IRONNODE

ironnode.io

Aeroscraper

Smart Contract Security Audit

by **IronNode** from **March 20 - April 10, 2024**

1	Executive overview	3
1.1	Introduction	4
1.2	Audit summary	4
	Objectives of the audit	5
1.3	Scope	5
1.4	Assessment summary & findings overview	6
2	Application Flow Analysis	7
2.1	User Role	8
	Description	8
	Functions	8
2.2	Admin Role	9
	Description	9
	Functions	9
3	Oracle Helper	10
3.1	Admin Role	10
	Functions	10
4	Findings	11
4.1	Unknown collateral types in the get_trove_liquidity function leads to panic	12
	Code location	12
	Recommendation	13

4.2 Calling instructions with non-existing troves leads to ambiguous errors 14

Code location 14

Recommendation 16

4.3 Optimization required for redundant checks 17

Code location 17

Recommendation 17

4.4 Late check and ambiguous error on liquidate function 18

Code location 18

Recommendation 18

4.5 Missing validate_denom check in SetData and SetDataBatch instructions 19

Code location 19

Recommendation 19

4.6 Missing admin update oracle_helper contract description 20

Recommendation 20

5 Manual testing 21

5.1 Open trove 22

5.2 Add collateral 22

5.3 Borrow loan 22

5.4 Repay loan 23

5.5 Liquidate troves 23

Executive overview

1 Executive overview

1.1 Introduction

Aeroscraper is a decentralized lending-borrowing protocol that innovates in the sphere of digital finance by providing interest-free, over-collateralized stablecoin and DeFi loans. Designed with a user-centric approach, Aeroscraper facilitates a fully automated and governance-free environment where users can engage in unauthorized lending and borrowing activities without traditional financial intermediaries.

This decentralized platform allows individuals to deposit collateral securely and obtain loans in stablecoins that are pegged to the US dollar. Uniquely, Aeroscraper distinguishes itself by offering these financial services at a 0% interest rate, opting instead for a one-time transaction fee, which mitigates the usual financial burdens associated with variable interest rates.

In response to a request from Aeroscraper, IronNode, conducted a thorough security audit of the Aeroscraper protocol to ensure its integrity, security, and user trustworthiness. The audit was carried out from March 20th, 2024 to April 10th, 2024.

1.2 Audit summary

The security audit team at IronNode was allocated three weeks to conduct an extensive review of the Aeroscraper protocol. A dedicated team, including a lead security engineer with profound expertise in blockchain technology, smart contract security, and cybersecurity, spearheaded the audit.

1.2.1 Objectives of the audit

- Ensure the Robustness of Business Logic: Validate the underlying business logic of Aeroscraper for any security flaws that could be exploited maliciously.
- Stability and Security of DeFi Loans: Investigate the protocol's mechanisms for issuing DeFi loans to ensure they are free from vulnerabilities that could affect stability and security.
- Collateral Management: Assess the security measures in place for managing user collateral and preventing unauthorized access or losses.
- Fee Structure Integrity: Verify that the one-time fee model is implemented securely and operates as intended without hidden risks.
- Smart Contract Examination: Conduct in-depth testing of the smart contracts involved in loan issuance and collateral management to identify potential vulnerabilities, such as reentrancy attacks, overflow bugs, and improper exception handling.
- User Interaction Security: Evaluate the security protocols concerning user interactions with the platform to ensure that sensitive user data is handled securely.
- External Dependency Security: Review all external libraries and dependencies used in the protocol to confirm they do not introduce security vulnerabilities.
- Audit Trails and Monitoring: Check the adequacy of the logging and monitoring mechanisms that help in identifying and mitigating potential threats in real-time.

