SECURITY AUDIT REPORT

for

Persistence Dexter

Prepared By: Xiaomi Huang

PeckShield

February 13, 2022

Document Properties

Client	Persistence
Title	Security Audit Report
Target	Dexter
Version	1.0
Author	Xiaotao Wu
Auditors	Xiaotao Wu, Xuxian Jiang
Reviewed by	Xiaomi Huang
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	February 13, 2022	Xiaotao Wu	Final Release
1.0-rc2	October 27, 2022	Xiaotao Wu	Release Candidate #2
1.0-rc1	September 30, 2022	Xiaotao Wu	Release Candidate #1

Contact

For more information about this document and its contents, please contact PeckShield Inc.

Name	Xiaomi Huang
Phone	+86 183 5897 7782
Email	contact@peckshield.com

Contents

1 Introduction

Given the opportunity to review the design document and related smart contract source code of the Dexter protocol, we outline in the report our systematic approach to evaluate potential security issues in the smart contract implementation, expose possible semantic inconsistencies between smart contract code and design document, and provide additional suggestions or recommendations for improvement. Our results show that the given version of smart contracts can be further improved due to the presence of several issues related to either security or performance. This document outlines our audit results.

1.1 About Dexter

Dexter is a DEX which is implemented as a generalized state transition executor where the transition's math computes are queried from the respective pool contracts, enabling a decentralized, non-custodial aggregated liquidity and exchange rate discovery among different tokens on Persistence. The pool types that Dexter will be supporting at launch includes XYK Pool, Stableswap Pool, Stable5Pool, and Weighted Pool. The basic information of audited contracts is as follows:

Table 1.1: Basic Information of Dexter

Item	Description
Name	Persistence
Website	https://persistence.one/
Type	Cosmos
Language	Rust
Audit Method	Whitebox
Latest Audit Report	February 13, 2022

In the following, we show the Git repository of reviewed files and the commit hash value used in this audit.

- https://github.com/dexter-zone/dexter_core.git (40069a4)

And here is the commit ID after all fixes for the issues found in the audit have been checked in:

- https://github.com/dexter-zone/dexter_core.git (a3406cb)

1.2 About PeckShield

PeckShield Inc. [12] is a leading blockchain security company with the goal of elevating the security, privacy, and usability of current blockchain ecosystems by offering top-notch, industry-leading services and products (including the service of smart contract auditing). We are reachable at Telegram (https://t.me/peckshield), Twitter (http://twitter.com/peckshield), or Email (contact@peckshield.com).

Table 1.2: Vulnerability Severity Classification

Impact \ Likelihood	*High*	*Medium*	*Low*
High	Critical	High	Medium
Medium	High	Medium	Low
Low	Medium	Low	Low

1.3 Methodology

To standardize the evaluation, we define the following terminology based on OWASP Risk Rating Methodology [11]:

- Likelihood represents how likely a particular vulnerability is to be uncovered and exploited in the wild;
- Impact measures the technical loss and business damage of a successful attack;
- Severity demonstrates the overall criticality of the risk.

Likelihood and impact are categorized into three ratings: *H*, *M* and *L*, i.e., *high*, *medium* and *low* respectively. Severity is determined by likelihood and impact, and can be accordingly classified into four categories, i.e., *Critical*, *High*, *Medium*, *Low* shown in Table 1.2.

To evaluate the risk, we go through a list of check items and each would be labeled with a severity category. For one check item, if our tool or analysis does not identify any issue, the

Table 1.3: The Full List of Check Items

Category	Check Item
Basic Coding Bugs	Constructor Mismatch
	Ownership Takeover
	Redundant Fallback Function
	Overflows & Underflows
	Reentrancy
	Money-Giving Bug
	Blackhole
	Unauthorized Self-Destruct
	Revert DoS
	Unchecked External Call
	Gasless Send
	Send Instead Of Transfer
	Costly Loop
	(Unsafe) Use Of Untrusted Libraries
	(Unsafe) Use Of Predictable Variables
	Transaction Ordering Dependence
	Deprecated Uses
Semantic Consistency Checks	Semantic Consistency Checks
Advanced DeFi Scrutiny	Business Logics Review
	Functionality Checks
	Authentication Management
	Access Control & Authorization
	Oracle Security
	Digital Asset Escrow
	Kill-Switch Mechanism
	Operation Trails & Event Generation
	ERC20 Idiosyncrasies Handling
	Frontend-Contract Integration
	Deployment Consistency
	Holistic Risk Management
Additional Recommendations	Avoiding Use of Variadic Byte Array
	Using Fixed Compiler Version
	Making Visibility Level Explicit
	Making Type Inference Explicit
	Adhering To Function Declaration Strictly
	Following Other Best Practices

contract is considered safe regarding the check item. For any discovered issue, we might further deploy contracts on our private testnet and run tests to confirm the findings. If necessary, we would additionally build a PoC to demonstrate the possibility of exploitation. The concrete list of check items is shown in Table 1.3.

