Audit Report

Mars Red Bank Updates

v1.0

August 1, 2023

Table of Contents

License

Disclaimer

This audit has been performed by

Oak Security

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

Introduction

Purpose of This Report

Oak Security has been engaged by Delphi Labs Ltd. to perform a security audit of updates to the Mars Red Bank CosmWasm smart contracts.

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/red-bank
Commit	`7149f580bf6f7593d2ccaa4c09a2d63dc482d5ff`
Scope	In the scope of this audit were all changes and their integration since our previous Mars Outposts audit, which was performed on commit `62666cc07627ff41acda7196ca34ad8dbc1f1f8d`.

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 multichain money market built on CosmWasm that leverages the Cosmos ecosystem's interoperability and composability. The audit scope includes updates to Mars's Red Bank contracts since our previous audit.

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**, 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-High	-
Code readability and clarity	Medium-High	-
Level of documentation	High	The Mars team provided sufficient documentation including flow diagrams.
Test coverage	Medium-High	`cargo tarpaulin` reports a test coverage of 82.33%.

Summary of Findings

No	Description	Severity	Status
1	Missing denom validation when adding incentives could lead to insufficient funds error	Critical	Resolved
2	Incorrect calculation when simulating with multiple routes	Critical	Resolved
3	Utilization rate can be exploited to surpass `100%` for new markets	Critical	Resolved
4	Removing whitelisted denoms causes leftover rewards to get stuck	Major	Acknowledged
5	Owner cannot update minimum emission without consequences	Major	Resolved
6	Updating base denom would cause incorrect price results	Major	Acknowledged
7	Incentive rewards might be distributed to depositors outside the epoch period	Major	Acknowledged
8	Duplicated denoms might cause incorrect whitelist counts	Minor	Resolved
9	Oracle centralization risks with Fixed price sources	Minor	Acknowledged
10	Misconfiguring tolerance value to be higher than window size causes incorrect price reported	Minor	Resolved
11	Removing price sources in the oracle could stop reward collector operations	Minor	Acknowledged
12	Missing prerequisites check when adding a `Pyth` price source	Minor	Resolved
13	Overflow checks not enabled for release profile	Informational	Resolved
14	Funds in the swapper contract can be stolen	Informational	Acknowledged

Detailed Findings

1. Missing denom validation when adding incentives could lead to insufficient funds error

Severity: Critical

In `contracts/incentives/src/helpers.rs:95-100`, the `validate_incentive_schedule` function lacks validation to ensure that the sent denomination matches the incentive denomination.

This is problematic because it allows potential attackers to create an incentive schedule with a different denomination as long as the amount sent is correct.

Consequently, legitimate users may encounter difficulties in claiming their rightful rewards due to an insufficient funds error.

A test case reproducing this issue is provided in the Appendix in the `test_incorrect_denom_deposit` test case.

Recommendation

We recommend ensuring the sent denom is the expected incentive denom.

Status: Resolved

2. Incorrect calculation when simulating with multiple routes

Severity: Critical

In `contracts/oracle/wasm/src/helpers.rs:145-160`, the `add_route_prices` function loads the price source for the denom and multiplies it with the current price. This is incorrect because the third route asset's currency is not considered.

To illustrate, assume a price source is created with three route assets (e. g. `ATOM` => `MARS` => `OSMO`) while the base denom is configured as `USDC`. The first route asset will be simulated in Astroport, making the price currency `MARS`. After that, the function loads the `MARS` price from storage and multiplies it, resulting in the price currency becoming `USDC`.

The problem arises when multiplying with the third route asset's price, `OSMO`. When the price source is retrieved, the price currency is expected to be `OSMO`, not `USDC`. The function will calculate the price as follows:

$$price_{\frac{ATOM}{OSMO}} = price_{\frac{ATOM}{MARS}} \cdot price_{\frac{MARS}{USDC}} \cdot price_{\frac{OSMO}{USDC}}$$

This is incorrect, because the price currency after the second route asset multiplication (`MARS`) is not denominated as the third route asset's currency (`OSMO`).

Consequently, this discrepancy in the expected currency versus the actual currency introduces a flaw in the calculation process.

Recommendation

We recommend considering the currency of each route asset and ensuring alignment between the expected price currency and the actual price currency.

