Audit Report

Mars Perps

v1.0

December 4, 2024

Table of Contents

License

Disclaimer

This audit has been performed by

Oak Security GmbH

https://oaksecurity.io/
info@oaksecurity.io

Introduction

Purpose of This Report

Oak Security GmbH has been engaged by Mars Protocol Foundation to perform a security audit of the Mars Perps codebase.

The objectives of the audit are as follows:

1. Determine the correct functioning of the protocol, in accordance with the project specification.
2. Determine possible vulnerabilities, which could be exploited by an attacker.
3. Determine smart contract bugs, which might lead to unexpected behavior.
4. Analyze whether best practices have been applied during development.
5. Make recommendations to improve code safety and readability.

This report represents a summary of the findings.

As with any code audit, there is a limit to which vulnerabilities can be found, and unexpected execution paths may still be possible. The author of this report does not guarantee complete coverage (see disclaimer).

Codebase Submitted for the Audit

The audit has been performed on the following target:

Repository	https://github.com/mars-protocol/core-contracts
Commit	`60bed0b22e3fdb3bec4762c51053385a0c101529`
Scope	This audit covers changes since our previous audit, which was performed at commit `01364db126c077abd6dc70799692de4551fc08d8`. All contracts except `contracts/mock-credit-manager`, `contracts/mock-health`, and `contracts/mock-incentives` were in scope.
Fixes verified at commit	`526798c0a2cb9da958bf4f4538faacc48ee45299`

	Note that only fixes to the issues described in this report have been reviewed at this commit. Any further changes such as additional features have not been reviewed.

Methodology

The audit has been performed in the following steps:

1. Gaining an understanding of the code base's intended purpose by reading the available documentation.
2. Automated source code and dependency analysis.
3. Manual line-by-line analysis of the source code for security vulnerabilities and use of best practice guidelines, including but not limited to:
    a. Race condition analysis
    b. Under-/overflow issues
    c. Key management vulnerabilities
4. Report preparation

Functionality Overview

Mars Protocol is a multi-chain money market built on CosmWasm that leverages the Cosmos ecosystem's interoperability and composability. The changes covered by this audit introduce support for perps and trigger orders.

How to Read This Report

This report classifies the issues found into the following severity categories:

Severity	Description
Critical	A serious and exploitable vulnerability that can lead to loss of funds, unrecoverable locked funds, or catastrophic denial of service.
Major	A vulnerability or bug that can affect the correct functioning of the system, lead to incorrect states or denial of service.
Minor	A violation of common best practices or incorrect usage of primitives, which may not currently have a major impact on security, but may do so in the future or introduce inefficiencies.
Informational	Comments and recommendations of design decisions or potential optimizations, that are not relevant to security. Their application may improve aspects, such as user experience or readability, but is not strictly necessary. This category may also include opinionated recommendations that the project team might not share.

The status of an issue can be one of the following: **Pending, Acknowledged**, **Partially Resolved**, or **Resolved**.

Note that audits are an important step to improving the security of smart contracts and can find many issues. However, auditing complex codebases has its limits and a remaining risk is present (see disclaimer).

Users of the system should exercise caution. In order to help with the evaluation of the remaining risk, we provide a measure of the following key indicators: **code complexity**, **code readability**, **level of documentation**, and **test coverage**. We include a table with these criteria below.

Note that high complexity or low test coverage does not necessarily equate to a higher risk, although certain bugs are more easily detected in unit testing than in a security audit and vice versa.

Code Quality Criteria

The auditor team assesses the codebase's code quality criteria as follows:

Criteria	Status	Comment
Code complexity	**Medium**	The audited codebase is large, with several external dependencies/integrations, which increases the complexity.
Code readability and clarity	**High**	-
Level of documentation	**High**	-
Test coverage	**High**	92.74% test coverage

Summary of Findings

No	Description	Severity	Status
1	Incorrect migration guard unlock causes failed migration and loss of rewards	Critical	Resolved
2	Users will lose accrued rewards when withdrawing liquidity	Critical	Resolved
3	Withdrawal may fail due to out-of-gas error when iterating positions	Major	Resolved
4	Missing account ID validation allows removal of other user's trigger orders	Major	Resolved
5	Liquidation will fail for zero amounts	Major	Resolved
6	Instantiating too many markets causes a denial of service	Minor	Resolved
7	Potential overflow error for tokens with large decimals	Minor	Resolved
8	Risk management framework neglects correlated, triggered and strategic orders	Informational	Acknowledged
9	Grief attack vector allows for minor market manipulations	Informational	Acknowledged
10	Duplicate storage load consumes unnecessary gas	Informational	Resolved