In particular, we perform the audit according to the following procedure:

- Basic Coding Bugs: We first statically analyze given smart contracts with our proprietary static code analyzer for known coding bugs, and then manually verify (reject or confirm) all the issues found by our tool.
- Semantic Consistency Checks: We then manually check the logic of implemented smart contracts and compare with the description in the white paper.
- Advanced DeFi Scrutiny: We further review business logics, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.
- Additional Recommendations: We also provide additional suggestions regarding the coding and development of smart contracts from the perspective of proven programming practices.

To better describe each issue we identified, we categorize the findings with Common Weakness Enumeration (CWE-699) [10], which is a community-developed list of software weakness types to better delineate and organize weaknesses around concepts frequently encountered in software development. Though some categories used in CWE-699 may not be relevant in smart contracts, we use the CWE categories in Table 1.4 to classify our findings. Moreover, in case there is an issue that may affect an active protocol that has been deployed, the public version of this report may omit such issue, but will be amended with full details right after the affected protocol is upgraded with respective fixes.

1.4 Disclaimer

Note that this security audit is not designed to replace functional tests required before any software release, and does not give any warranties on finding all possible security issues of the given smart contract(s) or blockchain software, i.e., the evaluation result does not guarantee the nonexistence of any further findings of security issues. As one audit-based assessment cannot be considered comprehensive, we always recommend proceeding with several independent audits and a public bug bounty program to ensure the security of smart contract(s). Last but not least, this security audit should not be used as investment advice.

Table 1.4: Common Weakness Enumeration (CWE) Classifications Used in This Audit

Category	Summary
Configuration	Weaknesses in this category are typically introduced during the configuration of the software.
Data Processing Issues	Weaknesses in this category are typically found in functionality that processes data.
Numeric Errors	Weaknesses in this category are related to improper calculation or conversion of numbers.
Security Features	Weaknesses in this category are concerned with topics like authentication, access control, confidentiality, cryptography, and privilege management. (Software security is not security software.)
Time and State	Weaknesses in this category are related to the improper management of time and state in an environment that supports simultaneous or near-simultaneous computation by multiple systems, processes, or threads.
Error Conditions, Return Values, Status Codes	Weaknesses in this category include weaknesses that occur if a function does not generate the correct return/status code, or if the application does not handle all possible return/status codes that could be generated by a function.
Resource Management	Weaknesses in this category are related to improper management of system resources.
Behavioral Issues	Weaknesses in this category are related to unexpected behaviors from code that an application uses.
Business Logics	Weaknesses in this category identify some of the underlying problems that commonly allow attackers to manipulate the business logic of an application. Errors in business logic can be devastating to an entire application.
Initialization and Cleanup	Weaknesses in this category occur in behaviors that are used for initialization and breakdown.
Arguments and Parameters	Weaknesses in this category are related to improper use of arguments or parameters within function calls.
Expression Issues	Weaknesses in this category are related to incorrectly written expressions within code.
Coding Practices	Weaknesses in this category are related to coding practices that are deemed unsafe and increase the chances that an exploitable vulnerability will be present in the application. They may not directly introduce a vulnerability, but indicate the product has not been carefully developed or maintained.

2 | Findings

2.1 Summary

Here is a summary of our findings after analyzing the implementation of the `Dexter` protocol. During the first phase of our audit, we study the smart contract source code and run our in-house static code analyzer through the codebase. The purpose here is to statically identify known coding bugs, and then manually verify (reject or confirm) issues reported by our tool. We further manually review business logics, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.

Severity	# of Findings	
Critical	0	
High	1	■
Medium	2	■ ■
Low	5	■ ■ ■ ■ ■
Informational	0	
Total	8	

We have so far identified a list of potential issues: some of them involve subtle corner cases that might not be previously thought of, while others refer to unusual interactions among multiple contracts. For each uncovered issue, we have therefore developed test cases for reasoning, reproduction, and/or verification. After further analysis and internal discussion, we determined a few issues of varying severities need to be brought up and paid more attention to, which are categorized in the above table. More information can be found in the next subsection, and the detailed discussions of each of them are in Section 3.

