// Security Assessment

02.27.2025 - 03.20.2025

BabyDEX

Tower

/ DRAFT /

HALBORN

BabyDEX - Tower

Prepared by: HALBORN

Last Updated 04/03/2025

Date of Engagement: February 27th, 2025 - March 20th, 2025

Summary

0% ⓘ OF ALL REPORTED FINDINGS HAVE BEEN ADDRESSED

ALL FINDINGS	CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL
5	0	0	0	0	5

TABLE OF CONTENTS

1. Introduction

`Tower` engaged Halborn to conduct a security assessment of the `BabyDEX`, beginning on February 27th, 2025 and ending on March 20th, 2025. This security assessment was scoped to the smart contracts in the BabyDEX GitHub repository. Commit hashes and further details can be found in the **Scope** section of this report.

2. Assessment Summary

The team at Halborn assigned a full-time security engineer to verify the security of the smart contracts. 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 assessment 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 should be addressed by the `Tower team`. The main ones were the following:

- `Remove all Astro token reward logic and vesting-related code to reduce storage access, optimize gas usage, and improve contract maintainability.`
- `Set info.sender as the default owner during instantiation and use propose_new_owner for ownership delegation to prevent misconfigurations and loss of control.`
- `Dynamically generate a unique LP token symbol based on asset pairs and use LP_TOKEN_PRECISION instead of hardcoded values to ensure clarity, flexibility, and maintainability.`
- `Remove redundant asset validation in instantiate and rely on the factory contract to ensure correct asset_infos, reducing unnecessary gas consumption.`
- `Initialize block_time_last with env.block.time instead of 0 to ensure accurate time_elapsed calculations and prevent potential distortions in price accumulation.`

3. Test Approach And 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 assessment. While manual testing is recommended to uncover flaws in logic, process, and implementation; automated testing techniques help enhance coverage of the code and can quickly identify items that do not follow the security best practices. The following phases and associated tools were used during the assessment:

- Research into architecture, purpose, and use of the platform.
- Manual code read and walk through.
- Manual Assessment of use and safety for the critical Rust variables and functions in scope to identify any arithmetic related vulnerability classes.
- Architecture related logical controls.
- Cross contract call controls.
- Scanning of Rust files for vulnerabilities(`cargo audit`)
- Review and improvement of integration tests.
- Verification of integration tests and implementation of new ones.

4. RISK METHODOLOGY

Every vulnerability and issue observed by Halborn is ranked based on **two sets** of **Metrics** and a **Severity Coefficient**. This system is inspired by the industry standard Common Vulnerability Scoring System.

The two **Metric sets** are: **Exploitability** and **Impact**. **Exploitability** captures the ease and technical means by which vulnerabilities can be exploited and **Impact** describes the consequences of a successful exploit.

The **Severity Coefficients** is designed to further refine the accuracy of the ranking with two factors: **Reversibility** and **Scope**. These capture the impact of the vulnerability on the environment as well as the number of users and smart contracts affected.

The final score is a value between 0-10 rounded up to 1 decimal place and 10 corresponding to the highest security risk. This provides an objective and accurate rating of the severity of security vulnerabilities in smart contracts.

The system is designed to assist in identifying and prioritizing vulnerabilities based on their level of risk to address the most critical issues in a timely manner.

4.1 EXPLOITABILITY

ATTACK ORIGIN (AO):

Captures whether the attack requires compromising a specific account.

ATTACK COST (AC):

Captures the cost of exploiting the vulnerability incurred by the attacker relative to sending a single transaction on the relevant blockchain. Includes but is not limited to financial and computational cost.

ATTACK COMPLEXITY (AX):

Describes the conditions beyond the attacker's control that must exist in order to exploit the vulnerability. Includes but is not limited to macro situation, available third-party liquidity and regulatory challenges.

