Audit Report

Mars Rover Updates

v1.0

February 3, 2023

Table of Contents

Table of Contents **2**

License **3**

Disclaimer **3**

Introduction **5**
- Purpose of This Report 5
- Codebase Submitted for the Audit 5
- Methodology 6
- Functionality Overview 6

How to Read This Report **7**

Code Quality Criteria **8**

Summary of Findings **9**

Detailed Findings **10**
- 1. Vault deposits are not affected by delisted coins 10
- 2. account-nft's contract UpdateConfig message cannot be executed after the minter role is transferred to credit-manager 10
- 3. Allowing deposit or duplicate vault tokens may cause unexpected outcomes 11
- 4. Transactions including ProvideLiquidity or WithdrawLiquidity actions may run out of gas 12
- 5. Coin whitelist update may run out of gas 12
- 6. Use of magic numbers decreases maintainability 13
- 7. Dependency on unreleased node version 13
- 8. Proposed new minter cannot be removed 14
- 9. Withdrawing liquidity requires the coin to be whitelisted 14

Appendix A: Test Cases **15**
- 1. Test case for "Allowing deposit or duplicate vault tokens would cause unexpected outcomes" 15

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 Mars Rover 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 GitHub repository:

https://github.com/mars-protocol/rover

Commit hash: `d58f03cbdeeacc87226526b5e7c202ab989883d9`

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

The submitted code implements Mars Rover, a generalized credit protocol built on the Mars lending market.

This audit covers the changes since our previous audit, which was performed on commit `1c10aa538eaa1c2dc93f10faba6223b2eaed3ec6`.

New features added to existing contracts are shown below:

- `account-nft`
    - Added burn guard functionality.
- `credit-manager`
    - Added delisting logic.
    - Allow specifying an optional amount during repayment.
    - Allow specifying an optional amount for swapping.
    - Allow specifying an optional amount when withdrawing liquidity.
    - Added a vault utilization query.
    - Modified vault pricing method and decimal math.

Additionally, a new `zapper` contract is added as part of the audit. The contract takes denoms and adds them to an Osmosis liquidity pool in exchange for LP tokens, which are later used to enter into auto-compounding vaults.

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	-
Code readability and clarity	Medium-High	Most functions are well-documented with clear and concise comments.
Level of documentation	Medium-High	The client shared the summary of changes since our previous audit along with links to merged pull requests, which act as a good reference for new functionalities to review.
Test coverage	Medium-High	`cargo tarpaulin` reports a 92.22% code coverage.

Summary of Findings

No	Description	Severity	Status
1	Vault deposits are not affected by delisted coins	Major	Resolved
2	`account-nft`'s contract `UpdateConfig` message cannot be executed after the minter role is transferred to `credit-manager`	Major	Resolved
3	Allowing deposit or duplicate vault tokens may cause unexpected outcomes	Minor	Partially Resolved
4	Transactions including `ProvideLiquidity` or `WithdrawLiquidity` actions may run out of gas	Informational	Acknowledged
5	Coin whitelist update may run out of gas	Informational	Acknowledged
6	Use of magic numbers	Informational	Acknowledged
7	Dependency on unreleased node version	Informational	Acknowledged
8	Proposed new minter cannot be removed	Informational	Acknowledged
9	Withdrawing liquidity requires the coin to be whitelisted	Informational	Acknowledged

Detailed Findings

1. Vault deposits are not affected by delisted coins

Severity: Major

In `contracts/credit-manager/src/health.rs:160`, when a coin is delisted, the LTV is set to zero. However, vault deposits of the delisted coin are not affected, as seen in lines `100-128`. Consequently, users can maintain the collateral value of coins deposited prior to delisting. They could even time an attack based on the previous LTV by frontrunning the delist transaction or entering a vault when an announcement to delist a specific coin is published.

For example, suppose `ATOM` is to be delisted by the Mars team. A borrower notices it and enters a locked vault that accepts `ATOM` as a base deposit. Despite `ATOM` being delisted, the borrower's collateral in the form of vault tokens still contributes to the overall LTV. The borrower can execute the `RequestVaultUnlock` message to convert the vault tokens back to `ATOM`, contributing to the total collateral value using the red bank's LTV.

Additionally, the client also identified that delisted vaults still contribute to the total collateral value.

We classify this issue as major because this affects the correct functioning of the overall system.

Recommendation

We recommend calculating the base and vault token's LTV as zero if the base token is delisted.

Status: Resolved

2. `account-nft`'s contract `UpdateConfig` message cannot be executed after the `minter` role is transferred to `credit-manager`

Severity: Major

The comment in `contracts/account-nft/src/msg/instantiate.rs:22-24` states that the minter role will be transferred to the `credit-manager` contract. However, the `credit-manager` contract has no message defined to call the `account-nft`'s `UpdateConfig`, which implies that the `max_value_for_burn` and `proposed_new_minter` configuration values can no longer be updated once the ownership has been transferred.

We classify this issue as major because this affects the correct functioning of the overall system.

Recommendation

We recommend implementing a message in the `credit-manager` that enables its `owner` to update the configurations of the linked `account-nft` contract.

