Audit Report

Mars Rover

v1.0

December 9, 2022

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 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: `1c10aa538eaa1c2dc93f10faba6223b2eaed3ec6`

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 Rovers are a new DeFi primitive of the Mars ecosystem. They are isolated credit accounts where users can aggregate DeFi activities in one spot with a single liquidation LTV. Also, they are bundles of on-chain transactions, which are represented by transferable NFTs that can be sold and fractionalized.

The audit scope comprehends the `account-nft`, `credit-manager`, `oracle-adapter`, and `swapper` CosmWasm smart contracts.

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	-
Level of documentation	Medium-High	Provided documentation and flow diagrams contain all the information needed.
Test coverage	Medium-High	`cargo tarpaulin` reports code coverage of 91.93%

Summary of Findings

No	Description	Severity	Status
1	Borrowers can prevent liquidation leading to bad debt accumulating	Critical	Resolved
2	Liquidators can extract a higher value by looping small amounts of liquidations	Major	Resolved
3	Contract version and name are overwritten during instantiation	Minor	Resolved
4	Duplicate keys should be removed to prevent misconfigurations	Minor	Resolved
5	Update of the `account-nft` address can lead to state inconsistencies	Minor	Acknowledged
6	Funds held by swapper contract may be unintendedly withdrawn	Minor	Acknowledged
7	Update of the red bank address in the `credit-manager` could lead to state inconsistency	Minor	Resolved
8	The execution of the `AssertBelowMaxLTV` callback at the end of the `UpdateCreditAccount` transaction could let the execution to run out of gas	Minor	Acknowledged
9	`MAX_CLOSE_FACTOR` is not validated	Informational	Resolved
10	Query silently returns input when no pricing method is found	Informational	Resolved
11	Redundant checks on received funds	Informational	Acknowledged
12	Contracts should implement a two step ownership transfer	Informational	Resolved
13	Custom access controls implementation	Informational	Resolved
14	Overflow checks not enabled for release profile	Informational	Acknowledged
15	Unbounded number of steps during route registration	Informational	Acknowledged
16	Unused callback can be removed	Informational	Resolved

Detailed Findings

1. Borrowers can prevent liquidation leading to bad debt accumulating

Severity: Critical

When liquidating collateral in `contracts/credit-manager/src/vault/liquidate_vault.rs:154-166`, all the borrower's unlocking positions are processed in a loop. A borrower can create many tiny unlocking positions using the `RequestVaultUnlock` message, causing the total unlocking positions cardinality to grow to the point where the `liquidate_unlocking` function runs out of gas.

This is highly problematic, since borrowers can prevent being liquidated, which can result in bad debt accumulating.

Recommendation

We recommend enforcing a maximum number of unlocking positions per borrower.

Status: Resolved

2. Liquidators can extract a higher value by looping small amounts of liquidations

Severity: Major

In `contracts/credit-manager/src/liquidate_coin.rs:119-127`, the `request_amount` is rounded up using the `ceil` function. This allows a liquidator to extract a higher value by liquidating small amounts of collateral within a loop.

Additionally, the maximum close factor can be bypassed as long as a single liquidation message does not exceed the limit, allowing the total liquidated amount to be higher than the configured limit.

We consider this a major instead of critical issue because performing multiple small liquidations consumes a lot of gas, which decreases the profitability of the attack.

Please see the `test_repeated_single_liquidation` test case to reproduce the issue.

Recommendation

We recommend verifying that the total liquidated collateral does not exceed the maximum close factor of the borrower's debt value after a series of liquidation actions.

Status: Resolved

3. Contract version and name are overwritten during instantiation

Severity: Minor

In `contracts/account-nft/src/contract.rs:23-36`, during instantiation of the `account-nft` contract, the version and name of the contract are set twice. First in `contract/account-nft/src/contract.rs:29-33` and then again during the call of the parent contract instantiation in `contract/account-nft/src/contract.rs:35`.

This implies that the stored `CW2` metadata would be the one from the `cw721-base` contract and not the intended one, potentially causing problems in future migrations.

