Mars Protocol - Red Bank

CosmWasm Financial Security Audit

Prepared by: **Halborn**

Date of Engagement: **December 12th, 2022 - January 13th, 2023**

Visit: **Halborn.com**

DOCUMENT REVISION HISTORY

VERSION	MODIFICATION	DATE	AUTHOR
0.1	Document Creation	12/12/2022	Luis Quispe Gonzales
0.2	Document Update	12/19/2022	Luis Quispe Gonzales
0.3	Draft Version	01/16/2023	Luis Quispe Gonzales
0.4	Draft Review	01/16/2023	Gabi Urrutia
1.0	Remediation Plan	01/19/2023	Luis Quispe Gonzales
1.1	Remediation Plan Review	01/20/2023	Gabi Urrutia

CONTACTS

CONTACT	COMPANY	EMAIL
Rob Behnke	Halborn	Rob.Behnke@halborn.com
Steven Walbroehl	Halborn	Steven.Walbroehl@halborn.com
Gabi Urrutia	Halborn	Gabi.Urrutia@halborn.com
Luis Quispe Gonzales	Halborn	Luis.QuispeGonzales@halborn.com

EXECUTIVE OVERVIEW

1.1 INTRODUCTION

Mars Protocol engaged Halborn to conduct a security audit on their smart contracts beginning on December 12th, 2022 and ending on January 13th, 2023. The security assessment was scoped to the smart contracts provided in the GitHub repository outposts, commit hashes and further details can be found in the **Scope** section of this report.

1.2 AUDIT SUMMARY

The team at Halborn assigned a full-time security engineer to audit the security of the smart contract. The security engineer is a blockchain and smart-contract security expert with advanced penetration testing, smart-contract hacking, and deep knowledge of multiple blockchain protocols.

The purpose of this audit is to:

- Ensure that smart contract functions operate as intended
- Identify potential security issues with the smart contracts

In summary, Halborn identified some improvements to reduce the likelihood and impact of risks, which were mostly addressed by Mars Protocol team. The main ones are the following:

- Use 'refund_amount' as 'liquidity_taken' in repay function.
- Restrict the conditions to update the uncollateralized loan limit to 0 for C2C partners.
- Split ownership transfer functionality to allow the recipient to complete the transfer.
- Update liquidation_compute_amounts function to throw an error message if the value of 'debt_amount_to_repay' or 'collateral_amount_to_liquidate' is 0.
- Include a ramp change schema for 'liquidation_threshold' in update_asset function.
- Validate the format of 'denom' argument.

1.3 TEST APPROACH & METHODOLOGY

This framework provides a risk-based approach to assess the likelihood of a financial security event based on auditing the interactions and inputs around environmental factors of a smart contract or DeFi protocol.

Given the dynamic nature of such an audit, several approaches are combined to perform a holistic assessment of which developers can make the best effort to protect themselves from a revenue impacting event through risk awareness and mitigating factors.

Halborn performed a combination of manual and automated security testing to balance efficiency, timeliness, practicality, and accuracy in regard to the scope of this audit. While manual testing is recommended to uncover flaws in logic, process, and implementation; automated testing techniques help enhance coverage of the Rust code and can quickly identify items that do not follow security best practices. The following phases and associated tools were used throughout the term of the audit:

- Research into architecture and purpose.
- Smart contract manual code review and walkthrough.
- Manual testing by custom scripts and fuzzers.
- Scanning of Rust files for vulnerabilities, security hotspots or bugs.
- Static Analysis of security for scoped contract, and imported functions.
- Testnet deployment.

RISK METHODOLOGY:

Vulnerabilities or issues observed by Halborn are ranked based on the risk assessment methodology by measuring the **LIKELIHOOD** of a security incident and the **IMPACT** should an incident occur. This framework works for communicating the characteristics and impacts of technology vulnerabilities. The quantitative model ensures repeatable and accurate measurement while enabling users to see the underlying vulnerability characteristics that were used to generate the Risk scores. For every vulnerability, a risk

level will be calculated on a scale of 5 to 1 with 5 being the highest likelihood or impact.