2.2 Key Findings

Overall, these smart contracts are well-designed and engineered, though the implementation can be improved by resolving the identified issues (shown in Table 2.1), including 1 high-severity vulnerability, 2 medium-severity vulnerabilities, and 5 low-severity vulnerabilities.

Table 2.1: Key Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Low	Improved Implementation Logic in generator::create_pool()	Business Logic	Resolved
PVE-002	Medium	Incorrect Accumulate Prices Updates in Dexter	Business Logic	Resolved
PVE-003	Low	Improved Sanity Checks Of System/Function Parameters	Coding Practices	Resolved
PVE-004	Low	Improved Precision in xyk_pool/stable_pool::accumulate_prices()	Numeric Errors	Resolved
PVE-005	Low	Revisited Logic in weighted_pool::query_on_join_pool()	Business Logic	Resolved
PVE-006	High	Incorrect Implementation Logic in stable_5pool	Business Logic	Resolved
PVE-007	Medium	Trust Issue of Admin Keys	Security Features	Confirmed
PVE-008	Low	Revisited Logic in in router::query_simulate_multihop()	Business Logic	Resolved

Beside the identified issues, we emphasize that for any user-facing applications and services, it is always important to develop necessary risk-control mechanisms and make contingency plans, which may need to be exercised before the mainnet deployment. The risk-control mechanisms should kick in at the very moment when the contracts are being deployed on mainnet. Please refer to Section 3 for details.

3 Detailed Results

3.1 Improved Implementation Logic in generator::create_pool()

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: generator
- Category: Business Logic [8]
- CWE subcategory: CWE-841 [5]

Description

Within the Dexter protocol, the generator contract provides a public create_pool() function for the privileged owner account to add a new pool to the list of active pools. While examining the routine, we notice the current implementation logic can be improved.

To elaborate, we show below its code snippet. It comes to our attention that a pool can be created without validating whether it is allowed by the vault. In other words, a pool can only be created when the is_generator_disabled state associated with this pool type is set to false in the vault contract.

```
    pub fn create_pool(
        deps: DepsMut,
        env: &Env,
        lp_token: &Addr,
        cfg: &Config,
    ) -> Result<PoolInfo, ContractError> {
        // Create Pool
        POOL_INFO.save(
            deps.storage,
            lp_token,
            &PoolInfo {
                last_reward_block: cfg.start_block.max(Uint64::from(env.block.height)),
                reward_proxy: None,
                accumulated_proxy_rewards_per_share: Default::default(),
                proxy_reward_balance_before_update: Uint128::zero(),
                orphan_proxy_rewards: Default::default(),
```

```
                accumulated_rewards_per_share: Decimal::zero(),
            },
        )?;

        Ok(POOL_INFO.load(deps.storage, lp_token)?)
    }
```

Listing 3.1: `dexter_generator/generator/src/contract.rs::create_pool()`

Recommendation Only allow to create a new pool when the `is_generator_disabled` state associated with this pool type is set to false in the `vault` contract.

Status The issue has been fixed by this commit: `48c27b8`.

3.2 Incorrect Accumulate Prices Updates in Dexter

- ID: PVE-002
- Severity: Medium
- Likelihood: High
- Impact: Low
- Target: `Multiple contracts`
- Category: Business Logic [8]
- CWE subcategory: CWE-837 [4]

Description

When building smart contracts that integrate with `DeFi` protocols, developers will inevitably run into the price oracle problem. To build highly decentralized and manipulation-resistant on-chain price oracles, the `Dexter` protocol introduces time-weighted average prices (`TWAPs`), which is originally designed by `Uniswap Protocol`. The `TWAP` is constructed by reading the cumulative price from a token pair at the beginning and at the end of the desired interval. The difference in this cumulative price can then be divided by the length of the interval to create a `TWAP` for that period. While examining the price cumulative mechanism in `Dexter`, we notice the current implementation is not correct.