Recommendation

We recommend calling the `set_contract_version` function after the instantiation of the parent contract to ensure that the final stored values are those specified in `Cargo.toml`.

Status: Resolved

4. Duplicate keys should be removed to prevent misconfigurations

Severity: Minor

In `contracts/credit-manager/src/instantiate.rs:19-25` and `contracts/oracle-adapter/src/contract.rs:47-49`, the `msg.allowed_vaults`, `msg.vault_pricing` and `msg.allowed_coins` are not deduplicated before storing them. If any allowed vault addresses or coin denominations are duplicates, earlier configurations would be overwritten, and only the last key would be saved successfully.

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

Recommendation

We recommend deduplicating both inputs to prevent a potential misconfiguration.

Status: Resolved

5. Update of the `account-nft` address can lead to state inconsistencies

Severity: Minor

In `contracts/credit-manager/src/execute.rs:65-80`, the contract owner is able to update the address of the `account-nft` contract. This can cause state inconsistencies as account ids associated with the previous NFT contract are removed.

Also, owners of NFTs from the old contract will lose access to their funds managed in the `credit-manager` that will be virtually transferred to the owners of the NFTs from the new contract.

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

Recommendation

We recommend adding guards to ensure post-instantiation of the `account-nft` contract alterations can only be performed via migration.

Status: Acknowledged

6. Funds held by `swapper` contract may be unintendedly withdrawn

Severity: Minor

When performing a swap in the `swapper` contract, the output is transferred to the recipient using a `TransferResult` message created in `contract/swapper/base/src/contract.rs:187-194`.

Then any amount of the `denom_in` and `denom_out` is transferred to the recipient in `contract/swapper/base/src/contract.rs:234`.

However, in the case that the swapper contract holds additional funds of either denomination those would be automatically sent to the recipient in addition to the input and output amounts.

We classify this issue as minor despite the fact that no funds should be present in the swapper contract there may be situations where funds are sent to the contract, e.g. an airdrop or inadvertent usage.

Recommendation

We recommend calculating the amounts to transfer taking into account funds potentially held in the contract in order to prevent any additional funds from being transferred to the recipient.

Status: Acknowledged

7. Update of the red bank address in the `credit-manager` could lead to state inconsistency

Severity: Minor

In `contracts/credit-manager/src/execute.rs:65-80`, the contract owner is able to update the address of the red bank in the `credit-manager`.

This would cause state inconsistencies since all the data stored in `TOTAL_DEBT_SHARES` and `DEBT_SHARES` will not be updated accordingly.

Recommendation

We recommend removing the functionality for updating the red bank address, a migration should be used instead.

Status: Resolved

8. The execution of the `AssertBelowMaxLTV` callback at the end of the `UpdateCreditAccount` transaction could run out of gas

Severity: Minor

In `contracts/credit-manager/src/execute.rs:285-288`, the `AssertBelowMaxLTV` is always added to the list of `callbacks` to execute after the other `Actions` provided as an input in the `UpdateCreditAccount` message.

This callback is executing two times an unbounded loop through the `COIN_BALANCES`, `DEBT_SHARES` and `VAULT_POSITIONS` vectors that have not a capped length. Also, it is executed after all the other `Actions` and `Callbacks` so it has only a fraction of the initially provided gas. This implies that if the cardinality of the vectors is significant, the remaining gas could not be enough to pay the computation leading to out of gas errors.

While this scenario is not likely to happen at the launch of the protocol, the risk of running out of gas will increase with more assets supported and wider adoption.

Recommendation

We recommend analyzing and optimizing the `AssertBelowMaxLTV` callback gas consumption.

Status: Acknowledged

9. `MAX_CLOSE_FACTOR` is not validated

Severity: Informational

In `contracts/credit-manager/src/instantiate.rs:15`, the contract owner defines the `MAX_CLOSE_FACTOR` of liquidations which determines the maximum amount of a position that can be liquidated.