1.3 Scope

1.3.1 Code repositories

Aeroscaper
Branch: injective-v2
Commit ID: https://github.com/findolor/aeroscraper-contracts commit/4b1365c8930726a0effc74edf3124841bd9e17ed

1.4 Assessment summary & findings overview

CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL
0	0	0	0	6

Security analysis	Risk Level	Remediation date
Unknown collateral types in the get_trove_liquidity function leads to panic	INFORMATIONAL	ACKOWLEDGED
Calling instructions with non-existing troves leads to ambiguous errors	INFORMATIONAL	ACKOWLEDGED
Optimization required for redundant checks	INFORMATIONAL	ACKOWLEDGED
Late check and ambiguous error on liquidate function	INFORMATIONAL	ACKOWLEDGED
Missing validate_denom check in SetData and SetDataBatch instructions	INFORMATIONAL	ACKOWLEDGED
Missing admin update oracle_helper contract description	INFORMATIONAL	ACKOWLEDGED

Application Flow Analysis

2 Aeroscraper Protocol

This section of the report provides an in-depth analysis of the application flow for the Aeroscraper Protocol contract. The analysis covers the logical flow of the code, with an emphasis on two primary roles: User and Admin.

A Trove is where you take out and maintain your loan. Each Trove is linked to an address and each address can have just one Trove. Every trove has some amount of collateral and debt saved. An user cannot create a trove below minimum collateral ratio.

2.1 User Role

The user role encompasses wallet addresses that interact with the Aeroscraper Protocol. Users can open, update, remove their troves. Stake their stable coins in the stability pool to claim liquidation rewards pro-rata. Redeem their stable coins into native tokens set by the protocol. And also trigger liquidations (under necessary conditions) to gain rewards.

2.1.1 Functions

- execute_open_trove: Allows users to create a new trove if it wasn't present before and the minimum collateral ratio is met. This functionality mints new stable coins. Created trove is stored in the system and a response is emitted on successful transaction.
- execute_add_collateral: Allows users to add collateral to their trove. A response is emitted on successful transaction.
- execute_remove_collateral: Allows users to remove collateral from their trove (if minimum collateral ratio is met), protocol transfers the collateral amount back to the user. A response is emitted on successful transaction.
- execute_borrow_loan: Allows users to borrow loan from the protocol (if minimum collateral ratio is met), this functionality mints new stable coins. A response is emitted on successful transaction.
- execute_repay_loan: Allows users to repay their loan to the protocol. A response is emitted on successful transaction.
- execute_stake: Allows users to deposit stable coins from their wallet to earn rewards. A response is emitted on successful transaction.
- execute_unstake: Allows users to withdraw their deposited stable coins to their wallet. A response is emitted on successful transaction.
- execute_withdraw_liquidation_gains: Allows users to withdraw their liquidation gains, based on their pool share. A response is emitted on successful transaction.
- execute_redeem: Allows users convert stable coins to collateral. A response is emitted on successful transaction.
- execute_liquidate_troves: Allows users to initiate liquidation for troves below minimum collateral ratio. The initiator receives reward for this service. A response is emitted on successful transaction.

2.2 Admin Role

The admin role is the entity responsible for managing 2 configuration settings over the Aeroscraper Protocol: Setting the possible minimum collateral ratio value, and setting the percentage of protocol fee.

2.2.1 Functions

- SetMinimumCollateralRatio: Allows admin to set minimum collateral ratio value for the protocol.
- SetProtocolFee: Allows admin to set protocol fee value for the protocol.

