Mango

Mango v4

Audit

Presented by:

OtterSec contact@osec.io

Nicola Vela nick0ve@osec.io

Harrison Green hgarrereyn@osec.io

Contents

Appendices

01 | Executive Summary

Overview

Mango engaged OtterSec to perform an assessment of the `mango-v4` program. This assessment was conducted between February 22nd and March 24th, 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 April 2nd, 2023.

Key Findings

Over the course of this audit engagement, we produced 8 findings total.

In particular, we found an issue related to `HealthRegion`, which allows multiple instructions to be executed without performing health checks. This may allow malicious users to create debt and immediately liquidate it in the same transaction, stealing funds from the protocol (OS-MNG-ADV-00). We also noted an issue with token registrations that may lead to a denial of service. (OS-MNG-ADV-01).

Additionally, we discovered an issue in how the protocol handles USD to USDC conversions (OS-MNG-SUG-01). We also made recommendations about general quality of life issues (OS-MNG-SUG-05) and returning unclear errors to the users (OS-MNG-SUG-03).

Overall, we commend the Mango 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/blockworks-foundation/mango-v4. This audit was performed against commit `5c7a2e3`.

Mango-v4 is a decentralized exchange built on Solana, offering a variety of on-chain trading options for cryptocurrencies.

Users can build accounts with a variety of positions, while every position has to be overcollateralized so that the risk associated with margin trading can be managed by the protocol through permissionless liquidations.

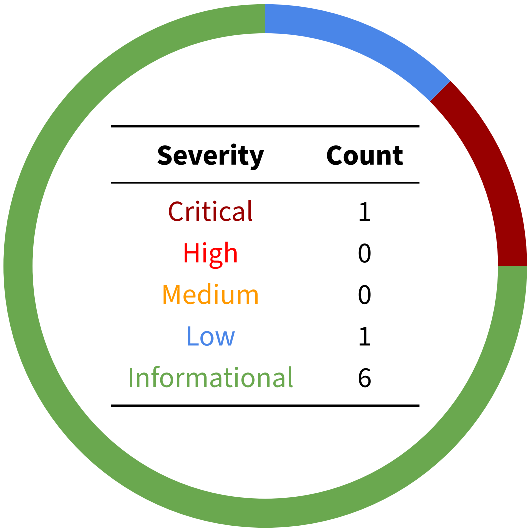
Its key features include:

1. Borrowing and lending capabilities, offering users to lend and earn interest based on how many users borrow.
2. Perpetual futures trading across a variety of assets through its own orderbook.
3. Spot trading through Openbook DEX.

03 | Findings

Overall, we reported 8 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	0
Low	1
Informational	6

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-MNG-ADV-00	Critical	Resolved	Malicious users may exploit the lack of access controls within `HealthRegions` to generate bad debt and immediately liquidate it within a single transaction.
OS-MNG-ADV-01	Low	Resolved	A token may be registered with a reserved value of `token_index`, leading to user token accounts being interpreted as deactivated.

OS-MNG-ADV-00 [crit] | Inadequate Access Control Inside HealthRegions

Description

HealthRegion allows multiple instructions to be executed without performing health checks, thereby reducing gas costs, as long as the account remains in a healthy position at the end. However, this feature also introduces new attack surfaces. For instance, an attacker may bypass the limitations of the FlashLoan implementation, which prevents users from calling Mango after receiving a flash loan.

To execute a flash loan without using FlashLoanBegin/Start, an attacker can:

1. Borrow any quantity of tokens within a HealthRegion, which would render their account in an unchecked and unhealthy state.
2. Conduct certain operations using the borrowed tokens.
3. Deposit the tokens back to restore their account to a healthy state.

Furthermore, rather than returning the borrowed tokens to restore the health of the account, an attacker may exploit the TokenLiqBankruptcy instruction to liquidate the bad debt. This enables the attacker to force the protocol to cover the cost of the liquidation, resulting in the unauthorized appropriation of funds.

Proof of Concept

To demonstrate how this attack can be carried out, let us consider a scenario where the attacker has two accounts: A and B. Account A will serve as the liquidatee, while Account B will be the liquidator. The attacker may execute a transaction with the following instructions:

1. HealthRegionBegin(A) - pre_init_health is zero.
2. A.TokenWithdraw(amt) - A where amt is the amount of tokens to be borrowed from the liab_bank. As A is within the HealthRegion, it can proceed into a liquidable stage without restriction.
3. B.TokenLiqBankruptcy(A) - B liquidates A through the token_liq_bankruptcy instruction. The protocol repays A's liabilities, and post_init_health becomes zero.
4. HealthRegionEnd(A).