METRICS:

EXPLOITABILITY METRIC (M_E)	METRIC VALUE	NUMERICAL VALUE
Attack Origin (AO)	Arbitrary (AO:A) Specific (AO:S)	1 0.2

EXPLOITABILITY METRIC (M_E)	METRIC VALUE	NUMERICAL VALUE
Attack Cost (AC)	Low (AC:L) Medium (AC:M) High (AC:H)	1 0.67 0.33
Attack Complexity (AX)	Low (AX:L) Medium (AX:M) High (AX:H)	1 0.67 0.33

Exploitability E is calculated using the following formula:

$$E = \prod m_e$$

4.2 IMPACT

CONFIDENTIALITY (C):

Measures the impact to the confidentiality of the information resources managed by the contract due to a successfully exploited vulnerability. Confidentiality refers to limiting access to authorized users only.

INTEGRITY (I):

Measures the impact to integrity of a successfully exploited vulnerability. Integrity refers to the trustworthiness and veracity of data stored and/or processed on-chain. Integrity impact directly affecting Deposit or Yield records is excluded.

AVAILABILITY (A):

Measures the impact to the availability of the impacted component resulting from a successfully exploited vulnerability. This metric refers to smart contract features and functionality, not state. Availability impact directly affecting Deposit or Yield is excluded.

DEPOSIT (D):

Measures the impact to the deposits made to the contract by either users or owners.

YIELD (Y):

Measures the impact to the yield generated by the contract for either users or owners.

METRICS:

IMPACT METRIC (M_I)	METRIC VALUE	NUMERICAL VALUE
Confidentiality (C)	None (I:N) Low (I:L) Medium (I:M) High (I:H) Critical (I:C)	0 0.25 0.5 0.75 1
Integrity (I)	None (I:N) Low (I:L) Medium (I:M) High (I:H) Critical (I:C)	0 0.25 0.5 0.75 1
Availability (A)	None (A:N) Low (A:L) Medium (A:M) High (A:H) Critical (A:C)	0 0.25 0.5 0.75 1
Deposit (D)	None (D:N) Low (D:L) Medium (D:M) High (D:H) Critical (D:C)	0 0.25 0.5 0.75 1
Yield (Y)	None (Y:N) Low (Y:L) Medium (Y:M) High (Y:H) Critical (Y:C)	0 0.25 0.5 0.75 1

Impact I is calculated using the following formula:

$$I = max(m_I) + \frac{\sum m_I - max(m_I)}{4}$$

4.3 SEVERITY COEFFICIENT

REVERSIBILITY (R):

Describes the share of the exploited vulnerability effects that can be reversed. For upgradeable contracts, assume the contract private key is available.

SCOPE (S):

Captures whether a vulnerability in one vulnerable contract impacts resources in other contracts.

METRICS:

SEVERITY COEFFICIENT (C)	COEFFICIENT VALUE	NUMERICAL VALUE
Reversibility (r)	None (R:N) Partial (R:P) Full (R:F)	1 0.5 0.25

SEVERITY COEFFICIENT (C)	COEFFICIENT VALUE	NUMERICAL VALUE
Scope (s)	Changed (S:C) Unchanged (S:U)	1.25 1

Severity Coefficient C is obtained by the following product:

$$C = rs$$

The Vulnerability Severity Score S is obtained by:

$$S = min(10, EIC * 10)$$

The score is rounded up to 1 decimal places.

SEVERITY	SCORE VALUE RANGE
Critical	9 - 10
High	7 - 8.9
Medium	4.5 - 6.9
Low	2 - 4.4
Informational	0 - 1.9

5. SCOPE

FILES AND REPOSITORY

(a) Repository: babydex

(b) Assessed Commit ID: 6cdceed

(c) Items in scope:

- contracts/periphery/native_coin_registry/src/error.rs
- contracts/periphery/native_coin_registry/src/lib.rs
- contracts/periphery/native_coin_registry/src/state.rs
- contracts/periphery/native_coin_registry/src/contract.rs
- contracts/pair_concentrated/src/error.rs
- contracts/pair_concentrated/src/lib.rs
- contracts/pair_concentrated/src/state.rs
- contracts/pair_concentrated/src/contract.rs
- contracts/pair_concentrated/src/queries.rs
- contracts/pair_concentrated/src/utils.rs
- contracts/tokenomics/incentives/src/execute.rs
- contracts/tokenomics/incentives/src/error.rs
- contracts/tokenomics/incentives/src/lib.rs
- contracts/tokenomics/incentives/src/query.rs
- contracts/tokenomics/incentives/src/migrate.rs
- contracts/tokenomics/incentives/src/state.rs
- contracts/tokenomics/incentives/src/reply.rs
- contracts/tokenomics/incentives/src/traits.rs
- contracts/tokenomics/incentives/src/utils.rs
- contracts/tokenomics/incentives/src/instantiate.rs
- contracts/factory/src/error.rs
- contracts/factory/src/lib.rs
- contracts/factory/src/state.rs
- contracts/factory/src/contract.rs
- contracts/factory/src/testing.rs
- contracts/factory/src/mock_querier.rs
- contracts/pair/src/error.rs
- contracts/pair/src/lib.rs
- contracts/pair/src/state.rs
- contracts/pair/src/contract.rs
- contracts/pair/src/testing.rs
- contracts/pair/src/mock_querier.rs
- contracts/router/src/error.rs
- contracts/router/src/lib.rs
- contracts/router/src/state.rs
- contracts/router/src/contract.rs

- contracts/router/src/operations.rs
- packages/astroport/src/vesting.rs
- packages/astroport/src/factory.rs
- packages/astroport/src/lib.rs
- packages/astroport/src/asset.rs
- packages/astroport/src/incentives.rs
- packages/astroport/src/pair.rs
- packages/astroport/src/router.rs
- packages/astroport/src/cosmwasm_ext.rs
- packages/astroport/src/pair_concentrated.rs
- packages/astroport/src/native_coin_registry.rs
- packages/astroport/src/querier.rs
- packages/astroport/src/common.rs
- packages/astroport_pcl_common/src/consts.rs
- packages/astroport_pcl_common/src/error.rs
- packages/astroport_pcl_common/src/lib.rs
- packages/astroport_pcl_common/src/math/mod.rs
- packages/astroport_pcl_common/src/math/math_f64.rs
- packages/astroport_pcl_common/src/math/math_decimal.rs
- packages/astroport_pcl_common/src/state.rs
- packages/astroport_pcl_common/src/utils.rs

Out-of-Scope: Third party dependencies and economic attacks.

Out-of-Scope: New features/implementations after the remediation commit IDs.

6. ASSESSMENT SUMMARY & FINDINGS OVERVIEW

CRITICAL	HIGH	MEDIUM	LOW
0	0	0	0

INFORMATIONAL
5

SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
HAL-01 - UNUSED CODE INCREASE GAS COSTS AND CONTRACT COMPLEXITY	INFORMATIONAL	PENDING
HAL-02 - UNSAFE OWNER ASSIGNMENT DURING INSTANTIATION	INFORMATIONAL	PENDING
HAL-03 - LP TOKEN INSTANTIATION USES HARDCODED SYMBOL AND PRECISION	INFORMATIONAL	PENDING
HAL-04 - REDUNDANT ASSET VALIDATION IN INSTANTIATE INCREASES GAS COSTS	INFORMATIONAL	PENDING
HAL-05 - INCORRECT INITIALIZATION OF BLOCK_TIME_LAST	INFORMATIONAL	PENDING

7. FINDINGS & TECH DETAILS

7.1 (HAL-01) UNUSED CODE INCREASE GAS COSTS AND CONTRACT COMPLEXITY

// INFORMATIONAL

Description

The **Incentives** contract still contains logic related to Astro token rewards and vesting, even though these functionalities are no longer used. This introduces unnecessary gas costs, storage usage, and execution overhead. The presence of vesting-related code increases contract complexity and makes maintenance more difficult.
Impact:

- **Higher gas costs:** Unused calculations and state storage increase transaction costs for LP stakers.
- **Code hygiene issues:** Maintaining deprecated functionality increases the risk of future integration errors and complicates audits.
- **Potential security risks:** If Astro-related functions remain partially accessible and wrong configured, they could be exploited to extract rewards improperly.

BVSS

AO:A/AC:L/AX:H/R:N/S:U/C:N/A:N/I:L/D:N/Y:N (0.8)

Recommendation

It is recommended to remove all Astro token reward logic that is no longer in use and eliminate vesting-related code and state variables to reduce unnecessary storage access. Refactor the contract to focus only on external reward mechanisms, optimizing gas usage and contract maintainability.

