// HALBORN

A41 - SuperNova

CosmWasm Smart Contract Security Audit

Prepared by: **Halborn**

Date of Engagement: **October 20th, 2022 - November 9th, 2022**

Visit: **Halborn.com**

DOCUMENT REVISION HISTORY

VERSION	MODIFICATION	DATE	AUTHOR
0.1	Document Creation	10/20/2022	Gustavo Dutra
0.2	Document Updates	11/08/2022	Gustavo Dutra
0.3	Draft Version	11/11/2022	Gustavo Dutra
0.4	Draft Review	11/15/2022	Gabi Urrutia
1.0	Remediation Plan	12/14/2022	Gustavo Dutra
1.1	Remediation Plan Review	12/21/2022	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
Gustavo Dutra	Halborn	Gustavo.Dutra@halborn.com

EXECUTIVE OVERVIEW

1.1 INTRODUCTION

A41 engaged Halborn to conduct a security audit on their smart contracts beginning on October 20th, 2022 and ending on November 9th, 2022. The security assessment was scoped to the smart contracts provided to the Halborn team.

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 A41 team. The main ones are the following:

- Implement access control in the pair creation in the factory contract.
- Enforce a maximum slippage threshold when adding liquidity to the liquidity pairs.
- Implement a way to pause a liquidity pool in the contracts in case of emergency.
- Implement a two-step process for ownership transfer.

1.3 TEST APPROACH & METHODOLOGY

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.

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.

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

1.4 SCOPE

1. CosmWasm Smart Contracts
   (a) Repository: supernova-core-contract
   (b) Commit ID: 0de9dccd609417ec5b89b697c36a8c74414023d3
   (c) Contracts in scope:
      i. factory
      ii. pair
      iii. vault
      iv. token

2. ASSESSMENT SUMMARY & FINDINGS OVERVIEW

CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL
0	1	1	2	2

LIKELIHOOD

IMPACT

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

EXECUTIVE OVERVIEW

SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
(HAL-01) ACCESS CONTROL NOT ENFORCED FOR CREATING PAIRS	High	SOLVED - 11/27/2022
(HAL-02) SLIPPAGE NOT ENFORCED	Medium	RISK ACCEPTED
(HAL-03) POOL PAUSE MECHANISM NOT IMPLEMENTED	Low	SOLVED - 11/30/2022
(HAL-04) PRIVILEGED ADDRESS CAN BE TRANSFERRED WITHOUT CONFIRMATION	Low	SOLVED - 12/12/2022
(HAL-05) UNCHECKED ARITHMETIC	Informational	ACKNOWLEDGED
(HAL-06) INCOMPLETE DOCUMENTATION	Informational	ACKNOWLEDGED

FINDINGS & TECH DETAILS

3.1 (HAL-01) ACCESS CONTROL NOT ENFORCED FOR CREATING PAIRS - HIGH

Description:

As explicated in the provided documentation, the instruction for pair creation in the factory contract should be only called by the factory's owner.

However, the code does not reflect the documentation because there is no check if `msg.sender` is the factory's owner in the function `execute_create_pair`.

With this, anyone can create liquidity pools, that should be created by the factory's owner.

Proof of Concept:

In the following link is included a walkthrough video with the proof of concept.

Code Location:

Fragment of `execute_create_pair`:

Listing 1: contracts/factory/contract.rs

```
146 fn execute_create_pair(
147     deps: DepsMut,
148     env: Env,
149     pair_type: PairType,
150     asset_infos: [AssetInfo; 2],
151     init_params: Option<Binary>,
152 ) -> Result<Response, ContractError> {
153     asset_infos[0].check(deps.api)?;
154     asset_infos[1].check(deps.api)?;
155
156     if asset_infos[0] == asset_infos[1] {
157         return Err(ContractError::DoublingAssets {});
```

```
158     }
159
160     let config = CONFIG.load(deps.storage)?;
161
162     if PAIRS
163         .may_load(deps.storage, &pair_key(&asset_infos))?
164         .is_some()
165     {
166         return Err(ContractError::PairWasCreated {});
167     }
168
169     // Get pair type from config
170     let pair_config = PAIR_CONFIGS
171         .load(deps.storage, pair_type.to_string())
172         .map_err(|_| ContractError::PairConfigNotFound {})?;
173
174     // Check if pair config is disabled
175     if pair_config.is_disabled {
176         return Err(ContractError::PairConfigDisabled {});
177     }
178 }
```

Risk Level:

Likelihood - 5
Impact - 4

Recommendation:

The `msg.sender` should be checked in the function `execute_create_pair` to be the factory's owner.