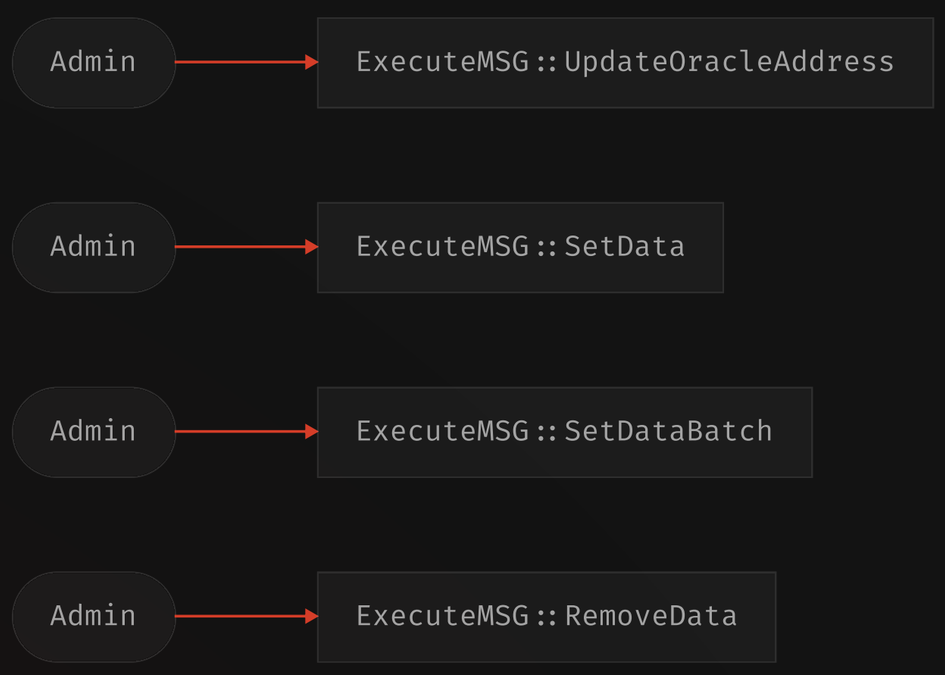
3 Oracle Helper

This section of the report provides an in-depth analysis of the application flow for the oracle_helper contract. The analysis covers the logical flow of the code, consisting single Admin role.

3.1 Admin Role

Functions

- UpdateOracleAddress: Allows admin to set oracle address value for the oracle_helper contract.
- SetData: Allows admin to set data for a denom for the oracle_helper contract.
- SetDataBatch: Allows admin to set data for multiple denoms for the oracle_helper contract.
- RemoveData: Allows admin to remove denom data from the oracle_helper contract.



Findings

4 Findings

4.1 Unknown Collateral Types In The `Get_trove_liquidity` Function Leads To Panic

The `get_trove_liquidity` function calculates the liquidity of a trove. It takes prices from the price list and parameter multiplies them by the token amounts in the trove to calculate the total liquidity. When an unwanted/unexpected type of collateral is sent to the contract, a `panic` can occur. Program is trying to access a value in the `collateral_prices` HashMap that may not exist. Using `unwrap()` directly on the result of `HashMap.get()` can cause the program to panic if the key does not exist in the map, which seems to be the case here. This approach, if it does not meet expected conditions, leads to irretrievable loss of used gas. Following an approach that ensures the program returns clear and understandable error messages for unwanted collateral types instead of using `unwrap()`, helps users better understand the reason for the error.

4.1.1 Code location

`protocol/src/trove_helpers.rs (Line 39)`

```
pub fn get_trove_liquidity(
    storage: &dyn Storage,
    collateral_prices: &HashMap<String, PriceResponse>,
    owner: Addr,
) -> Result<Vec<LiquidityData>, ContractError> {
    //...snipped...
            let price_data = collateral_prices.get(&denom).unwrap();
    //...snipped...
}
```

4.1.2 Recommendation

Safe Handling of Missing Data: Instead of unwrapping directly, we now use `match` to safely handle the possibility that `collateral_prices.get(&denom)` might not return a value. This prevents the function from panicking and instead allows it to return a specific error if the price data for a given collateral type is not found.