7.2 (HAL-02) UNSAFE OWNER ASSIGNMENT DURING INSTANTIATION

// INFORMATIONAL

Description

In the `instantiate` function of **Factory**, **NativeCoinRegistry** and **Incentives** contracts, ownership is assigned based on the `msg.owner` field instead of the sender of the transaction (`info.sender`). This allows the deployer to specify any address as the owner, potentially leading to:

- **Loss of control**: If an incorrect address is provided, the deployer might lose control over the contract immediately upon deployment.
- **Operational risks**: If the provided address is incorrect or unowned, the contract might be orphaned, meaning no one can manage its configuration or perform privileged actions.

Code Location

Code snippet of `instantiate` function from **Factory** contract:

```
40  pub fn instantiate(
41      deps: DepsMut,
42      _env: Env,
43      _info: MessageInfo,
44      msg: InstantiateMsg,
45  ) -> Result<Response, ContractError> {
46      set_contract_version(deps.storage, CONTRACT_NAME, CONTRACT_VERSION)?;
47
48      let mut config = Config {
49          owner: deps.api.addr_validate(&msg.owner)?,
50          token_code_id: msg.token_code_id,
51          fee_address: None,
52          incentives_address: None,
53          coin_registry_address: deps.api.addr_validate(&msg.coin_registry_address)?,
54      };
```

BVSS

AO:A/AC:L/AX:H/R:N/S:U/C:N/A:L/I:N/D:N/Y:N (0.8)

Recommendation

It is recommended to set the contract owner as `info.sender` by default during instantiation. If ownership delegation is necessary, a separate function (`propose_new_owner`) should be used to transfer ownership after deployment. This approach ensures that the deployer has initial control over the contract and can securely transfer ownership if needed.

7.3 (HAL-03) LP TOKEN INSTANTIATION USES HARDCODED SYMBOL AND PRECISION

// INFORMATIONAL

Description

The LP token instantiation in the **PairConcentrated** and **Pair** contracts could be improved due to the following issues:

- The LP token is instantiated with a hardcoded symbol `uLP`, which is the same as in the XYK pair, making it difficult to distinguish between different pool types (e.g., XYK vs. PCL).
- The decimal precision is hardcoded as 6 instead of using `LP_TOKEN_PRECISION`, reducing flexibility and maintainability.

Code Location

Code snippet of `instantiate` function from **PairConcentrated** contract:

```
137     let sub_msg = SubMsg::reply_on_success(
138         wasm_instantiate(
139             msg.token_code_id,
140             &cw20_base::msg::InstantiateMsg {
141                 name: format_lp_token_name(&msg.asset_infos, &deps.querier)?,
142                 symbol: "uLP".to_string(),
143                 decimals: 6,
144                 initial_balances: vec![],
145                 mint: Some(MinterResponse {
146                     minter: env.contract.address.to_string(),
147                     cap: None,
148                 }),
149                 marketing: None,
150             },
151             vec![],
152             "LP token".to_string(),
153         )?,
154         INSTANTIATE_TOKEN_REPLY_ID,
155     );
```

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:N/I:N/D:N/Y:N (0.0)

Recommendation

It is recommended to dynamically generate a unique LP token symbol based on the asset pair to clearly distinguish different LP tokens. Example: "LP-USDC-ETH" instead of "uLP".
Also use LP_TOKEN_PRECISION instead of hardcoding 6, ensuring consistency and easier future updates.

7.4 (HAL-04) REDUNDANT ASSET VALIDATION IN INSTANTIATE INCREASES GAS COSTS

// INFORMATIONAL

Description

The `instantiate` function in both the **PairConcentrated** and **Pair** contracts unnecessarily revalidates `asset_infos`, even though this validation is already performed at the factory level. Both contracts redundantly check the assets using the `check_asset_infos` function, despite the factory ensuring this before passing the data to the pair contract. This duplicate validation results in unnecessary gas consumption without adding any additional security.