To elaborate, we use the `xyk_pool` contract as an example and show below the related code snippet. Specifically, it should use the old asset amounts to calculate the accumulated prices, instead of current newly updated asset amounts (line 211).

```
189     pub fn execute_update_pool_liquidity(
190         deps: DepsMut,
191         env: Env,
192         info: MessageInfo,
193         assets: Vec<Asset>,
194     ) -> Result<Response, ContractError> {
195         // Get config and twap info
196         let mut config: Config = CONFIG.load(deps.storage)?;
197         let mut twap: Twap = TWAPINFO.load(deps.storage)?;
```

```

        // Acess Check :: Only Vault can execute this function
        if info.sender != config.vault_addr {
            return Err(ContractError::Unauthorized {});
        }

        // Update state
        config.assets = assets;
        config.block_time_last = env.block.time.seconds();
        CONFIG.save(deps.storage, &config)?;

        // Accumulate prices for the assets in the pool
        if let Some((price0_cumulative_new, price1_cumulative_new, block_time)) =
            accumulate_prices(env, &twap, config.assets[0].amount, config.assets[1].
                amount)?
        {
            twap.price0_cumulative_last = price0_cumulative_new;
            twap.price1_cumulative_last = price1_cumulative_new;
            twap.block_time_last = block_time;
            TWAPINFO.save(deps.storage, &twap)?;
        }

        let event = Event::new("dexter-pool::update-liquidity")
            .add_attribute("pool_id", config.pool_id.to_string())
            .add_attribute(
                config.assets[0].info.as_string(),
                twap.price0_cumulative_last.to_string(),
            )
            .add_attribute(
                config.assets[1].info.as_string(),
                twap.price1_cumulative_last.to_string(),
            )
            .add_attribute("block_time_last", twap.block_time_last.to_string());

        Ok(Response::new().add_event(event))
    }
```

Listing 3.2: xyk_pool/src/contract.rs::execute_update_pool_liquidity()

Note similar issue also exists in the Stableswap Pool, Stable5Pool, and Weighted Pool.

Recommendation Use the correct asset amounts to calculate the accumulated prices.

Status The issue has been fixed by this commit: 8190b8a.

3.3 Improved Sanity Checks Of System/Function Parameters

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `Multiple contracts`
- Category: Coding Practices [7]
- CWE subcategory: CWE-1126 [1]

Description

In the `Dexter` protocol, new pool contracts can be created by executing the `execute_create_pool_instance()` function defined in the `vault` contract. While reviewing the `instantiate()` function of the `xyk_pool/stable_pool` contracts, we notice that it can benefit from additional sanity checks.

To elaborate, we use the `xyk_pool` contract as an example and show below the related code snippet. Specifically, there is a lack of length verification for the input assets. For `XYK Pool` and `Stableswap Pool` , the `msg.asset_infos.len()` should always be equal to 2.

```
#[cfg_attr(not(feature = "library"), entry_point)]
pub fn instantiate(
  deps: DepsMut,
  env: Env,
  _info: MessageInfo,
  msg: InstantiateMsg,
) -> Result<Response, ContractError> {
  set_contract_version(deps.storage, CONTRACT_NAME, CONTRACT_VERSION)?;

  // Validate token info : token name and symbol
  msg.validate()?;

  // Create [`Asset`] from [`AssetInfo`]
  let assets = msg
    .asset_infos
    .iter()
    .map(a Asset {
      info: a.clone(),
      amount: Uint128::zero(),
    })
    .collect();

  // Create Config
  let config = Config {
    pool_id: msg.pool_id,
    lp_token_addr: None,
    vault_addr: msg.vault_addr.clone(),
    assets,
    pool_type: msg.pool_type,
    fee_info: msg.fee_info,
    block_time_last: env.block.time.seconds(),
```

```
  };

  ...
}
```

Listing 3.3: `xyk_pool/src/contract.rs::instantiate()`

Note the `execute_add_to_registery()` routine of the `vault` contract can also benefit from additional sanity checks. Specifically, the input argument `new_pool_config.code_id` could not be 0.

```
pub fn execute_add_to_registery(
  deps: DepsMut,
  _env: Env,
  info: MessageInfo,
  new_pool_config: PoolConfig,
) -> Result<Response, ContractError> {
  let config: Config = CONFIG.load(deps.storage)?;

  // permission check : Only owner can execute it
  if info.sender != config.owner {
    return Err(ContractError::Unauthorized {});
  }

  // Check :: If pool type is already registered
  let mut pool_config = REGISTERY
    .load(deps.storage, new_pool_config.pool_type.to_string())
    .unwrap_or_default();
  if pool_config.code_id != 0u64 {
    return Err(ContractError::PoolTypeAlreadyExists {});
  }

  // Set pool config
  pool_config = new_pool_config;

  // validate fee bps limits
  if !pool_config.fee_info.valid_fee_info() {
    return Err(ContractError::InvalidFeeInfo {});
  }

  // Save pool config
  REGISTERY.save(
    deps.storage,
    pool_config.pool_type.to_string(),
    &pool_config,
  )?;

  // Emit Event
  let event = Event::new("dexter-vault::add_new_pool")
    .add_attribute("pool_type", pool_config.pool_type.to_string())
    .add_attribute("code_id", pool_config.code_id.to_string());
  Ok(Response::new().add_event(event))
```

```
}
```