Custom Error: Include a potential `MissingPriceData` error in the `ContractError` enum (you may need to add this if it doesn't exist), enhancing error specificity and helping diagnose issues more accurately during operation.

```
pub fn get_trove_liquidity(
    storage: &dyn Storage,
    collateral_prices: &HashMap<String, PriceResponse>,
    owner: Addr,
) -> Result<Vec<LiquidityData>, ContractError> {
    let res = USER_COLLATERAL_AMOUNT
        .prefix(owner.clone())
        .range(storage, None, None, Order::Ascending)
        .map(|item| {
            let (denom, amount) = item?;
            match collateral_prices.get(&denom) {
                Some(price_data) => {
                    let decimal = match price_data.decimal {
                        6 => DECIMAL_FRACTION_6,
                        18 => DECIMAL_FRACTION_18,
                        _ => return Err(ContractError::InvalidDecimal {}),
                    };
                    let liquidity =
Decimal256::new(amount.checked_mul(Uint256::from_u128(DECIMAL_FRACTION_18))?)
                            .checked_mul(Decimal256::new(Uint256::from_u128(
                                price_data.price as u128,
                            )))?
                            .checked_div(Decimal256::new(Uint256::from_u128(decimal)))?
                            .checked_div(Decimal256::new(Uint256::from_u128(100000000)))?;
                    Ok(LiquidityData {
                        denom,
                        liquidity,
                        decimal: price_data.decimal,
                    })
                },
                None => return Err(ContractError::MissingPriceData { denom }),
            }
        })
        .collect::<Result<Vec<LiquidityData>, ContractError>>()?;
    Ok(res)
}
```

4.2 Calling Instructions With Non-Existing Troves Leads To Ambiguous Errors

When attempting operations such as adding collateral to a trove that is not registered in the system, program throws ambiguous errors, due to improper handling of non-existing trove IDs. These late checks also leads to gas waste. All of these kind of checks could be possibly broken into two parts.

protocol/src/execute/borrow_loan.rs (Lines 29-31)

```
// New total debt amount
let new_total_debt_amount = TOTAL_DEBT_AMOUNT
 .load(deps.storage)?
 .checked_add(loan_amount)?;
TOTAL_DEBT_AMOUNT.save(deps.storage, &new_total_debt_amount)?;

let new_debt_amount = USER_DEBT_AMOUNT
 .load(deps.storage, info.sender.clone())?
 .checked_add(loan_amount)?;
USER_DEBT_AMOUNT.save(deps.storage, info.sender.clone(), &new_debt_amount)?;
let collateral_amount = USER_COLLATERAL_AMOUNT
 .load(deps.storage, (info.sender.clone(), trove_denoms[0].clone()))?;

// Update liquidity threshold for the user
let ratio = get_trove_icr(deps.storage, &collateral_prices, info.sender.clone())?;
let nominal_ratio = get_trove_nicr(collateral_amount, new_debt_amount)?;
check_trove_icr_with_ratio(deps.storage, ratio)?;
```

protocol/src/execute/add_collateral.rs (Lines 62-68)

```
// Calculate new total collateral amount
let new_total_collateral_amount = old_total_collateral_amount.checked_add(collateral_amount)?;
// Update total collateral amount
TOTAL_COLLATERAL_AMOUNT.save(
 deps.storage,
 collateral_denom.clone(),
 &new_total_collateral_amount,
)?;

// Existing user collateral and debt amount
let new_collateral_amount = USER_COLLATERAL_AMOUNT
 .load(
  deps.storage,
  (info.sender.clone(), collateral_denom.clone()),
 )
 .unwrap_or(Uint256::zero())
 .checked_add(collateral_amount)?;
USER_COLLATERAL_AMOUNT.save(
 deps.storage,
 (info.sender.clone(), collateral_denom.clone()),
 &new_collateral_amount,
)?;
let loan_amount = USER_DEBT_AMOUNT.load(deps.storage, info.sender.clone())?;
```