However, the value provided has no validation which could lead to the variable being ineffectual and not providing a limit to the value that can be liquidated in a single transaction.

Recommendation

We recommend performing basic validation of the `MAX_CLOSE_FACTOR` during instantiation and update to ensure that the value is less than or equal to one.

Status: Resolved

10. Query silently returns input when no pricing method is found.

Severity: Informational

The function `query_priceable_underlying` in `contracts/oracle-adapter/src/contract.rs` finds the pricing methodology of the coin denom and queries the relevant vault to calculate the value of redemption.

If the submitted coin denom does not have a vault price then the input is simply returned to the user. This could lead to wrong assumptions and may negatively impact the usability of the query.

Recommendation

We recommend that in the event a user submits a coin that has no pricing methodology, an error is returned.

Status: Resolved

11. Redundant checks on received funds

Severity: Informational

In `packages/rover/src/coins.rs:136-158`, the `try_from` function attempts to verify that no zero amount denom and no duplicate denoms are provided. This check is unnecessary because Cosmos SDK will prevent zero amounts from being sent (an error will occur) and will automatically combine duplicate denoms into one single denom (eg. [200 `ATOM`, 100 `ATOM`] will become [300 `ATOM`]).

Recommendation

We recommend removing the checks to reduce gas consumption.

Status: Acknowledged

12. Contracts should implement a two step ownership transfer

Severity: Informational

The contracts within the scope of this audit allow the current owner to execute a one-step ownership transfer. While this is common practice, it presents a risk for the ownership of the contract to become lost if the owner transfers ownership to the incorrect address. A two-step ownership transfer will allow the current owner to propose a new owner, and then the account that is proposed as the new owner may call a function that will allow them to claim ownership and actually execute the config update.

Recommendation

We recommend implementing a two-step ownership transfer. The flow can be as follows:

1. The current owner proposes a new owner address that is validated and lowercased.
2. The new owner account claims ownership, which applies the configuration changes.

Status: Resolved

13. Custom access controls implementation

Severity: Informational

The contracts within the scope of this audit implement custom access controls. Although no instances of broken controls or bypasses have been found, using a battle-tested implementation reduces potential risks and the complexity of the codebase.

Also, the access control logic is duplicated across the handlers of each function, which negatively impacts the code's readability and maintainability.

Recommendation

We recommend using a well-known access control implementation such as `cw_controllers::Admin` (https://docs.rs/cw-controllers/0.14.0/cw_controllers/struct.Admin.html).

Status: Resolved

14. Overflow checks not enabled for release profile

Severity: Informational

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

- `contracts/account-nft/Cargo.toml`
- `contracts/credit-manager/Cargo.toml`
- `contracts/oracle-adapter/Cargo.toml`
- `contracts/swapper/Cargo.toml`
- `packages/rover/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: Acknowledged

15. Unbounded number of steps during route registration

Severity: Informational

In `contracts/swapper/base/src/contract.rs:246-274`, the owner is able to define custom swap routes between token pairs in order to facilitate the exchange of tokens that do not have a shared pool. However, the number of steps a route is able to host is unbounded which could lead to inefficient routes being defined.

Recommendation

We recommend limiting the number of steps a new route can contain during validation in `contracts/swapper/osmosis/src/route.rs`.

Status: Acknowledged

16. Unused callback should be removed

Severity: Informational

The `ForceExitVault` `callback` defined in `packages/rover/src/msg/execute.rs:147-151`, is not used anywhere in the protocol and should be removed from the codebase to increase its maintainability.

Recommendation

We recommend removing the `ForceExitVault callback` from the codebase since it is not used.

Status: Resolved

Appendix: Test Cases

1. Test case for "[Liquidators can extract more money by looping small amounts of liquidations]"

The test case demonstrates that the liquidator is able to liquidate an extra 33 `OSMO` compared to the old `test_debt_amount_adjusted_to_close_factor_max` test case. Note that the max close factor is instantiated at 5%, which implies that the restriction is bypassed.