RISK SCALE - LIKELIHOOD

5 - Almost certain an incident will occur.
4 - High probability of an incident occurring.
3 - Potential of a security incident in the long term.
2 - Low probability of an incident occurring.
1 - Very unlikely issue will cause an incident.

RISK SCALE - IMPACT

5 - May cause devastating and unrecoverable impact or loss.
4 - May cause a significant level of impact or loss.
3 - May cause a partial impact or loss to many.
2 - May cause temporary impact or loss.
1 - May cause minimal or un-noticeable impact.

The risk level is then calculated using a sum of these two values, creating a value of 10 to 1 with 10 being the highest level of security risk.

CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL

10 - CRITICAL
9 - 8 - HIGH
7 - 6 - MEDIUM
5 - 4 - LOW
3 - 1 - VERY LOW AND INFORMATIONAL

1.4 SCOPE

The security assessment was scoped to financial security attacks on the following items:

1. CosmWasm Smart Contracts
   (a) Repository: outposts
   (b) Commit ID: 4a1a55a
   (c) Contract in scope:
      - red-bank

Out-of-scope: Not-specified contracts and external libraries.

2. ASSESSMENT SUMMARY & FINDINGS OVERVIEW

CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL
0	1	1	4	2

LIKELIHOOD

IMPACT

(HAL-03)				(HAL-01)
(HAL-04) (HAL-05) (HAL-06)		(HAL-02)		
(HAL-07)				
	(HAL-08)			

EXECUTIVE OVERVIEW

SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
(HAL-01) ERRONEOUS UPDATE OF BORROW AND LIQUIDITY RATES WHEN REPAYING	High	SOLVED - 01/12/2023
(HAL-02) UPDATE OF LOAN LIMIT COULD PRODUCE UNDESIRED CONSEQUENCES TO C2C PARTNERS	Medium	SOLVED - 01/18/2023
(HAL-03) PRIVILEGED ADDRESS CAN BE TRANSFERRED WITHOUT CONFIRMATION	Low	SOLVED - 01/13/2023
(HAL-04) LIQUIDATORS COULD DRAIN COLLATERALS WITHOUT SPENDING COINS	Low	SOLVED - 01/19/2023
(HAL-05) LIQUIDATION THRESHOLD VALUE CAN BE CHANGED UNRESTRICTEDLY	Low	RISK ACCEPTED
(HAL-06) DENOM IS NOT VALIDATED IN SOME FUNCTIONS	Low	SOLVED - 01/05/2023
(HAL-07) USERS COULD LIQUIDATE WITHOUT RECEIVING COLLATERALS IN RETURN	Informational	SOLVED - 01/19/2023
(HAL-08) REPAYMENT OF UNCOLLATERALIZED LOANS COULD ERRONEOUSLY FAIL	Informational	ACKNOWLEDGED

FINDINGS & TECH DETAILS

3.1 (HAL-01) ERRONEOUS UPDATE OF BORROW AND LIQUIDITY RATES WHEN REPAYING - HIGH

Description:

When users repay their debts using the repay function in **red-bank** contract, the following values of a market are updated:

- borrow_rate: Rate charged to borrowers
- liquidity_rate: Rate paid to depositors

If a user repays more than his full debt, the excess payment is refunded, but the update of borrow and liquidity rates are erroneously calculated. As a consequence, the value of borrow_rate decreases and the value of liquidity_rate increases (both of them regarding the expected values) at expenses of Mars protocol. Also, more excess in users' payment regarding their total debt, more distorted are the borrow and liquidity rates.

