarginfiResult {
    let total_deposit_shares: I80F48 =
↪ self.total_deposit_shares.into();
    let old_deposit_share_vaule: I80F48 =
↪ self.deposit_share_value.into();

    let new_share_value = total_deposit_shares
        .checked_mul(old_deposit_share_vaule)
        .ok_or_else(math_error!())?
        .checked_sub(loss_amount)
        .ok_or_else(math_error!())?
        .checked_div(total_deposit_shares)
        .ok_or_else(math_error!())?;

    self.deposit_share_value = new_share_value.into();

    Ok(())
}
```

Remediation

Explicitly check for the loss socialization amount to be less than the existing assets.

OS-MRG-SUG-01 | Tighter Liquidation Access Control

Description

The protocol currently allows the liquidator and liquidatee to be the same during a liquidation process. While health checks currently prevent this from ever being a concern, it might make sense to explicitly rule out this condition as a defense in depth measure.

Remediation

Implement a check in the `LendingAccountLiquidate` instruction to enforce that the liquidator and liquidatee are different accounts.

OS-MRG-SUG-02 | Outdated Bank Interest

Description

There are various places where the protocol does not enforce that the state of the bank is recent before performing various operations. For example, when creating positions, users might be able to accrue slightly more interest than intended because the account isn't up to date on position creation.

In practice, this is unlikely to be an issue if the crank is run regularly.

Remediation

Enforce that the state of the bank is updated before performing health checks.

Fixed in commit ee99fa2.

src/state/marginfi_account.rs DIFF

```
    pub fn bank_deposit(ctx: Context<BankDeposit>, amount: u64) ->
    ↪  MarginfiResult {
        ...
+       bank_loader.load_mut()?.accrue_interest(&Clock::get()?)?;
        ...
    }

    pub fn bank_withdraw(ctx: Context<BankWithdraw>, amount: u64) ->
    ↪  MarginfiResult {
        ...
+       bank_loader.load_mut()?.accrue_interest(&Clock::get()?)?;
        ...
    }

    pub fn lending_account_liquidate(
        ...
+       {
+           let clock = Clock::get()?;
+           ctx.accounts
+               .asset_bank
+               .load_mut()?
+               .accrue_interest(&clock)?;
+           ctx.accounts.liab_bank.load_mut()?.accrue_interest(&clock)?;
+       }
```

OS-MRG-SUG-03 | Incorrect Bank Deserialization

Description

The function `Bank::load_from_account_info` attempts to manually implement the deserialization of the Bank struct.

src/state/marginfi$_{g}roup.rs RUST

```rust
    #[inline]
    pub fn load_from_account_info(bank_account_ai: &AccountInfo) ->
    ↪  MarginfiResult<Self> {
        let bank =
    ↪  *bytemuck::from_bytes::<Bank>(&bank_account_ai.data.borrow());
        Ok(bank)
    }
```

However, the function does not account for the 8-byte long discriminator added by the anchor's `#[account]` macro, resulting in incorrect deserialization.

This function is currently unused, although it is important to this issue to avoid any potential mistakes in the future.

Remediation

Remove the function `Bank::load_from_account_info` and use the deserialization utilities provided by anchor, such as AccountDeserialize::try_deserialize.

Patch

Fixed in commit 9444194.

OS-MRG-SUG-04 | Weak Weight Validation

Description

The liquidation logic relies on an implicit relationship between collateral and liability weights.

src/instructions/marginfi_account.rs RUST

```
/// The fundamental idea behind the risk model is that the liquidation
    ↪ always leaves the liquidatee account in better health than
    ↪ before.
/// This can be achieved by ensuring that for each token the liability
    ↪ has a LTV > 1, each collateral has a risk haircut of < 1,
```

However, this relationship is not explicitly enforced at the protocol level.

Remediation

Enforce that the $collateral\_weight$ should be less than $(1 - (liquidation\_insurance\_fee + liquidation\_liquidator\_fee)) \cdot liability\_weight$ for any given pair of collateral and liabilities.

A | Vulnerability Rating Scale

We rated our findings according to the following scale. Vulnerabilities have immediate security implications. Informational findings can be found in the General Findings section.

Critical	Vulnerabilities that immediately lead to loss of user funds with minimal preconditions Examples: • Misconfigured authority or access control validation • Improperly designed economic incentives leading to loss of funds
High	Vulnerabilities that could lead to loss of user funds but are potentially difficult to exploit. Examples: • Loss of funds requiring specific victim interactions • Exploitation involving high capital requirement with respect to payout
Medium	Vulnerabilities that could lead to denial of service scenarios or degraded usability. Examples: • Malicious input that causes computational limit exhaustion • Forced exceptions in normal user flow
Low	Low probability vulnerabilities which could still be exploitable but require extenuating circumstances or undue risk. Examples: • Oracle manipulation with large capital requirements and multiple transactions
Informational	Best practices to mitigate future security risks. These are classified as general findings. Examples: • Explicit assertion of critical internal invariants • Improved input validation

B | Procedure

As part of our standard auditing procedure, we split our analysis into two main sections: design and implementation.

When auditing the design of a program, we aim to ensure that the overall economic architecture is sound in the context of an on-chain program. In other words, there is no way to steal funds or deny service, ignoring any chain-specific quirks. This usually requires a deep understanding of the program's internal interactions, potential game theory implications, and general on-chain execution primitives.

One example of a design vulnerability would be an on-chain oracle that could be manipulated by flash loans or large deposits. Such a design would generally be unsound regardless of which chain the oracle is deployed on.

On the other hand, auditing the implementation of the program requires a deep understanding of the chain's execution model. While this varies from chain to chain, some common implementation vulnerabilities include reentrancy, account ownership issues, arithmetic overflows, and rounding bugs.

As a general rule of sum, implementation vulnerabilities tend to be more "checklist" style. In contrast, design vulnerabilities require a strong understanding of the underlying system and the various interactions: both with the user and cross-program.

As we approach any new target, we strive to get a comprehensive understanding of the program first. In our audits, we always approach targets with a team of auditors. This allows us to share thoughts and collaborate, picking up on details that the other missed.

While sometimes the line between design and implementation can be blurry, we hope this gives some insight into our auditing procedure and thought process.