Status: Resolved

3. Utilization rate can be exploited to surpass `100%` for new markets

Severity: Critical

In `contracts/red-bank/src/interest_rates.rs:290-294`, the utilization rate of a market is determined by the ratio of total debt divided by total collateral. If a new market is instantiated with zero deposits, an attacker can inflate the utilization rate to steal funds from the contract.

An exemplary step-by-step attack follows:

1. `ATOM` market is instantiated.
2. The attacker becomes the first depositor and deposits `1 uatom`, increasing the market's total collateral.
3. The attacker donates `1000 ATOM` to the contract directly.
4. Using another address, the attacker deposits `10000 OSMO` as collateral.
5. Using another address, the attacker borrows the donated `1000 ATOM`, increasing the market's total debt.
6. The current utilization rate will exceed `100%` because the total debt is higher than the total collateral.
7. The attacker receives around `84000 ATOM` after 10 seconds due to the inflated utilization rate.

For more information on this type of attack, please refer to the Silo Finance Vulnerability Disclosure.

A test case reproducing this issue is provided in the Appendix in the `zero_deposit_poc` test case.

Recommendation

We recommend limiting the utilization rate to a maximum value of 100%.

Status: Resolved

4. Removing whitelisted denoms causes leftover rewards to get stuck

Severity: Major

In `contracts/incentives/src/contract.rs:173-180`, the `execute_update_whitelist` function removes emissions without handling the case of leftover rewards in the contract. This is problematic because any undistributed rewards cannot be withdrawn or refunded, causing a loss of funds for affected users.

Recommendation

We recommend refunding the excess rewards to the rewards collector contract.

Status: Acknowledged

5. Owner cannot update minimum emission without consequences

Severity: Major

In `contracts/incentives/src/contract.rs:187-198`, the `execute_update_whitelist` function allows the owner to add denoms with minimum emission. If the owner wants to update the minimum emission, they cannot directly add the denom with the new value because it would incorrectly increase the whitelist count.

On the other hand, the owner cannot remove the existing denom and add it back because removing a denom would cause future emissions to be canceled, causing leftover rewards to be stuck in the contract.

We classify this issue as major because it affects the correct functioning of the protocol.

Recommendation

We recommend adding a new entry point that allows the owner to modify existing denoms with new minimum emissions.

Status: Resolved

6. Updating base denom would cause incorrect price results

Severity: Major

In `contracts/oracle/base/src/contract.rs:198`, the `update_config` function allows the owner to update the base denom in the protocol. This is problematic because it would cause price sources to fail to work or return incorrect prices. Below we illustrate some issues this causes.

Firstly, the `query_arithmetic_twap_price` function in `contracts/oracle/osmosis/src/price_source.rs:433-449` attempts to query the pool's stored denom and base denom. If the base denom is updated to another denom, the query will fail because the new base denom does not exist in the pool.

Secondly, the contract instantiation phase in `contracts/oracle/wasm/src/contract.rs:49-53` stores a fixed price of base denom in USD value. Assuming the original base denom is `USDC` and the owner updates into `OSMO`. This would cause the query to reflect that one unit of `USDC` currency equals 1 `OSMO`, which is incorrect.

Thirdly, the `query_astroport_spot_price` function in `contracts/oracle/wasm/src/price_source.rs:269-291` attempts to simulate the swap and return it in base denom currency. Assuming the pair contract holds `USDT` denom and the base denom, which is `USDC`. If the owner updates the base denom to `OSMO`, the query will simulate the swap and return the amount of `USDC` received for one unit of `USDT`, but users and other price sources will see it return as `OSMO` currency.

We classify this issue as major because it affects the correct functioning of the protocol.

Recommendation

We recommend not allowing the owner to update the base denom.

Status: Acknowledged

7. Incentive rewards might be distributed to depositors outside the epoch period

Severity: Major

In `contracts/incentives/src/contract.rs:79`, the `SetAssetIncentives` message allows users to establish incentive schedules by defining an emission period.

When no deposits are made into the red bank during this emission period, the function `update_incentive_index` in `contracts/incentives/src/helpers.rs:162-164` will not modify the global index. This implies that no rewards are accumulated for the depositors during this period.

However, an inconsistency arises here. Despite the global index remaining unchanged, new stakers who make deposits into the red bank following the emission period end up receiving the rewards earmarked for the previous emission period.