Detailed Findings

1. Incorrect migration guard unlock causes failed migration and loss of rewards

Severity: Critical

As part of the migration process, the incentives contract's `USER_UNCLAIMED_REWARDS` and `USER_ASSET_INDICES` states are required to be migrated via the `migrate_user_unclaimed_rewards` and `migrate_user_asset_indices` functions defined in `contracts/incentives/src/migrations/v2_0_1.rs:90-95`. To prevent storage entries from being committed during migration, a `MIGRATION_GUARD` state is introduced to temporarily lock the `BalanceChange` and `ClaimRewards` messages from being executed. These messages can only be executed when the guard is unlocked, which indicates the migration has been completed.

The issue is that both the `migrate_user_unclaimed_rewards` and `migrate_user_asset_indices` function each unlocks the guard when it has completed its migration without considering that the other one may not have been completed (see `contracts/incentives/src/migrations/v2_0_1.rs:139-142` and `199-202`). This causes failures due to the `MIGRATION_GUARD.assert_locked` requirement and allows the `BalanceChange` and `ClaimRewards` messages to commit state entries.

For example, if the `migrate_user_unclaimed_rewards` function has completed its migration, the old `USER_ASSET_INDICES` state entries will not be migrated. When users deposit funds to the red bank contract, their asset index will be zero, causing them to receive more rewards than intended. Since the migration functions are permissionless, attackers can exploit this to steal funds from the protocol, causing a loss of funds.

Recommendation

We recommend only unlocking the guard if all the `USER_UNCLAIMED_REWARDS` and `USER_ASSET_INDICES` state migrations have been completed.

Status: Resolved

2. Users will lose accrued rewards when withdrawing liquidity

Severity: Critical

To withdraw liquidity from the perps contract, users must request their shares to be unlocked and wait for the cooldown period to elapse. As part of the reward distribution logic, a `BalanceChange` message will be dispatched to compute rewards in the incentives contract. The reward computation logic depends on the passed `user_amount` parameter in `contracts/perps/src/vault.rs:231`, representing the number of shares the user owns.

The issue is that the `withdraw` function supplies the `user_amount` parameter with the user's `DEPOSIT_SHARES` state value in `contracts/perps/src/vault.rs:213`. This is incorrect because the `unlock` function has already decreased the user's `DEPOSIT_SHARES` state value in `contracts/perps/src/vault.rs:143`, meaning that the amount of the unlocked shares needs to be added back to compute rewards correctly.

Consequently, users' share amount will be incorrectly computed to a lesser amount, causing them to lose rewards they staked during this period.

Recommendation

We recommend computing the `user_amount` parameter to include the number of unlocked shares. This can be achieved by adding the `user_shares_before` and `total_unlocked_shares` variables together.

Status: Resolved

3. Withdrawal may fail due to out-of-gas error when iterating positions

Severity: Major

To withdraw liquidity from the perps contract, users must request their shares to be unlocked and wait for the cooldown period to elapse. When unlocking shares in `contracts/perps/src/vault.rs:147-157`, the `unlock` function appends a new entry of `UnlockState` to the `UNLOCKS` state without any cap limit.

This is problematic because the `withdraw` function will iterate over all the `UnlockState` entries in `contracts/perps/src/vault.rs:197` when users want to withdraw unlocked funds. If there are too many positions to iterate, the transaction will fail due to an out-of-gas error, preventing users from withdrawing their funds.

Recommendation

We recommend capping the number of `UnlockState` entries that can be added to the `UNLOCKS` state.

Status: Resolved

4. Missing account ID validation allows removal of other user’s trigger orders

Severity: Major

In `contracts/credit-manager/src/execute.rs:273-279`, the `DeleteTriggerOrder` action allows users to delete trigger orders. It does so by accessing the `TRIGGER_ORDERS` state from the provided `account_id` and `trigger_order_id` parameters and removing the order from storage.