Remediation Plan:

SOLVED: The issue was fixed in commit `7560afb6d5f53a911f3df92bb7fec88d1099ba02`.

3.2 (HAL-02) SLIPPAGE NOT ENFORCED - MEDIUM

Description:

In the pair liquidity providing functionality in the pair contract, there is no maximum threshold being asserted to the liquidity pool. This can severely affect users' amount of token received in return of the provided liquidity.

In the code, there is a commented call to a assert_slippage_tolerance function that has not been implemented.

Code Location:

Fragment of provide_liquidity:

Listing 2: contracts/pair/contract.rs (Lines 330,331)

```
312     if amp == MINIMUM_AMP && !pools[0].amount.is_zero() && !pools
 ↳ [1].amount.is_zero() {
313         let reserve_a = Uint256::from(pools[0].amount);
314         let reserve_b = Uint256::from(pools[1].amount);
315         let amount_a = Uint256::from(deposits[0]);
316         let amount_b = Uint256::from(deposits[1]);
317 
318         let real_amount_b = reserve_b * amount_a / reserve_a;
319         let real_amount_a = reserve_a * amount_b / reserve_b;
320         let optimal_amount_b = Uint128::try_from(real_amount_b)?;
321         let optimal_amount_a = Uint128::try_from(real_amount_a)?;
322 
323         if deposits[1] > optimal_amount_b {
324             deposits[1] = optimal_amount_b;
325         } else if deposits[0] > optimal_amount_a {
326             deposits[0] = optimal_amount_a;
327         }
328     }
329 
330     // Assert that slippage tolerance is respected
```

```
331     // assert_slippage_tolerance(&slippage_tolerance, &deposits, &
 ↳ pools)?;
332
333     // decimals of each token.
334     let token_precision_0 = query_token_precision(&deps.querier,
 ↳ pools[0].info.clone())?;
335     let token_precision_1 = query_token_precision(&deps.querier,
 ↳ pools[1].info.clone())?;
336     let greater_precision = token_precision_0.max(
 ↳ token_precision_1);
337 }
```

Risk Level:

Likelihood - 3
Impact - 3

Recommendation:

Enforce the use of a default maximum threshold when users add liquidity to the pairs.

Remediation Plan:

RISK ACCEPTED: The A41 team accepted the risk for this finding.

3.3 (HAL-03) POOL PAUSE MECHANISM NOT IMPLEMENTED - LOW

Description:

There is no mechanism implemented to pause the pools in the **pair** or **factory** contract.

It is important to have such mechanism in case an incident occur involving some pool in the protocol.

Risk Level:

Likelihood - 1
Impact - 4

Recommendation:

Implement a way to pause a liquidity pool in the contracts.

Remediation Plan:

SOLVED: The issue was fixed in commit f01837f816fb6b6eba8ca099dbc9030f14f9a261.

3.4 (HAL-04) PRIVILEGED ADDRESS CAN BE TRANSFERRED WITHOUT CONFIRMATION - LOW

Description:

An incorrect use of the update_owner function from the **factory** contract could set the owner to an invalid address, unwillingly losing control of the contract, which cannot be undone in any way. Currently, the owner of the contracts can change its address using the aforementioned function in a single transaction and without confirmation from the new address.

Code Location:

Listing 3: contracts/factory/contract.rs

```
110 fn execute_update_owner(
111     deps: DepsMut,
112     info: MessageInfo,
113     new_owner: String,
114 ) -> Result<Response, ContractError> {
115     let mut config = CONFIG.load(deps.storage)?;
116
117     if info.sender != config.owner {
118         return Err(ContractError::Unauthorized {});
119     }
120
121     config.owner = deps.api.addr_validate(new_owner.as_str())?;
122     CONFIG.save(deps.storage, &config)?;
123     Ok(Response::new().add_attribute("action", "update_owner"))
124 }
125     }
```

Risk Level:

Likelihood - 1
Impact - 4

Recommendation:

The update_owner function should follow a two steps process, being split into set_owner and accept_owner functions. The latter one requiring the transfer to be completed by the recipient, effectively protecting the contract against potential typing errors compared to single-step owner transfer mechanisms.