Consequently, these new users are incorrectly awarded staking rewards even though they did not deposit funds into the red bank during the specified emission period.

A test case reproducing this issue is provided in the Appendix in the `test_receive_rewards_for_unstake_period` test case.

We classify this issue as major because it affects the correct functioning of the protocol.

Recommendation

We recommend ensuring the distribution of incentive rewards only to users who deposited within the designated emission period.

Status: Acknowledged

8. Duplicated denoms might cause incorrect whitelist counts

Severity: Minor

In `contracts/incentives/src/contract.rs:173` and `187`, the `execute_update_whitelist` function does not dedupe denoms in the `add_denoms` and `remove_denoms` vectors.

This is problematic because providing duplicate denoms to be added or removed would inflate the `whitelist_count` variable, causing an incorrect state stored in the contract.

We classify this issue as minor because only the contract owner can cause it.

Recommendation

We recommend deduping the denoms in the `add_denoms` and `remove_denoms` vectors.

Status: Resolved

9. Oracle centralization risks with `Fixed` price sources

Severity: Minor

In the current design, the owner can post arbitrary prices to the oracle using a `Fixed` price source in `contracts/oracle/wasm/src/price_source.rs:184-195` and `contracts/oracle/osmosis/src/price_source.rs381-392`. Prices are not validated, and every value is accepted.

This can be problematic since an attacker that gets access to the private key of the owner can arbitrarily manipulate prices without any restrictions. For example, the attacker could set the price of all assets to 0, which would allow the attacker to liquidate all users at their loss.

We classify this issue as minor because only the contract owner can cause it.

Recommendation

We recommend removing the `Fixed` price source from the oracle. Alternatively, the provided price should be validated to be non-zero, and a maximum price movement per time unit could be implemented.

Status: Acknowledged

10. Misconfiguring tolerance value to be higher than window size causes incorrect price reported

Severity: Minor

In `contracts/oracle/wasm/src/price_source.rs:316-321`, the `query_astroport_twap_price` function iterates over all snapshots to find those whose period falls within a specified tolerable window. This window is defined as the period ranging from the window size minus tolerance to window size plus tolerance. This calculation implies that the window size exceeds the tolerance period.

Suppose the owner misconfigures either the tolerance to be higher than the window size or the window size to be lower than the tolerance. In such situations, the above assumption will not hold, and the tolerable window period will not be effectively enforced.

Consequently, the valid tolerable window period could be a value smaller than window size minus tolerance or higher than window size plus tolerance, affecting the final price reported.

We classify this issue as minor because only the contract owner can cause it.

Recommendation

We recommend ensuring the window size is always larger than the tolerance period.

Status: Resolved

11. Removing price sources in the oracle could stop reward collector operations

Severity: Minor

In `contracts/oracle/base/contract.rs:106`, the contract owner can execute `RemovePriceSource` messages in order to remove price sources from the oracle.

However, the removal of asset prices could have a detrimental impact on the operations of the swapper and reward collector contracts.

Specifically, in `contracts/swapper/astroport/src/route.rs:91-92`, the swapper retrieves oracle prices during a swap process. If the owner proceeds to remove the price

source associated with the `fee_collector_denom` or the `safety_fund_denom`, the swap operation will encounter an error.

This would render it impossible to carry out asset swaps within the reward collector.

Recommendation

We recommend checking that `RemovePriceSource` messages do not target reward collector `fee_collector_denom` and `safety_fund_denom` denoms.

Status: Acknowledged

12. Missing prerequisites check when adding a `Pyth` price source

Severity: Minor

In `contracts/oracle/wasm/src/price_source.rs:184` and `contracts/oracle/osmosis/src/price_source.rs:381`, when validating a `Pyth` price source, no validation is performed to enforce prerequisites.

In fact, since `Pyth` prices are denominated in `USD`, a price source from `USD` to `base_denom` is needed in order to compute prices correctly.

We classify this issue as minor because only the contract owner can cause it.

Recommendation

We recommend checking if a price source from `USD` to `base_denom` is available before storing `Pyth` price sources.