Proof of Concept:

Scenario 1: User fully repays his debt

1. User borrows 5_000_000 uusdc.

```
// user_1 borrows some usdc (no usdc in the account before)
let borrowed_usdc: u128 = 5_000_000u128;
red_bank.borrow(&mut mock_env, sender: &user_1, denom: "uusdc", amount: borrowed_usdc).unwrap();
```

2. After some blocks, he decided to fully repay, so he would have to pay 5_000_069 uusdc.

```
// few blocks passed, debt should increase for user_1
let debt = red_bank.query_user_debt(&mut mock_env, &user_1, denom: "uusdc");

// repay full debt for user_1
let repayed = debt.amount; // it should be 5_000_069 uusdc
assert_eq!(repayed, Uint128::from(5_000_069u128));

red_bank.repay(&mut mock_env, sender: &user_1, coin(amount: repayed.u128(), denom: "uusdc")).unwrap();
```

3. After the repayment, the debt amount for the user is 0 and the values for the rates are the following:
   borrow_rate: 0.716667701657739867
   liquidity_rate: 0.344002276982931938

```
let debt = red_bank.query_user_debt(&mut mock_env, &user_1, denom: "uusdc");
assert_eq!(debt.amount, Uint128::zero());
assert_eq!(debt.amount_scaled, Uint128::zero());

let result = red_bank.query_market(&mut mock_env, denom: "uusdc");
assert_eq!(result.borrow_rate, Decimal::from_ratio(716667701657739867u128, 1000000000000000000u128));
assert_eq!(result.liquidity_rate, Decimal::from_ratio(344002276982931938u128, 1000000000000000000u128));
```

Scenario 2: User repays more than his full debt

1. User borrows 5_000_000 uusdc.

```
// user_1 borrows some usdc (no usdc in the account before)
let borrowed_usdc: u128 = 5_000_000u128;
red_bank.borrow(&mut mock_env, sender: &user_1, denom: "uusdc", amount: borrowed_usdc).unwrap();
```

2. Now, instead of repaying 5_000_069 uusdc, the user repays 1_000_000_000 uusdc (much more than his debt).

```
// repay full debt for user_1 with a huge excess
let repayed: u128 = 1_000_000_000u128;

red_bank.repay(&mut mock_env, sender: &user_1, coin(amount: repayed, denom: "uusdc")).unwrap();
```

3. After the repayment, the debt amount for the user is 0 and the excess amount is refunded. However, the values for the rates are the following:
   borrow_rate: 0.550139885552314804 (less than previous scenario)
   liquidity_rate: 0.44195886736363317 (more than previous scenario)

```
let debt = red_bank.query_user_debt(&mut mock_env, &user_1, denom: "uusdc");
assert_eq!(debt.amount, Uint128::zero());
assert_eq!(debt.amount_scaled, Uint128::zero());

let result = red_bank.query_market(&mut mock_env, denom: "uusdc");
assert_eq!(result.borrow_rate, Decimal::from_ratio(550139885552314804u128, 1000000000000000000u128));
assert_eq!(result.liquidity_rate, Decimal::from_ratio(44195886736363317u128, 1000000000000000000u128));
```

Code Location:

Listing 1: contracts/red-bank/src/execute.rs (Line 718)

```
715 market.decrease_debt(debt_amount_scaled_delta)?;
716 user.decrease_debt(deps.storage, &denom, debt_amount_scaled_delta)
 ↳ ?;
717
718 response = update_interest_rates(&deps, &env, &mut market, Uint128
 ↳ ::zero(), &denom, response)?;
719 MARKETS.save(deps.storage, &denom, &market)?;
```

Risk Level:

Likelihood - 5
Impact - 4

Recommendation:

Update the logic of repay function to use liquidity_taken = **refund_amount** (instead of 0).

Remediation plan:

SOLVED: The issue was fixed in commit 1dbe2ba.