Remediation Plan:

SOLVED: The issue was fixed in commit 428f3ab553cc54bf5e50d65366e3de4eee466bd7.

3.5 (HAL-05) UNCHECKED ARITHMETIC - INFORMATIONAL

Description:

During our analysis, it was found the use of unchecked multiplications and divisions in multiple source files. This could potentially lead to panic the contracts' execution under some scenarios without showing users the reason of the error.

It is worth to mention that this issue was classified as informational because the flag overflow-checks has been set to **true** in the Cargo.toml file. Unsafe math was found in the following files and functions:

- contracts/vault/contract.rs: execute_claim
- contracts/pair/contract.rs: provide_liquidity
- contracts/pair/utils.rs: compute_offer_amount
- contracts/pair/utils.rs: assert_max_spread
- contracts/pair/utils.rs: start_changing_amp
- contracts/pair/utils.rs: compute_current_amp
- contracts/pair/utils.rs: get_share_in_assets

Code Location:

Fragment of execute_claim:

Listing 4: contracts/vault/contract.rs (Line 101)

```
92      update_pool(
93          &deps.querier,
94          deps.storage,
95          env.clone(),
96          &mut vault_info,
97          None,
98      )?;
99
100     let reward_per_share = vault_info.reward_per_share.div(
 ↳ multiplier());
```

```
101     let pending_reward = user_info.amount * reward_per_share -
 ↳ user_info.reward_debt;
102
103     // check pool has enough balance to send a reward
104     let pool_reward = query_asset_balance(
105         &deps.querier,
106         env.contract.address,
107         config.rewar
108 }
```

Fragment of execute_claim:

Listing 5: contracts/vault/contract.rs (Lines 46,47)

```
29 pub fn compute_offer_amount(
30     offer_pool: Uint128,
31     offer_precision: u8,
32     ask_pool: Uint128,
33     ask_precision: u8,
34     ask_amount: Uint128,
35     commission_rate: Decimal,
36     amp: u64,
37 ) -> StdResult<(Uint128, Uint128, Uint128)> {
38     // ask => offer
39
40     let greater_precision = offer_precision.max(ask_precision);
41     let offer_pool = adjust_precision(offer_pool, offer_precision,
 ↳  greater_precision)?;
42     let ask_pool = adjust_precision(ask_pool, ask_precision,
 ↳ greater_precision)?;
43     let ask_amount = adjust_precision(ask_amount, ask_precision,
 ↳ greater_precision)?;
44
45     let one_minus_commission = Decimal::one() - commission_rate;
46     let inv_one_minus_commission: Decimal = Decimal::one() /
 ↳ one_minus_commission;
47     let before_commission_deduction = ask_amount *
 ↳ inv_one_minus_commission;
48
49     let offer_amount = Uint128::new(
50         calc_offer_amount(
51             offer_pool.u128(),
52             ask_pool.u128(),
```

```
53              before_commission_deduction.u128(),
54              amp,
55          )
56          .unwrap(),
57      );
58  }
```

Risk Level:

Likelihood - 1
Impact - 1

Recommendation:

In “release” mode, Rust does not panic on overflows and overflown values just “wrap” without any explicit feedback to the user. It is recommended then to use vetted safe math libraries for arithmetic operations consistently throughout the smart contract system. Consider replacing the multiplication operator with Rust’s checked_div method, the multiplication operator with Rust’s checked_mul method, and so on.

Remediation Plan:

ACKNOWLEDGED: The A41 Team acknowledged this finding.

3.6 (HAL-06) INCOMPLETE DOCUMENTATION - INFORMATIONAL

Description:

The documentation provided is incomplete. For instance, the documentation included in the GitHub repository should include a contract diagram, instructions for users on how to interact with the contracts, list of the contracts with usage purpose and a walkthrough on how to deploy and test the smart contracts.

Risk Level:

Likelihood - 1
Impact - 1

Recommendation:

Consider updating the documentation in GitHub for clarifying data flow and work of the contract for the users and greater ease when contracts are deployed and tested. Have a Non-Developer or QA resource work through the process to make sure it addresses any gaps in the set-up steps due to technical assumptions.

Remediation Plan:

ACKNOWLEDGED: The A41 Team acknowledged this finding.

THANK YOU FOR CHOOSING

// HALBORN