This results in the attacker stealing amt tokens from the bank.

Remediation

Harden the current HealthRegion implementation to prevent liquidations and withdrawals from being executed inside it. This can be executed by leveraging the existing instruction introspection logic.

Patch

Resolved in b22a1e7 by explicitly allowing only the instructions necessary to interact with the perp and spot markets inside `HealthRegions`.

DIFF

```
+   let allowed_inner_ix = [
+       crate::instruction::PerpCancelAllOrders::discriminator(),
+       crate::instruction::PerpCancelAllOrdersBySide::discriminator(),
+       crate::instruction::PerpCancelOrder::discriminator(),
+
     ↪  crate::instruction::PerpCancelOrderByClientOrderId::discriminator(),
+       crate::instruction::PerpPlaceOrder::discriminator(),
+       crate::instruction::PerpPlaceOrderPegged::discriminator(),
+       crate::instruction::Serum3CancelAllOrders::discriminator(),
+       crate::instruction::Serum3CancelOrder::discriminator(),
+       crate::instruction::Serum3PlaceOrder::discriminator(),
+       crate::instruction::Serum3SettleFunds::discriminator(),
+       crate::instruction::Serum3SettleFundsV2::discriminator(),
+   ];
```

OS-MNG-ADV-01 [low] | Inadequate Sanitization Of Token Indexes

Description

When registering a token, the protocol does not adequately check the provided `token_index`. While the value `TokenIndex::MAX` is reserved to represent a disabled `TokenPosition`, it is not explicitly disallowed from being registered. As a result, if a token is registered with `TokenIndex::MAX`, all user token accounts for that token will be interpreted as deactivated, and using them will result in an error.

RUST

```rust
impl Default for TokenPosition {
    ...
        TokenPosition {
            ...
            token_index: TokenIndex::MAX,
            ...
        }
    ...
}
```

This may lead to all user token accounts for this token being interpreted as deactivated, and the attempt to trade them will result in an error.

Moreover, there is inconsistent behaviour between `is_active_for_token` and `is_active` in this context. Specifically, when `token_index == TokenIndex::MAX`, then `is_active_for_token` returns true, while `is_active` returns false.

RUST

```rust
impl TokenPosition {
    ...
    pub fn is_active(&self) -> bool {
        self.token_index != TokenIndex::MAX
    }

    pub fn is_active_for_token(&self, token_index: TokenIndex) ->
↪   bool {
        self.token_index == token_index
    }
    ...
}
```

Similar behaviour occurs for the Serum3MarketIndex and PerpMarketIndex indices which also have a reserved MAX value that is reserved for deactivated entries.

Remediation

Disallow the reserved value TokenIndex::MAX for token registration. Additionally, modify the inconsistent behaviour between is_active_for_token and is_active.

DIFF

```diff
    pub fn token_register(
        ...
        token_index: TokenIndex,
        ...
    ) -> Result<()> {
        ...
+       require_neq!(token_index, TokenIndex::MAX);
        ...
    }

    pub fn token_register_trustless(
        ...
        token_index: TokenIndex,
        ...
    ) -> Result<()> {
        ...
+       require_neq!(token_index, TokenIndex::MAX);
        ...
    }

    pub fn is_active_for_token(&self, token_index: TokenIndex) -> bool {
-       self.token_index == token_index
+       self.token_index == token_index && self.token_index !=
     ↪  TokenIndex::MAX
    }
```

Patch

Resolved in 99360e6.

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-MNG-SUG-00	Admin-only instructions with incorrect parameters may lead to unexpected behaviour.
OS-MNG-SUG-01	The protocol assumes that one USDC is always pegged to one USD, which may not always be true.
OS-MNG-SUG-02	`fill_from_str` uses 31 bytes out of 32 bytes.
OS-MNG-SUG-03	Multiple instances of unwrapping may cause panic for users.
OS-MNG-SUG-04	Named constants should be used instead of magic numbers to improve code clarity.
OS-MNG-SUG-05	The use of different oracle prices at different time periods may lead to misleading net deposit calculations.

OS-MNG-SUG-00 | Improve Safety Checks On Admin-Only Instructions

Description

Various admin-gated instructions to configure the protocol do not perform adequate validation on the parameters supplied, potentially resulting in future vulnerabilities. While this may not pose an immediate security risk, it leaves the system open to exploitation in the future.