The issue is that no validation ensures the supplied `account_id` equals the caller’s account ID from `contracts/credit-manager/src/execute.rs:124`. This allows malicious users to pass the `account_id` and `trigger_order_id` parameters as other users’ account and order ID values to remove their trigger order.

Recommendation

We recommend using the caller’s account ID as the `account_id` parameter.

Status: Resolved

5. Liquidation will fail for zero amounts

Severity: Major

In `contracts/credit-manager/src/liquidate_lend.rs:27-29` and `contracts/credit-manager/src/liquidate_astro_lp.rs:31-33`, liquidation will fail if the collateral balance is zero. This is problematic because if the requested collateral balance is zero and the account has a negative net value (indicating the account has bad debt), liquidation cannot be executed to prevent additional debt accrual.

Consequently, positions with negative net value may continue accruing bad debt, ultimately risking the protocol solvency.

This issue has been independently reported by the client.

Recommendation

We recommend allowing liquidation with zero collateral balance.

Status: Resolved

6. Instantiating too many markets causes a denial of service

Severity: Minor

In `contracts/perps/src/market.rs:605-647`, the `compute_total_pnl` function iterates over all the entries in the `MARKET_STATES` state. This function is called in several instances that require computing the vault's profit and loss amount, particularly in the deposit and withdrawal operations.

This is problematic because there is no restriction on how many markets can be instantiated. If there are too many markets to iterate, the transaction will fail due to an out-of-gas error, preventing users from depositing and withdrawing their funds.

We classify this issue as Minor since it can only be caused by the owner/admin.

Recommendation

We recommend capping the number of markets that can be instantiated.

Status: Resolved

7. Potential overflow error for tokens with large decimals

Severity: Minor

In `contracts/perps/src/market.rs:591-598`, the `increase_accumulators` function increases the market's total squared and multiplied positions based on the position size. However, the position size (indicated as `size.abs` variable) implements a `Uint128` integer type, as seen in `packages/types/src/signed_uint.rs:18`. This is problematic because if the token denom has a large number of decimals (e.g., 18 decimals), the computation will result in an overflow error because the integer is not sufficient to hold the value.

This issue has been independently reported by the client.

Recommendation

We recommend using a larger integer type (e.g., `Int256`) to prevent an overflow error.

Status: Resolved

8. Risk management framework neglects correlated, triggered and strategic orders

Severity: Informational

The current risk management framework makes a key assumption: all traders enter their positions simultaneously, and the market impact is computed as if traders close their positions independently of other positions. This assumption poses a risk, as real-world trading behaviors are often more complex:

1. Traders, especially those with larger holdings, tend to strategically engineer trades to trigger actions from other traders.
2. Some traders monitor the mempool and place conditional trades in the market based on this information.
3. The trading protocol itself provides mechanisms for placing conditional (so called trigger) orders.

The existing risk management framework does currently not account for scenarios involving conditional and dependent trades. A series of such trades can lead to a loss for the protocol.

Recommendation

We recommend extending the risk framework to account for correlated, triggered, and strategic orders.

Status: Acknowledged

The client states: “When estimating the risk parameters (Max OI and Max Skew) for each perp market, we made a set of very conservative assumptions representing the worst-case scenario for the vault, see the list below:

1. All open positions are on one side of the market. In this scenario, the protocol is effectively the sole counterparty to all positions.
2. The positions set is static. There are no arbitrageurs and traders do not close positions to contract the skew.
3. The perp market price extremely changes towards the skew direction over a short time horizon.
4. Compounding funding payments accumulated over the risk horizon are excluded.
5. Closing fees are excluded.

Specifically, we calibrate parameters assuming that the skew is at the theoretical maximum and the price extremely changed towards the skew direction if favor of users incurring losses for the vault. We ignored closing fees and funding fees. In reality, when the skew tends to the maximum, funding rates will rise significantly making user’s open positions unprofitable and encouraging closure (this is the key idea behind the calibration of our Max Funding Velocity risk parameter). At the same time, arbitrageurs will intervene to take profit from high funding rates and adjust the skew, whereas we assume zero arbitrage during the entire risk horizon.