3.2 (HAL-02) UPDATE OF LOAN LIMIT COULD PRODUCE UNDESIRED CONSEQUENCES TO C2C PARTNERS - MEDIUM

Description:

update_uncollateralized_loan_limit function in **red-bank** contract allows the owner to update the uncollateralized loan limits for C2C partners. However, if the limit is updated to 0 in a specific denom for a partner, its uncollateralized state changes to **false**. As a consequence, the following unexpected scenarios could arise:

- If the update was made mistakenly and the owner would want to change the new limit to a real one, the logic of the function does not allow it until the partner repays the entire debt of the affected denom.
- If the C2C partner has previously deposited collaterals, its health factor could be damaged and its position prone to be liquidated.
- On the other hand, if the C2C partner does not have deposited collaterals and now tries to do it, the amount should be large enough (because of its damaged health factor); otherwise, its position could be immediately liquidated.

Code Location:

Listing 2: contracts/red-bank/src/execute.rs

```
289 // Check that the user has no collateralized debt
290 let current_limit = UNCOLLATERALIZED_LOAN_LIMITS
291     .may_load(deps.storage, (&user_addr, &denom))?
292     .unwrap_or_else(Uint128::zero);
293 let current_debt = DEBTS
294     .may_load(deps.storage, (&user_addr, &denom))?
295     .map(|debt| debt.amount_scaled)
296     .unwrap_or_else(Uint128::zero);
297 if current_limit.is_zero() && !current_debt.is_zero() {
```

```
298     return Err(ContractError::UserHasCollateralizedDebt {});
299 }
300
301 UNCOLLATERALIZED_LOAN_LIMITS.save(deps.storage, (&user_addr, &
 ↳ denom), &new_limit)?;
```

Risk Level:

Likelihood - 3
Impact - 3

Recommendation:

Update the logic of update_uncollateralized_loan_limit function to throw an error message if owner tries to set new_limit to 0 for a C2C partner which current_debt is different from 0, as shown in the sample code below:

Listing 3: Sample code

```
1 if !current_limit.is_zero() && new_limit.is_zero()
2 && !current_debt.is_zero() {
3     return Err(ContractError::UserHasUncollateralizedDebt {});
4 }
```

Remediation plan:

SOLVED: The issue was fixed in commit 34d9f5b.

3.3 (HAL-03) PRIVILEGED ADDRESS CAN BE TRANSFERRED WITHOUT CONFIRMATION - LOW

Description:

An incorrect use of the update_config function in **red-bank** contract can set owner to an invalid address and inadvertently lose control of the contract, which cannot be undone in any way. Currently, the owner of the contract can change **owner address** using the aforementioned function in a single transaction and without confirmation from the new address.

It is worth noting that the likelihood for this to happen is very low because the contract is intended to be owned by governance indefinitely, who is the responsible one for this operation.

Code Location:

Listing 4: contracts/red-bank/src/execute.rs (Line 82)

```
69 if info.sender != config.owner {
70   return Err(MarsError::Unauthorized {}.into());
71 }
72
73 // Destructuring a structs fields into separate variables in order
↳  to force
74 // compile error if we add more params
75 let CreateOrUpdateConfig {
76   owner,
77   address_provider,
78   close_factor,
79 } = new_config;
80
81 // Update config
82 config.owner = option_string_to_addr(deps.api, owner, config.owner
↳ )?;
```

Risk Level:

Likelihood - 1
Impact - 4

Recommendation:

It is recommended to split **ownership transfer** functionality into set_owner and accept_ownership functions. The latter function allows the transfer to be completed by the recipient.

Remediation plan:

SOLVED: The issue was fixed in commit bce1663 by using the mars_owner crate for secure ownership transfer.

3.4 (HAL-04) LIQUIDATORS COULD DRAIN COLLATERALS WITHOUT SPENDING COINS - LOW

Description:

In some edges scenarios, `liquidation_compute_amounts` function in **red-bank** contract allows that the value of `debt_amount_to_repay` is `0`. As a consequence, some liquidators could drain collaterals and all their coins would be refunded, i.e.: without spending coins.