Code Location

Code snippet of `execute_create_pair` function from **Factory** contract:

```
263 pub fn execute_create_pair(
264     deps: DepsMut,
265     info: MessageInfo,
266     env: Env,
267     pair_type: PairType,
268     asset_infos: Vec<AssetInfo>,
269     init_params: Option<Binary>,
270 ) -> Result<Response, ContractError> {
271     check_asset_infos(deps.api, &asset_infos)?;
```

Code snippet of `instantiate` function from **PairConcentrated** contract:

```
53 pub fn instantiate(
54     mut deps: DepsMut,
55     env: Env,
56     _info: MessageInfo,
57     msg: InstantiateMsg,
58 ) -> Result<Response, ContractError> {
59     if msg.asset_infos.len() != 2 {
60         return Err(StdError::generic_err("asset_infos must contain exactly two elements").into());
61     }
62
63     check_asset_infos(deps.api, &msg.asset_infos)?;
```

Code snippet of `instantiate` function from **Pair** contract:

```
43 pub fn instantiate(
44     deps: DepsMut,
45     env: Env,
46     _info: MessageInfo,
47     msg: InstantiateMsg,
48 ) -> Result<Response, ContractError> {
49     if msg.asset_infos.len() != 2 {
50         return Err(StdError::generic_err("asset_infos must contain exactly two elements").into());
51     }
52
53     msg.asset_infos[0].check(deps.api)?;
54     msg.asset_infos[1].check(deps.api)?;
```

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:N/I:N/D:N/Y:N (0.0)

Recommendation

It is recommended to remove redundant validation from pair `instantiate` and rely on the factory contract to ensure correct `asset_infos`, reducing gas consumption and improving execution efficiency.

7.5 (HAL-05) INCORRECT INITIALIZATION OF BLOCK_TIME_LAST

// INFORMATIONAL

Description

The `instantiate` function of **Pair** contract initializes `block_time_last` with 0 instead of using the current block time (env.block.time). This results in a distorted `time_elapsed` calculation in the `accumulate_prices` function when it is first executed.
During its first execution, `accumulate_prices` will compute `time_elapsed` using an incorrect reference point (0 instead of the actual instantiation timestamp). However, this does not introduce a significant issue, as `accumulate_prices` only updates values if there is liquidity in the pool. Since the pool will have no liquidity on the first execution, the `time_elapsed` value is not used, preventing any immediate impact on price accumulation.

Code Location

Code snippet of `instantiate` function from **Pair** contract:

```
43  pub fn instantiate(
44      deps: DepsMut,
45      env: Env,
46      _info: MessageInfo,
47      msg: InstantiateMsg,
48  ) -> Result<Response, ContractError> {
49      if msg.asset_infos.len() != 2 {
50          return Err(StdError::generic_err("asset_infos must contain exactly two elements").into());
51      }
52
53      msg.asset_infos[0].check(deps.api)?;
54      msg.asset_infos[1].check(deps.api)?;
55
56      if msg.asset_infos[0] == msg.asset_infos[1] {
57          return Err(ContractError::DoublingAssets {});
58      }
59
60      set_contract_version(deps.storage, CONTRACT_NAME, CONTRACT_VERSION)?;
61
62      let config = Config {
63          pair_info: PairInfo {
64              contract_addr: env.contract.address.clone(),
65              liquidity_token: "".to_owned(),
66              asset_infos: msg.asset_infos.clone(),
67              pair_type: msg.pair_type,
68          },
69          factory_addr: deps.api.addr_validate(msg.factory_addr.as_str())?,
70          block_time_last: 0,
71          price0_cumulative_last: Uint128::zero(),
72          price1_cumulative_last: Uint128::zero(),
73          fee_share: None,
74      };
```

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:N/I:N/D:N/Y:N (0.0)

Recommendation

It is recommended to set `block_time_last` to env.block.time instead of 0 during instantiation. This ensures that `time_elapsed` is always calculated correctly from the first execution onward, avoiding unnecessary distortions.

Halborn strongly recommends conducting a follow-up assessment of the project either within six months or immediately following any material changes to the codebase, whichever comes first. This approach is crucial for maintaining the project's integrity and addressing potential vulnerabilities introduced by code modifications.