It is very unlikely that these events will be observed simultaneously. Assuming these stress conditions, the risk parameters are conservatively estimated so as to limit the potential losses for the vault at the 30% level in the probabilistic sense. Assuming the worst-case scenario is a common practice in the absence of more accurate approaches. The only assumption that requires further consideration is the estimation of chances that two or more markets are under the extreme conditions mentioned above. However, we believe that the chances are low and to mitigate this risk, we use quite a conservative potential loss threshold for the vault. We plan to investigate this topic once the proper data history is available.

We believe that the resulting parameters are determined quite conservatively and they should cover other simplifications, so there is no need to further complicate the risk framework. Modeling different scenarios for closing order and investigating the market impact effect may be subject to stress testing of the whole system. This will require the development of a more sophisticated framework (at a user level)."

9. Grief attack vector allows for minor market manipulations

Severity: Informational

The trigger feature allows orders to be triggered if a certain on-chain metric is met – e.g., the relative price of an asset dropping below some threshold. The `delete_trigger_order` function in `contracts/credit-manager/src/trigger.rs:72` allows the deletion of such trigger orders without any time constraints.

There is a grief attack opportunity by users who watch the mempool and cancel their existing trigger orders whenever they observe that someone tries to execute that trigger order. There are costs associated with such griefing attacks, though, since the user loses transaction fees, and the canceling party incurs cancellation costs. Due to these costs, the impact of griefing attacks is limited.

However, there might be asymmetric benefits for the attacker, which might increase the severity of the issue, conditional on the state of the market. In trending markets, the risk for the user is that they believe in opening a very short-term trade to be a highly leveraged receiving position (e.g., at a turning point) but end up with a trade opened as a paying position, potentially at the risk of liquidation. Users could be protected against such attacks through slippage parameters, for example, asserting a minimum/maximum funding rate after the position is opened.

However, since we are not aware of any other perpetual swap protocols using such slippage parameters, we do not report this issue with higher severity. In addition, this is likely to affect primarily sophisticated traders that build their own infrastructure (e.g., to inspect and trigger triggerable orders). Nevertheless, if such infrastructure develops, the side effects (i.e., higher volatility and sudden drops in liquidity) could negatively affect a broad range of users.

Recommendation

There are various options to address this issue. We recommend trigger orders to have a validity, such that they cannot be cancelled before a specific point in time. Option two would be a two-step deletion process, where users must request their deletion at time T and can only execute the deletion after buffer x has passed at time T + x. Another option to prevent such types of attacks, together with possible sandwiching attack vectors that only target short-term manipulations of the funding rate, would be to introduce slippage parameters for the funding rate for all transactions.

Status: Acknowledged

The client states: “This system has many similarities to a CLOB (centralized limit order book). As with an order book, these orders cannot be guaranteed to be executed. If a participant wants to make a trade or action based on another participant's trigger order being executed (e.g. price or funding rate arbitrage), they should consider submitting the `execute_trigger_order` message in the same transaction bundle as their action message. The atomic nature of blockchains will ensure that if the trigger order is cancelled their transaction does not execute. Alternatively, they could wait until the trigger order is successfully executed before submitting their order. Otherwise, market participants should not operate under the assumption that the order is guaranteed to be executed.

Whilst in theory, the PVP-like behavior described could materialize, similar behavior affects all CLOB, and we do not see it having a significant impact on liquidity or volatility in production systems that have operated for years.

Furthermore, our system provides more guarantees than a CLOB can, given the atomic nature of blockchains described above.

Whilst some PVP between sophisticated actors in these types of systems should be expected, we do not think it will result in a negative impact on liquidity or increased volatility.”

10. Duplicate storage load consumes unnecessary gas

Severity: Informational

In `contracts/credit-manager/src/liquidate_lend.rs:22` and `47`, the `liquidate_lend` function loads the `RED_BANK` state from the storage twice. This results in unnecessary gas consumption and reduces code readability, as the subsequent storage read can be optimized by using a variable.

Similarly, the `close_all_perps` function in `contracts/credit-manager/src/perp.rs:136` and `162` suffers from the same issue where the `PERPS` state is loaded twice.

Recommendation

We recommend loading the state once and storing it in a variable to be used throughout the function.

Status: Resolved