One example is the `init_base_asset_weight` parameter for editing a perp market using the `perp_edit_market` instruction. This parameter must be positive; however, this requirement is not enforced when creating the perp market through the `perp_create_market` instruction.

```rust
    pub fn perp_create_market(
            ...
            init_base_asset_weight: f32,
            ...
    ) -> Result<()> {
            ...
            let mut perp_market =
 ↪  ctx.accounts.perp_market.load_init()?;
            *perp_market = PerpMarket {
            ...
            init_base_asset_weight:
 ↪  I80F48::from_num(init_base_asset_weight),
            ...
            };
    }

    pub fn perp_edit_market(
            ...
            init_base_asset_weight_opt: Option<f32>,
            ...
    ) -> Result<()> {
            ...
            require_gte!(
                    init_base_asset_weight,
                    0.0,
                    MangoError::InitAssetWeightCantBeNegative
            );
            ...
    }
```

Remediation

Implement additional rigorous validation checks on the parameters supplied to admin-only instructions to prevent future vulnerabilities. Specifically, for the `perp_create_market` instruction, ensure that the `init_base_asset_weight` parameter is positive.

Patch

Better checks will be implemented gradually.

OS-MNG-SUG-01 | Broken Assumption Of USDC Peg To USD

Description

The protocol assumes that one USDC is always pegged to one USD. However, in exceptional events such as market instability or other macroeconomic factors, the peg may break down, and the value of USDC may deviate from one USD.

To determine the value of USDC/USD, the protocol employs a `StubOracle` that sets the price at one. When it comes to Token/USDC pricing, the protocol utilizes Token/USD oracles and derives the USDC pricing directly from the USD pricing, without any intermediate conversion.

This approach may result in potential loss of funds for users in situations where the USDC peg is not stable, and arbitrage bots are taking advantage of the price discrepancy between USDC and USD.

Remediation

Implement a more robust and adaptable pricing mechanism for USDC that takes into account the possibility of a broken USDC peg.

Patch

The issue is acknowledged and a better solution for removing the peg is being developed.

OS-MNG-SUG-02 | Wrong Input Data Length Check For Fill_From_Str

Description

`fill_from_str` is frequently used throughout the codebase to copy string data into 32-byte-sized buffers. However, it requires input data to be < 32 bytes instead of <= 32 bytes. As a result, the byte at index 31 is never used.

Remediation

Require input data to be less than or equal to 32 bytes.

Patch

Resolved in 658a220.

DIFF

```diff
  pub fn fill_from_str<const N: usize>(name: &str) -> Result<[u8; N]> {
    let name_bytes = name.as_bytes();
-   require!(name_bytes.len() < N, MangoError::SomeError);
+   require!(name_bytes.len() <= N, MangoError::SomeError);
    let mut name_ = [0u8; N];
    name_[..name_bytes.len()].copy_from_slice(name_bytes);
    Ok(name_)
  }
```

OS-MNG-SUG-03 | Replace Panics With Descriptive Errors

Description

The codebase contains multiple instances of unwrapping, which may cause panics that are difficult for users to understand. For example, `token_add_bank` may cause a panic when invoked while the maximum number of banks have already been added.

Remediation

Use descriptive errors that provide clear information about what caused the error.

Patch

The issue is acknowledged; more descriptive errors will be integrated into the codebase gradually.

OS-MNG-SUG-04 | Improve Clarity Of Token Index Check

Description

In `token_register_trustless`, there is a check to ensure that the token index is not equal to zero using the `require_neq!` macro. However, this check may be unclear to developers and difficult to understand without additional context.

Remediation

Use the `QUOTE_TOKEN_INDEX` constant, which is more descriptive of the intended behaviour.

```diff
pub fn token_register_trustless(
    ctx: Context<TokenRegisterTrustless>,
    token_index: TokenIndex,
    name: String,
) -> Result<()> {
-   require_neq!(token_index, 0);
+   require_neq!(token_index, QUOTE_TOKEN_INDEX);
```

Patch

Resolved in 99360e6.

OS-MNG-SUG-05 | Net Deposits Calculation May Be Misleading

Description

When depositing and withdrawing tokens into a bank, the `net_deposits` field of the user's `MangoAccount` is updated by adding or subtracting the USD value of the tokens deposited or withdrawn.

This value is calculated by multiplying the amount of tokens by the oracle price at the time of the operation. However, the oracle price used to calculate the value of tokens deposited or withdrawn may not be the same as the oracle price used in the previous operation. This may lead to misleading `net_deposits` calculations.

Consider the following scenario:
A user deposits 1 SOL when its value is $100 and later withdraws 1 SOL when its value is $10. In this case, the `net_deposits` field would display a result of $90, despite the user not having made any actual deposits into the bank.

Remediation

Ensure that the `net_deposits` field is zero to avoid such discrepancies.

Patch

This feature is working as intended although documentation will be improved to clarify the intent.

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.