protocol/src/execute/remove_collateral.rs (Lines 36-41)

```
TOTAL_COLLATERAL_AMOUNT.save(
 deps.storage,
 collateral_denom.clone(),
 &new_total_collateral_amount,
)?;

// Existing user collateral and debt amount
let new_collateral_amount = USER_COLLATERAL_AMOUNT
 .load(
  deps.storage,
  (info.sender.clone(), collateral_denom.clone()),
 )?
 .checked_sub(collateral_amount)?;

// New collateral amount for the user
USER_COLLATERAL_AMOUNT.save(
 deps.storage,
 (info.sender.clone(), collateral_denom.clone()),
 &new_collateral_amount,
)?;
```

protocol/src/execute/repay_loan.rs (Line 36)

```
// Protocol and user debts
let old_total_debt_amount = TOTAL_DEBT_AMOUNT.load(deps.storage)?;
let old_debt_amount = USER_DEBT_AMOUNT.load(deps.storage, sender.clone())?;

// New total debt amount
let new_total_debt_amount = old_total_debt_amount.checked_sub(amount)?;
TOTAL_DEBT_AMOUNT.save(deps.storage, &new_total_debt_amount)?;

if amount > old_debt_amount {
 return Err(ContractError::InvalidAmount {});
};
```

4.2.1 Recommendation

Using `may_load`: This function is used to simplify the error handling process by integrating the `Option` directly into the result.

Error Handling: By using a match statement on the result of `may_load`, you can cleanly handle cases where the debt amount does not exist (`None`).

```
// Load the current debt amount from storage using may_load for more idiomatic handling.
let current_debt_amount = USER_DEBT_AMOUNT.may_load(deps.storage, info.sender.clone())?;

// Check if the debt amount exists and handle the case where it does not.
let current_debt_amount = match current_debt_amount {
    Some(debt) => debt,
    None => return Err(ContractError::NonExistentTrove {
        sender: info.sender.to_string(),
    }),
};
```

4.3 Optimization Required For Redundant Checks

The `if !NODES.has(storage, id.clone())` check in the `reinsert_trove` function causes unnecessary gas consumption when repeated in the `remove_trove` and `insert_trove functions`. Optimizing these iterative checks can reduce gas consumption.

3.3.1 Code location

`protocol/src/sorted_troves.rs (Lines 104-106)`

```
pub fn reinsert_trove(
    storage: &mut dyn Storage,
    id: Addr,
    icr: Decimal256,
    prev_id: Option<Addr>,
    next_id: Option<Addr>,
) -> Result<(), ContractError> {
    if !NODES.has(storage, id.clone()) {
        return Err(ContractError::Unauthorized {});
    }
    if icr.is_zero() {
        return Err(ContractError::Unauthorized {});
    }

    remove_trove(storage, id.clone())?;
    insert_trove(storage, id, icr, prev_id, next_id)?;

    Ok(())
}
```

3.3.2 Recommendation

This redundant validation in this function here could be removed.

4.4 Late Check And Ambiguous Error On Liquidate Function

The `execute_liquidate_troves()` function performs the liquidation of troves specified in the lists sent by users. However, checking conditions after effects are made leads to the wastage of gas spent on the transactions and can cause security vulnerabilities. This situation is generally considered a method to be avoided in smart contract development practices. Performing all necessary validations at the beginning of transactions both optimizes gas consumption and enhances contract security.

4.4.1 Code location

`protocol/src/execute/liquidate_troves.rs (Lines 61-63)`

```
let mut collateral_attributes: Vec<Attribute> = vec![];
let mut debt_attributes: Vec<Attribute> = vec![];

// Check list lenghts
if liquidation_list.is_empty() {
 return Err(ContractError::InvalidList {});
}

let mut is_liquidation = false;
let mut is_stability_pool_liquidation = false;
let mut is_trove_distribution = false;

let mut troves_to_remove: Vec<Addr> = vec![];
let mut troves_to_update: HashMap<Addr, (Decimal256, Node)> = HashMap::new();
```

4.4.2 Recommendation

Applying Checks-Effects-Interactions pattern, and moving `liquidation_list.is_empty()` check just at the start of the function.