Status: Resolved

13. Overflow checks not enabled for release profile

Severity: Informational

The following packages and contracts do not enable `overflow-checks` for the release profile:

- `contracts/address-provider/Cargo.toml`
- `contracts/incentives/Cargo.toml`
- `contracts/oracle/base/Cargo.toml`
- `contracts/oracle/osmosis/Cargo.toml`
- `contracts/oracle/wasm/Cargo.toml`
- `contracts/red-bank/Cargo.toml`
- `contracts/rewards-collector/Cargo.toml`
- `contracts/swapper/base/Cargo.toml`

- `contracts/swapper/astroport/Cargo.toml`
- `contracts/swapper/osmosis/Cargo.toml`

While enabled implicitly through the workspace manifest, a future refactoring might break this assumption.

Recommendation

We recommend enabling overflow checks in all packages, including those that do not currently perform calculations, to prevent unintended consequences if changes are added in future releases or during refactoring. Note that enabling overflow checks in packages other than the workspace manifest will lead to compiler warnings.

Status: Resolved

14. Funds in the swapper contract can be stolen

Severity: Informational

In `contracts/swapper/base/src/contract.rs:155-182`, the `swap_exact_in` function swaps `Coins` defined in `coin_in` input parameter without checking that they are provided by the user in `info.funds`.

This vulnerability enables potential attackers to seize all the coins within the contract by sending swap messages with specifically chosen `coin_in` parameters.

We classify this issue as informational because the swapper contract is not intended to hold any funds, as the `TransferResult` message will distribute all the funds out.

Recommendation

We recommend verifying that `coin_in` and `info.funds` are equal. Alternatively, we recommend properly documenting that the swapper contract is not intended to hold any funds.

Status: Acknowledged

Appendix A: Test Cases

1. Test case for "Missing denom validation when adding incentives could lead to insufficient funds error"

```
#[test]
fn test_incorrect_denom_deposit() {
    let owner = Addr::unchecked("owner");
    let mut mock_env = MockEnvBuilder::new(None, owner).build();

    let red_bank = mock_env.red_bank.clone();
    red_bank.init_asset(&mut mock_env, "uusdc", default_asset_params());

    let collateral_denom = "uusdc";
    let incentive_denom = "umars";
    let emission_per_second = 10;
    let duration = ONE_WEEK_IN_SEC;

    let incentives = mock_env.incentives.clone();
    incentives.whitelist_incentive_denoms(&mut mock_env, &[(incentive_denom,
3)]);
    let current_block_time = mock_env.app.block_info().time.seconds();

    // incorrect denom provided
    let funds = [coin(emission_per_second * duration as u128,
collateral_denom)];

    mock_env.fund_account(&mock_env.owner.clone(), &funds);

    cw_multi_test::Executor::execute_contract(&mut mock_env.app,
mock_env.owner.clone(), mock_env.incentives.contract_addr.clone(),
&mars_red_bank_types::incentives::ExecuteMsg::SetAssetIncentive {
                    collateral_denom: collateral_denom.to_string(),
                    incentive_denom: incentive_denom.to_string(),
                    emission_per_second: emission_per_second.into(),
                    start_time: current_block_time,
                    duration,
                }, &funds)
            .unwrap();

}
```