Status: Resolved

3. Allowing deposit or duplicate vault tokens may cause unexpected outcomes

Severity: Minor

In `contracts/credit-manager/src/vault/utils.rs:92-101`, the `rover_vault_balance_value` function queries the balance of vault tokens to calculate the token's value. However, there is no validation to ensure the contract owner does not configure the vault token as allowed coins or that the configured vaults do not contain duplicate token denoms.

This is problematic because either of the above would cause the queried balance to include user deposits or other vault balances, causing the following functions to return a greater amount than expected:

- `vault_utilization_in_deposit_cap_denom`
- `query_all_total_vault_coin_balances`
- `query_total_vault_coin_balance`
- `assert_deposit_is_under_cap`

As a result, the first three functions would return incorrect information to users, while the last function would potentially cause a denial of service due to an `AboveVaultDepositCap` error when users try to deposit into vaults.

Please see the `test_duplicate_vault_tokens` test case to reproduce this issue.

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

Recommendation

We recommend preventing the vault token from being configured as an allowed coin and verifying no duplicate vault tokens are accepted.

Status: Partially Resolved

This issue is partially resolved because the client only implemented a fix to validate duplicate vault denoms. They mentioned that restricting the deposit of vault tokens is trickier because they currently have no way of distinguishing between a normal token and a vault token.

4. Transactions including `ProvideLiquidity` or `WithdrawLiquidity` actions may run out of gas

Severity: Informational

In `contracts/credit-manager/src/zap.rs:16` and `55`, the `provide_liquidity` and `withdraw_liquidity` functions could trigger a large number of sub-messages that may consume the transaction's entire gas. Those actions send a message to the `zapper` contract in order to execute a trade on `osmosis` and then return liquidity pool tokens and remainder coins to the `recipient`.

To achieve this, they have to send a sub-message to osmosis for the trade and `(n+1)` `ReturnCoin` sub-messages - where `n` is the number of `coins`. Additionally, for each of the sub-messages, a `Bank` message is created.

Since `ProvideLiquidity` and `WithdrawLiquidity` actions are only a part of the transaction, the transaction will also include `AssertOneVaultPositionOnly` and `AssertBelowMaxLTV` callbacks, which may consume an excessive amount of gas.

Recommendation

We recommend optimizing the `zapper`'s `ProvideLiquidity` and `WithdrawLiquidity` messages (see `contracts/zapper/base/src/contract.rs:92` and `136`) by merging `ReturnCoin` sub-messages into a single one.

Status: Acknowledged

5. Coin whitelist update may run out of gas

Severity: Informational

In `contracts/credit-manager/src/update_config.rs:45`, the `UpdateConfig` message handler clears all the `ALLOWED_COINS` map entries and then inserts all new coins provided in the message into the map.

Since this operation requires iterating on both the stored and the proposed whitelists in order to perform the update, the execution may run out of gas if the number of elements in the whitelist is significant.

Recommendation

We recommend implementing a mechanism to add or remove entries from the `ALLOWED_COINS` map instead of removing and substituting all its entries in a batch.

Status: Acknowledged

6. Use of magic numbers decreases maintainability

Severity: Informational

Throughout the codebase, hard-coded number literals without context or a description are used. Using such “magic numbers” goes against best practices as they reduce code readability and maintenance as developers are unable to easily understand their use and may make inconsistent changes across the codebase.

Instances of magic numbers are listed below:

- `contracts/zapper/base/src/contract.rs:113`
- `contracts/zapper/base/src/contract.rs:159`
- `contracts/account-nft/src/contract.rs:36`

Recommendation

We recommend defining magic numbers as constants with descriptive variable names and comments, where necessary.

Status: Acknowledged

7. Dependency on unreleased node version

Severity: Informational

The current codebase includes a fix for spot price queries from Osmosis GAMMs in `packages/chains/src/helpers.rs:73-76`. However, at the time of writing, the fix made in the node client is still unreleased.

We classify this issue as informational as it is not expected that these contracts will be deployed prior to the fix being released.

Recommendation

We recommend ensuring that the fix is released and deployed to the Osmosis mainnet prior to deployment of the contracts within this codebase.

Status: Acknowledged

8. Proposed new minter cannot be removed

Severity: Informational

In `contracts/account-nft/src/contract.rs:67-77`, there are no handlers exposed to remove the `proposed_new_minter` from the contract storage. This is problematic because if the pending minter does not intend to accept the role, the current minter cannot set the value back to `None`.

We classify this issue as informational because the contract minter can still overwrite the minter back to the current contract addresses, which is equivalent to an empty proposed new minter.

Recommendation

We recommend implementing a handler for the contract minter to set `proposed_new_minter` to `None`.

Status: Acknowledged

9. Withdrawing liquidity requires the coin to be whitelisted

Severity: Informational

In `contracts/credit-manager/src/zap.rs:61` and `79`, the liquidity pool token and withdrawn liquidity are validated to be whitelisted. In the delisting logic pull request, one of the changes allows the withdrawal of assets that are not included in the whitelisted assets. However, this is not applied when withdrawing liquidity tokens.