It is worth noting that the likelihood for this to happen is very limited because it would require that the user's collateral is very low, or the debt price is very high.

Step-by-step example:

Initial situation:

- `user_collateral_amount_scaled` = 320_000_000
- `user_debt_amount` = 800
- `sent_debt_amount` = 2
- `collateral_price` = 1.0
- `debt_price` = 300.0
- `close_factor` = 0.5
- `liquidation_bonus` = 0.1
- `SCALING_FACTOR` = 1_000_000
- `liquidity_index` = 1.0

In liquidation_compute_amounts function:

1. `debt_amount_to_repay` = min(2 , 0.5 * 800) = 2
2. `collateral_amount_to_liquidate` = (2 * 300.0 * (1 + 0.1)) / 1.0 = 660

3. collateral_amount_to_liquidate_scaled = (660 * 1_000_000) / 1.0 = 660_000_000
4. if collateral_amount_to_liquidate_scaled > user_collateral_amount_scaled => collateral_amount_to_liquidate_scaled = 320_000_000
5. collateral_amount_to_liquidate = (320_000_000 * 1.0) / 1_000_000 = 320
6. debt_amount_to_repay = (320 * 1.0) / (300 * (1 + 0.1)) = 0
7. refund_amount = 2 - 0 = 2

Code Location:

Listing 5: contracts/red-bank/src/execute.rs (Lines 974-980)

```
965 // If collateral amount to liquidate is higher than
 ↳ user_collateral_balance,
966 // liquidate the full balance and adjust the debt amount to repay
 ↳ accordingly
967 if collateral_amount_to_liquidate_scaled >
 ↳ user_collateral_amount_scaled {
968  collateral_amount_to_liquidate_scaled =
 ↳ user_collateral_amount_scaled;
969  collateral_amount_to_liquidate = get_underlying_liquidity_amount(
970    collateral_amount_to_liquidate_scaled,
971    collateral_market,
972    block_time,
973  )?;
974  debt_amount_to_repay = math::divide_uint128_by_decimal(
975    math::divide_uint128_by_decimal(
976      collateral_amount_to_liquidate * collateral_price,
977      debt_price,
978    )?,
979    Decimal::one() + collateral_market.liquidation_bonus,
980  )?;
981 }
982
983 let refund_amount = sent_debt_amount - debt_amount_to_repay;
```

Risk Level:

Likelihood - 1
Impact - 3

Recommendation:

Update the logic of `liquidation_compute_amounts` function to throw an error message if the value of `debt_amount_to_repay` is `0`.

Remediation plan:

SOLVED: The issue was fixed in commit `a61eec1`.

3.5 (HAL-05) LIQUIDATION THRESHOLD VALUE CAN BE CHANGED UNRESTRICTEDLY - LOW

Description:

update_asset function in **contracts/red-bank/src/execute.rs** allows the unrestricted modification of liquidation_threshold value. If mistakenly done, it would imply that many debt positions can be liquidated in unfair fashion, which severely affects borrowers' lending strategy.

It is worth noting that the likelihood for this to happen is very low because **red-bank** contract is intended to be owned by governance indefinitely, who is the responsible one for this operation.

Code Location:

update_asset function updates the value of liquidation_threshold and calls validate function to verify if this new value is appropriate:

Listing 6: contracts/red-bank/src/execute.rs (Lines 242-243,252)

```
239 let mut updated_market = Market {
240   max_loan_to_value: max_loan_to_value.unwrap_or(market.
 ↳ max_loan_to_value),
241   reserve_factor: reserve_factor.unwrap_or(market.reserve_factor),
242   liquidation_threshold: liquidation_threshold
243     .unwrap_or(market.liquidation_threshold),
244   liquidation_bonus: liquidation_bonus.unwrap_or(market.
 ↳ liquidation_bonus),
245   interest_rate_model: interest_rate_model.unwrap_or(market.
 ↳ interest_rate_model),
246   deposit_enabled: deposit_enabled.unwrap_or(market.
 ↳ deposit_enabled),
247   borrow_enabled: borrow_enabled.unwrap_or(market.borrow_enabled),
248   deposit_cap: deposit_cap.unwrap_or(market.deposit_cap),
249   ..market
250 };
```

```
251
252 updated_market.validate()?;
```

validate function verifies if the value of liquidation_threshold is lesser or equal than 1 and also greater than max_loan_to_value. However, it does not verify if the new value has a significant difference with the previous one:

Listing 7: packages/outpost/src/red_bank/market.rs (Lines 78,82)

```
75 pub fn validate(&self) -> Result<(), MarsError> {
76  decimal_param_lt_one(self.reserve_factor, "reserve_factor")?;
77  decimal_param_le_one(self.max_loan_to_value, "max_loan_to_value")
↳ ?;
78  decimal_param_le_one(self.liquidation_threshold, "
↳ liquidation_threshold")?;
79  decimal_param_le_one(self.liquidation_bonus, "liquidation_bonus")
↳ ?;
80
81  // liquidation_threshold should be greater than max_loan_to_value
82  if self.liquidation_threshold <= self.max_loan_to_value {
83   return Err(MarsError::InvalidParam {
84    param_name: "liquidation_threshold".to_string(),
85    invalid_value: self.liquidation_threshold.to_string(),
86    predicate: format!("> {} (max LTV)", self.max_loan_to_value),
87   });
88  }
89
90  self.interest_rate_model.validate()?;
91
92  Ok(())
93 }
```

Risk Level:

Likelihood - 1
Impact - 3

Recommendation:

Update the logic of update_asset function to include a **ramp change schema** for liquidation_threshold that includes the following criteria:

- New value should not differ more than a predefined amount / percentage from the previous one, e.g.: 90%
- Minimum time window between changes, e.g.: 24 hours

As a reference, **ramp change schema** for Curve protocol is included in the following link.

Remediation plan:

RISK ACCEPTED: The Mars Protocol team accepted the risk of this finding.

3.6 (HAL-06) DENOM IS NOT VALIDATED IN SOME FUNCTIONS - LOW

Description:

The format of denom argument in some functions from **red-bank** contract is not validated. As a consequence, the following unexpected scenarios could arise during the operations:

- Invalid markets initialized
- Coins stuck in invalid markets
- Unnecessary expense of gas in certain operations

Code Location:

Listing 8: Affected resources

```
Functions:
==========
init_asset:
  + argument: denom

update_asset:
  + argument: denom

update_uncollateralized_loan_limit:
  + argument: denom

withdraw:
  + argument: denom

borrow:
  + argument: denom

liquidate:
  + argument: collateral_denom

update_asset_collateral_status:
  + argument: denom
```

Risk Level:

Likelihood - 1
Impact - 3

Recommendation:

For the functions mentioned above, validate the format of denom argument: length and use of allowed characters (e.g.: alphanumeric, /, etc.).

Remediation plan:

SOLVED: The issue was fixed in commit c0b69d4 by applying the denom validation in init_asset function, the risk is minimized in such a way that the impact of scenarios mentioned above become negligible.

3.7 (HAL-07) USERS COULD LIQUIDATE WITHOUT RECEIVING COLLATERALS IN RETURN - INFORMATIONAL

Description:

In some edges scenarios, `liquidation_compute_amounts` function in **red-bank** contract allows that the value of `collateral_amount_to_liquidate` is `0`. As a consequence, some users could liquidate without receiving collaterals in return.

It is worth noting that the likelihood for this to happen is very limited because it would require that the debt amount to repay is very low, or the collateral price is very high.