2. Test case for “Utilization rate can be exploited over 100% for new markets”

```
#[test]
fn zero_deposit_poc() {

    // setup
    let close_factor = Decimal::percent(40);
    let atom_price = Decimal::from_ratio(12u128, 1u128);
    let osmo_price = Decimal::from_ratio(15u128, 10u128);
    let atom_max_ltv = Decimal::percent(60);
    let osmo_max_ltv = Decimal::percent(80);
    let atom_liq_threshold = Decimal::percent(75);
    let osmo_liq_threshold = Decimal::percent(90);
    let atom_liq_bonus = Decimal::percent(2);
    let osmo_liq_bonus = Decimal::percent(5);

    let owner = Addr::unchecked("owner");
    let mut mock_env = MockEnvBuilder::new(None,
owner).close_factor(close_factor).build();

    let oracle = mock_env.oracle.clone();
    oracle.set_price_source_fixed(&mut mock_env, "uatom", atom_price);
    oracle.set_price_source_fixed(&mut mock_env, "uosmo", osmo_price);

    // init two assets
    let red_bank = mock_env.red_bank.clone();
    red_bank.init_asset(
        &mut mock_env,
        "uatom",
        default_asset_params_with(atom_max_ltv, atom_liq_threshold,
atom_liq_bonus),
    );
    red_bank.init_asset(
        &mut mock_env,
        "uosmo",
        default_asset_params_with(osmo_max_ltv, osmo_liq_threshold,
osmo_liq_bonus),
    );

    // testing configurations
    let borrower = Addr::unchecked("borrower");
    let borrower2 = Addr::unchecked("borrower2");

    // initial deposit amount
    let funded_atom = 1_u128; // 1 uatom

    // donation to protocol to cause interest exceeds 100%
    let donated_atom = 1_000_000_000_u128; // 1k atom
```

```
    // amount needed to borrow all donated amount
    let funded_osmo = 10_000_000_000_u128; // 10k osmo

    // 1. deposit atom
    mock_env.fund_account(&borrower, &[coin(funded_atom, "uatom")]);
    red_bank.deposit(&mut mock_env, &borrower, coin(funded_atom,
"uatom")).unwrap();

    // 2. donate atom to protocol (amount larger than deposit in step 1)
    mock_env.fund_account(&red_bank.contract_addr, &[coin(donated_atom,
"uatom")]);

    // 3. from another account, deposit osmo and borrow atom donated from step 2
    mock_env.fund_account(&borrower2, &[coin(funded_osmo, "uosmo")]);
    red_bank.deposit(&mut mock_env, &borrower2, coin(funded_osmo,
"uosmo")).unwrap();
    red_bank.borrow(&mut mock_env, &borrower2, "uatom", donated_atom).unwrap();

    // 4. wait 10 seconds
    let user_res = red_bank.query_user_collateral(&mut mock_env, &borrower,
"uatom");
    assert_eq!(user_res.amount, Uint128::new(funded_atom));
    mock_env.app.update_block(|b| b.time = b.time.plus_seconds(10));

    // 5. analyze interest accrued
    let new_user_res = red_bank.query_user_collateral(&mut mock_env, &borrower,
"uatom");
    assert_eq!(new_user_res.amount, Uint128::new(84_559_445_421));

}
```

3. Test case for "Incentive rewards might be distributed to depositors outside the epoch period"

```
#[test]
fn test_receive_rewards_for_unstake_period() {

    const ONE_DAY : u64 = 86400;

    let owner = Addr::unchecked("owner");
    let mut mock_env = MockEnvBuilder::new(None, owner).build();

    let red_bank = mock_env.red_bank.clone();
    red_bank.init_asset(&mut mock_env, "uusdc", default_asset_params());

    let incentives = mock_env.incentives.clone();
    incentives.whitelist_incentive_denoms(&mut mock_env, &[("umars", 3)]);

    // init incentive, note there are no people staked yet
    incentives.init_asset_incentive_from_current_block(
        &mut mock_env,
        "uusdc",
        "umars",
        10,
        ONE_DAY,
    );

    // query incentive state
    let incentive_state :
mars_red_bank_types::incentives::IncentiveStateResponse =
mock_env.app.wrap().query_wasm_smart(incentives.contract_addr.clone(),
&mars_red_bank_types::incentives::QueryMsg::IncentiveState { collateral_denom:
"uusdc".to_string(), incentive_denom: "umars".to_string() }).unwrap();

    // finish incentive time
    mock_env.increment_by_time(ONE_DAY);

    let new_incentive_state :
mars_red_bank_types::incentives::IncentiveStateResponse =
mock_env.app.wrap().query_wasm_smart(incentives.contract_addr.clone(),
&mars_red_bank_types::incentives::QueryMsg::IncentiveState { collateral_denom:
"uusdc".to_string(), incentive_denom: "umars".to_string() }).unwrap();

    // notice the global index is still the same
    assert_eq!(incentive_state, new_incentive_state);

    // mint funds to user
    let user = Addr::unchecked("user_a");
    let funded_amt = 10_000_000_000u128;
    mock_env.fund_account(&user, &[coin(funded_amt, "uusdc")]);
```

```
    // deposit
    red_bank.deposit(&mut mock_env, &user, coin(funded_amt, "uusdc")).unwrap();

    // note that user didnt stake in the previous period, but still receives
reward
    let rewards_balance = incentives.query_unclaimed_rewards(&mut mock_env,
&user);
    assert_eq!(rewards_balance.len(), 1);
    assert_eq!(rewards_balance[0].amount, Uint128::zero());

}
```