Recommendation

We recommend allowing users to withdraw liquidity tokens even if they are not whitelisted.

Status: Acknowledged

Appendix A: Test Cases

1. Test case for “[Allowing deposit or duplicate vault tokens would cause unexpected outcomes]”

The test case should fail if the vulnerability is patched.

```rust
#[test]
fn test_duplicate_vault_tokens() {
    let lp_token = lp_token_info();

    // create two vaults with the same vault token
    let first_vault = unlocked_vault_info();
    let second_vault = unlocked_vault_info();

    let alice = Addr::unchecked("alice");
    let bob = Addr::unchecked("bob");

    let mut mock = MockEnv::new()
        .allowed_coins(&[lp_token.clone()])
        .vault_configs(&[
            first_vault.clone(),
            second_vault.clone()
        ])
        .fund_account(AccountToFund {
            addr: alice.clone(),
            funds: vec![lp_token.to_coin(200)],
        }).fund_account(AccountToFund {
            addr: bob.clone(),
            funds: vec![lp_token.to_coin(200)],
        })
        .build()
        .unwrap();

    let alice_account_id = mock.create_credit_account(&alice).unwrap();
    let bob_account_id = mock.create_credit_account(&bob).unwrap();

    // get vault addresses
    let all_vaults : Vec<VaultInfoResponse> = mock.app.wrap().query_wasm_smart(
        mock.rover.clone(),
        &QueryMsg::VaultsInfo {
            start_after: None,
            limit: None,
    }).unwrap();
    assert_eq!(all_vaults.len(), 2);
    let vault_one = all_vaults[0].vault.clone();
    let vault_two = all_vaults[1].vault.clone();
    assert_ne!(vault_one.address, vault_two.address);
```

```
    // Alice enters vault_one
    mock.update_credit_account(
        &alice_account_id,
        &alice,
        vec![
            Deposit(lp_token.to_coin(200)),
            EnterVault {
                vault: vault_one.clone(),
                coin: lp_token.to_action_coin(200),
            }
        ],
        &[lp_token.to_coin(200)],
    )
    .unwrap();

    // VaultsInfo{} snapshot (pre)
    let vaults_info_before : Vec<VaultInfoResponse> =
mock.app.wrap().query_wasm_smart(
        mock.rover.clone(),
        &QueryMsg::VaultsInfo {
            start_after: None,
            limit: None,
        },
    ).unwrap();

    // TotalVaultCoinBalance{} snapshot (pre)
    let total_vault_coin_balance_before : Uint128 =
mock.app.wrap().query_wasm_smart(
        mock.rover.clone(),
        &QueryMsg::TotalVaultCoinBalance {
            vault: vault_one.clone()
        },
    ).unwrap();

    // AllTotalVaultCoinBalances{} snapshot (pre)
    let all_total_vault_coin_balances_before : Vec<VaultWithBalance> =
mock.app.wrap().query_wasm_smart(
        mock.rover.clone(),
        &QueryMsg::AllTotalVaultCoinBalances {
            start_after: None,
            limit: None,
        },
    ).unwrap();

    // Bob enters vault_two
    mock.update_credit_account(
        &bob_account_id,
        &bob,
        vec![
```

```
            Deposit(lp_token.to_coin(200)),
            EnterVault {
                vault: vault_two.clone(),
                coin: lp_token.to_action_coin(200),
            }
        ],
        &[lp_token.to_coin(200)],
    )
    .unwrap();

     // VaultsInfo{} snapshot (post)
    let vaults_info_after : Vec<VaultInfoResponse> =
mock.app.wrap().query_wasm_smart(
        mock.rover.clone(),
        &QueryMsg::VaultsInfo {
            start_after: None,
            limit: None,
        },
    ).unwrap();

    // TotalVaultCoinBalance{} snapshot (post)
    let total_vault_coin_balance_after : Uint128 =
mock.app.wrap().query_wasm_smart(
        mock.rover.clone(),
        &QueryMsg::TotalVaultCoinBalance {
            vault: vault_one.clone()
        },
    ).unwrap();

    // AllTotalVaultCoinBalances{} snapshot
    let all_total_vault_coin_balances_after : Vec<VaultWithBalance> =
mock.app.wrap().query_wasm_smart(
        mock.rover,
        &QueryMsg::AllTotalVaultCoinBalances {
            start_after: None,
            limit: None,
        },
    ).unwrap();

    // TotalVaultCoinBalance{} query is affected
    assert_ne!(total_vault_coin_balance_before, total_vault_coin_balance_after);

    // AllTotalVaultCoinBalances{} query is affected
    assert_ne!(all_total_vault_coin_balances_before[0],
all_total_vault_coin_balances_after[0]);

    // VaultsInfo{} query is affected

    // even tho no deposit made for firs vault, the query utilization becomes
incorrect
```

```
    assert_ne!(vaults_info_before[0].utilization,
vaults_info_after[0].utilization);

    // despite both vaults have a deposit of same amount, their utilization rate
is different
    assert_ne!(vaults_info_before[0].utilization,
vaults_info_after[1].utilization);
}
```