Listing 3.4: `vault/src/contract.rs::execute_add_to_registery()`

Recommendation Add necessary sanity checks for the above mentioned functions.

Status The issue has been fixed by this commit: `1f926dd`.

3.4 Improved Precision in xyk_pool/stable_pool::accumulate_prices()

- ID: PVE-004
- Severity: low
- Likelihood: low
- Impact: low
- Target: `xyk_pool/stable_pool`
- Category: Numeric Errors [9]
- CWE subcategory: CWE-197 [2]

Description

As mentioned in Section 3.2, to build highly decentralized and manipulation-resistant on-chain price oracles, the `Dexter` protocol introduces time-weighted average prices (`TWAPs`). In particular, the `accumulate_prices()` function is used to accumulate prices for the assets in the pool. While reviewing the implementation of the `xyk_pool/stable_pool`, we notice that the calculation could be further improved to provide a more accurate result.

To elaborate, we use the `xyk_pool` contract as an example and show below the full implementation of the `accumulate_prices()` routine. Specifically, the newly accumulated prices are computed as `time_elapsed.checked_mul(price_precision)?.multiply_ratio(y, x)` (lines 812-814) and `time_elapsed.checked_mul(price_precision)?.multiply_ratio(x, y)` (lines 817-819). With the assumption that the decimal of asset `x` is 18 and the decimal of asset `y` is 6, the computed result for `time_elapsed.checked_mul(price_precision)?.multiply_ratio(y, x)` (lines 812-814) might be equal to 0.

```
791     pub fn accumulate_prices(
792     env: Env,
793     twap: &Twap,
794     x: Uint128,
795     y: Uint128,
796 ) -> StdResult<Option<(Uint128, Uint128, u64)>> {
797     let block_time = env.block.time.seconds();
798     if block_time <= twap.block_time_last {
799         return Ok(None);
800     }
801
```

```
    // we have to shift block_time when any price is zero to not fill an accumulator
        with a new price to that period

    let time_elapsed = Uint128::from(block_time - twap.block_time_last);

    let mut pcl0 = twap.price0_cumulative_last;
    let mut pcl1 = twap.price1_cumulative_last;

    if !x.is_zero() && !y.is_zero() {
        let price_precision = Uint128::from(10u128.pow(TWAP_PRECISION.into()));
        pcl0 = twap.price0_cumulative_last.wrapping_add(
            time_elapsed
                .checked_mul(price_precision)?
                .multiply_ratio(y, x),
        );
        pcl1 = twap.price1_cumulative_last.wrapping_add(
            time_elapsed
                .checked_mul(price_precision)?
                .multiply_ratio(x, y),
        );
    };

    Ok(Some((pcl0, pcl1, block_time)))
}
```

Listing 3.5: `xyk_pool/src/contract.rs::accumulate_prices()`

Recommendation Adjust the assets in the pool to the greatest precision before calculating the accumulated prices.

Status This issue has been addressed as the `Dexter` team has removed the `xyk_pool` and `stable_pool` from the code repo.

3.5 Revisited Logic in weighted_pool::query_on_join_pool()

- ID: PVE-005
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `weighted_pool`
- Category: Business Logic [8]
- CWE subcategory: CWE-841 [5]

Description

In the `Dexter` protocol, a user can join a pool by executing the `execute_join_pool()` function defined in the `vault` contract. The number of assets or `LP` tokens to be minted to the user is decided by the pool contract's math computations. The `vault` contract is responsible for the the transfer of assets and minting of `LP` tokens to the user.