Step-by-step example:

Initial situation:

- `user_debt_amount` = 20
- `sent_debt_amount` = 30
- `collateral_price` = 12.0
- `debt_price` = 1.0
- `close_factor` = 0.5
- `liquidation_bonus` = 0.1

In liquidation_compute_amounts function:

1. `debt_amount_to_repay` = min(30 , 0.5 * 20) = 10
2. `collateral_amount_to_liquidate` = (10 * 1.0 * (1 + 0.1)) / 12.0 = 0

Code Location:

Listing 9: contracts/red-bank/src/execute.rs (Lines 958-961)

```
953 // Debt: Only up to a fraction of the total debt (determined by
 ↳ the close factor) can be
954 // repayed.
955 let mut debt_amount_to_repay = min(sent_debt_amount, close_factor
 ↳ * user_debt_amount);
956
957 // Collateral: debt to repay in base asset times the liquidation
 ↳ bonus
958 let mut collateral_amount_to_liquidate = math::
 ↳ divide_uint128_by_decimal(
959  debt_amount_to_repay * debt_price * (Decimal::one() +
 ↳ collateral_market.liquidation_bonus),
960  collateral_price,
961 )?;
962 let mut collateral_amount_to_liquidate_scaled =
963  get_scaled_liquidity_amount(collateral_amount_to_liquidate,
 ↳ collateral_market, block_time)?;
```

Risk Level:

Likelihood - 1
Impact - 2

Recommendation:

Update the logic of liquidation_compute_amounts function to throw an error message if the value of collateral_amount_to_liquidate is 0.

Remediation plan:

SOLVED: The issue was fixed in commit a61eec1.

3.8 (HAL-08) REPAYMENT OF UNCOLLATERALIZED LOANS COULD ERRONEOUSLY FAIL - INFORMATIONAL

Description:

repay function in **red-bank** contract does not allow repaying on behalf of the **C2C partners**, which are the ones that have uncollateralized loan limits greater than 0. To achieve this goal, the code includes a conditional expression that throws a CannotRepayUncollateralizedLoanOnBehalfOf error message if the on_behalf_of argument is set with a **Some(<String>)** value and its uncollateralized_loan_limit is greater than 0.

However, if a **C2C partner** tries to repay its own debt and (mistakenly or not) sets the on_behalf_of argument with its **own address**, the operation will throw the error message mentioned above despite being a valid repayment scenario.

Code Location:

Listing 10: contracts/red-bank/src/execute.rs (Lines 658-660)

```
645 pub fn repay(
646  deps: DepsMut,
647  env: Env,
648  info: MessageInfo,
649  on_behalf_of: Option<String>,
650  denom: String,
651  repay_amount: Uint128,
652 ) -> Result<Response, ContractError> {
653  let user_addr: Addr;
654  let user = if let Some(address) = on_behalf_of {
655    user_addr = deps.api.addr_validate(&address)?;
656    let user = User(&user_addr);
657    // Uncollateralized loans should not have 'on behalf of'
 ↳ because it creates accounting complexity for them
658    if !user.uncollateralized_loan_limit(deps.storage, &denom)?.
 ↳ is_zero() {
```

```
659         return Err(ContractError::
↳ CannotRepayUncollateralizedLoanOnBehalfOf {});
660       }
661       user
662     } else {
663       User(&info.sender)
664     };
```

Risk Level:

Likelihood - 2
Impact - 1

Recommendation:

Update the conditional expression of `repay` function to also verify if `info.sender` is different from `user_addr` before throwing a `CannotRepayUncollateralizedLoanOnBehalfOf` error message, as shown in the sample code below:

Listing 11: Sample code (Line 1)

```
1 if !user.uncollateralized_loan_limit(deps.storage, &denom)?.
↳ is_zero() && info.sender != user_addr {
2  return Err(ContractError::
↳ CannotRepayUncollateralizedLoanOnBehalfOf {});
3 }
```

Remediation plan:

ACKNOWLEDGED: The Mars Protocol team acknowledged this finding.

THANK YOU FOR CHOOSING

// HALBORN