4.5 Missing `Validate_denom` Check In `SetData` And `SetDataBatch` Instructions

It has been identified that in the `SetData` and `SetDataBatch` operations, the denom parameters are not verified using the `cw_denom` library. This oversight can pose a risk in terms of data integrity and security.

4.5.1 Code location

`contracts/oracle_helper/src/contract.rs (Lines 50-61)`

```
ExecuteMsg::SetData { data } => {
 check_admin(&deps, &info)?;
 DATA.save(deps.storage, data.denom.clone(), &data)?;
 Ok(Response::new())
}
ExecuteMsg::SetDataBatch { data } => {
 check_admin(&deps, &info)?;
 for item in data.iter() {
  DATA.save(deps.storage, item.denom.clone(), item)?;
 }
 Ok(Response::new())
}
```

4.5.2 Recommendation

Integrate the `cw_denom` library to validate `denom` parameters in both `SetData` and `SetDataBatch` operations. This will ensure that all `denom` values conform to expected standards before any data is saved to storage, enhancing both security and data integrity.

```
use cw_denom::validate_denom;
//...snipped...
ExecuteMsg::SetData { data } => {
    check_admin(&deps, &info)?;
    validate_denom(&data.denom)?; // Validate denom using cw_denom library
    DATA.save(deps.storage, data.denom.clone(), &data)?;
    Ok(Response::new())
}
ExecuteMsg::SetDataBatch { data } => {
    check_admin(&deps, &info)?;
    for item in data.iter() {
        validate_denom(&item.denom)?; // Validate each denom in the batch
        DATA.save(deps.storage, item.denom.clone(), item)?;
    }
    Ok(Response::new())
}
```

4.6 Missing Admin Update `Oracle_helper` Contract Description

When an admin change is necessary, the inability to effectively transfer contract management leads to operational risks.

4.6.1 Recommendation

Implement an admin update instruction for the contract.

Manual testing

5 Manual testing

5.1 Open trove

- Test with valid inputs for creating a new trove, ensuring it is stored correctly and the corresponding event is emitted.
- Test storage load/saves.
- Attempt to create a trove for an existing address, expecting an TroveExists error.
- Attempt to create a trove without funds, expecting a MissingFunds error.
- Attempt to create a trove with more than one fund, expecting a ExtraFunds error.
- Attempt to create a trove with low loan amount, expecting an LoanAmountBelowMinimum error.
- Attempt to create a trove with low icr, expecting an CollateralBelowMinimum error.
- Attempt to create a trove with an unwanted collateral type.
- Attempt to create a trove with unwanted node id's.

5.2 Add collateral

- Test with valid inputs for creating a new trove, ensuring it is stored correctly and the corresponding event is emitted.
  - Single collateral
  - Multi collateral
- Test storage load/saves.
- Attempt to update the trove with final low collateral amount, expecting CollateralBelowMinimum
- Attempt to remove more collateral than what is available in the trove
- Attempt to remove collateral from a non-existing trove

5.3 Borrow loan

- Test with valid inputs for borrowing against an existing trove, ensuring the loan amount is correctly added to the trove's debt and the corresponding event is emitted.
- Test storage load/saves.
- Attempt to borrow with an amount that leads to an ICR (Initial Collateral Ratio) below the minimum, expecting CollateralBelowMinimum and unchanged storage.
- Attempt to borrow an amount that exceeds the maximum allowed by the protocol
- Attempt to borrow with an account that does not have an existing trove

5.4 Repay loan

- Test with valid inputs for repaying part of a loan on an existing trove, ensuring the debt is reduced correctly and the corresponding event is emitted
  - Repay partially
  - Repay completely
- Test storage load/saves.
- Attempt to repay more than the existing debt
- Attempt to repay with a trove that does not exist

5.5 Liquidate troves

- Test with valid inputs
- Exhaust resources by passing a large `liquidation_list`
- Attempt to send an empty list
- Attempt to liquidate non liquidatable troves

IRONNODE

ironnode.io

Thank you for choosing us!