As a comparison, the `test_debt_amount_adjusted_to_close_factor_max` test case fails when the max close factor is instantiated as 5%.

```
#[test]
fn test_repeated_single_liquidation() {
    // modification of test_debt_amount_adjusted_to_close_factor_max()
    // reproduced in contracts/credit-manager/tests/test_liquidate_coin.rs
    let uosmo_info = uosmo_info();
    let uatom_info = uatom_info();
    let liquidator = Addr::unchecked("liquidator");
    let liquidatee = Addr::unchecked("liquidatee");
    let mut mock = MockEnv::new()
    // set max close factor as 5% to see if we can bypass
    .max_close_factor(Decimal::from_ratio(5_u32, 100_u32))
        .allowed_coins(&[uosmo_info.clone(), uatom_info.clone()])
        .fund_account(AccountToFund {
            addr: liquidatee.clone(),
            funds: coins(300, uosmo_info.denom.clone()),
        })
        .fund_account(AccountToFund {
            addr: liquidator.clone(),
            funds: coins(300, uatom_info.denom.clone()),
        })
        .build()
        .unwrap();
    let liquidatee_account_id =
mock.create_credit_account(&liquidatee).unwrap();

    mock.update_credit_account(
        &liquidatee_account_id,
        &liquidatee,
        vec![
            Deposit(uosmo_info.to_coin(300)),
            Borrow(uatom_info.to_coin(100)),
        ],
        &[Coin::new(300, uosmo_info.denom.clone())],
    )
```

```rust
    .unwrap();

    mock.price_change(CoinPrice {
        denom: uatom_info.denom.clone(),
        price: 6.to_dec().unwrap(),
    });

    let liquidator_account_id =
mock.create_credit_account(&liquidator).unwrap();

    // vector of actions to execute
    let mut actions = vec![];

    // liquidator must deposit debt funds
    actions.push(Deposit(uatom_info.to_coin(50)));

    // instead of liquidating large amount of debt coin, we liquidate one denom
per each message
    // the ceil() operation will automatically round up the value, giving us
more funds
    for _ in 1..12 {
        actions.push( LiquidateCoin {
            liquidatee_account_id: liquidatee_account_id.clone(),
            debt_coin: uatom_info.to_coin(1),
            request_coin_denom: uosmo_info.denom.clone(),
        });
    }

    mock.update_credit_account(
        &liquidator_account_id,
        &liquidator,
        actions,
        &[uatom_info.to_coin(50)],
    )
    .unwrap();

    // Assert liquidatee's new position
    let position = mock.query_positions(&liquidatee_account_id);
    println!("\nLiquidatee position: \n{:?}\n\n", position);

    let osmo_balance = get_coin("uosmo", &position.coins);
    println!("Original Osmo balance is 36, found: {:?}", osmo_balance.amount);
    // assert_eq!(osmo_balance.amount, Uint128::new(36));
    let atom_balance = get_coin("uatom", &position.coins);
    println!("Original ATOM balance is 100, found: {:?}", atom_balance.amount);
    // assert_eq!(atom_balance.amount, Uint128::new(100));

    // assert_eq!(position.debts.len(), 1);
    let atom_debt = get_debt("uatom", &position.debts);
    println!("Original ATOM debt is 91, found: {:?}", atom_debt.amount);
```

```
    // assert_eq!(atom_debt.amount, Uint128::new(91));

    // Assert liquidator's new position
    let position = mock.query_positions(&liquidator_account_id);
    println!("\nLiquidator position: \n{:?}\n\n", position);
    let atom_balance = get_coin("uatom", &position.coins);
    println!("Original Atom balance is 40, found: {:?}", atom_balance.amount);
    // assert_eq!(atom_balance.amount, Uint128::new(40));
    let osmo_balance = get_coin("uosmo", &position.coins);
    println!("Original Osmo balance is 264, found: {:?}", osmo_balance.amount);
    println!("Liquidator extracted excess {} OSMO", osmo_balance.amount -
Uint128::from(264_u64));
    // assert_eq!(osmo_balance.amount, Uint128::new(264));
}
```