In the following, we show the code snippet of the math computations for joining the `Weighted Pool`, i.e., the `query_on_join_pool()` routine defined in the `weighted_pool` contract. While examining the current implementation logic, we notice there is a lack of validating whether the `total_share` is equal to 0 if a single asset is provided to join the pool (line 477). If the `total_share` is equal to 0, the math computations for a single asset join is meaningless.

```
423  pub fn query_on_join_pool(
424    deps: Deps,
425    _env: Env,
426    assets_in: Option<Vec<Asset>>,
427    _mint_amount: Option<Uint128>,
428    _slippage_tolerance: Option<Decimal>,
429  ) -> StdResult<AfterJoinResponse> {
430    // If the user has not provided any assets to be provided, then return a `Failure`
         response
431    if assets_in.is_none() {
432      return Ok(return_join_failure("No assets provided".to_string()));
433    }
434    // Sort the assets in the order of the assets in the config
435    let mut act_assets_in = assets_in.unwrap();
436    act_assets_in.sort_by(|a, b| {
437      a.info
438        .to_string()
439        .to_lowercase()
440        .cmp(&b.info.to_string().to_lowercase())
441    });
442
443    // 1) Get pool current liquidity + and token weights
444    // 2) If single token provided, do single asset join and exit.
445    // 3) If multi-asset join, first do as much of a join as we can with no swaps.
446    // 4) Update pool shares / liquidity / remaining tokens to join accordingly
447    // 5) For every remaining token to LP, do a single asset join, and update pool
         shares / liquidity.
448    //
449    // Note that all single asset joins do incur swap fee.
450    //
451    // Since CalcJoinPoolShares is non-mutative, the steps for updating pool shares /
         liquidity are
452    // more complex / don't just alter the state.
453
454    // We should simplify this logic further in the future, using balancer multi-join
         equations.
455
456    // 1) get all 'pool assets' (aka current pool liquidity + balancer weight)
457
458    let config: Config = CONFIG.load(deps.storage)?;
459    // let math_config: MathConfig = MATHCONFIG.load(deps.storage)?;
460    // Total share of LP tokens minted by the pool
```

```
    let total_share = query_supply(&deps.querier, config.lp_token_addr.clone().unwrap().clone())?;

    //  1) Get pool current liquidity + and token weights : Convert assets to WeightedAssets
    let mut pool_assets_weighted: Vec<WeightedAsset> = config
      .assets
      .iter()
      .map(|asset| {
        let weight = get_weight(deps.storage, &asset.info)?;
        Ok(WeightedAsset {
          asset: asset.clone(),
          weight,
        })
      })
      .collect::<StdResult<Vec<WeightedAsset>>>()?;

    // 2) If single token provided, do single asset join and exit.
    if act_assets_in.len() == 1 {
      let in_asset = act_assets_in[0].to_owned();
      let weighted_in_asset = pool_assets_weighted
        .iter()
        .find(|asset| asset.asset.info.equal(&in_asset.info))
        .unwrap();
      let num_shares: Uint128 = calc_single_asset_join(
        deps,
        &in_asset,
        config.fee_info.total_fee_bps,
        weighted_in_asset,
        total_share,
      )?;
      // Add assets which are omitted with 0 deposit
      pool_assets_weighted.iter().for_each(|pool_asset| {
        if !act_assets_in
          .iter()
          .any(|asset| asset.info.eq(&pool_asset.asset.info))
        {
          act_assets_in.push(Asset {
            info: pool_asset.asset.info.clone(),
            amount: Uint128::new(0),
          });
        }
      });
      // Return the response
      if !num_shares.is_zero() {
        return Ok(AfterJoinResponse {
          provided_assets: act_assets_in,
          new_shares: num_shares,
          response: dexter::pool::ResponseType::Success {},
          fee: None,
        });
      }
```

```
    }

    ...
  }
```

Listing 3.6: `weighted_pool/src/contract.rs::query_on_join_pool()`

Recommendation Validate the `total_share` is not equal to 0 if a single asset is provided to join a `Weighted Pool`.

Status The issue has been fixed by this commit: `b9553b3`.

3.6 Incorrect Implementation Logic in stable_5pool

- ID: PVE-006
- Severity: High
- Likelihood: High
- Impact: High
- Target: `stable_5pool`
- Category: Business Logic [8]
- CWE subcategory: CWE-841 [5]

Description

The `query_on_join_pool()` function in the `stable_5pool` contract implements the math computations for users to join the `Stable5Pool`. The numbers of tokens to be transferred from a user to the `vault` and the new `LP` shares to be minted for the user are decided by the result of math computations. Our analysis with this routine shows the current precision-adjusting logic for the input assets is not correct.

To elaborate, we show below the related code snippet. Specifically, the action for adjusting the assets amount to the greatest precision in lines 455-461 is not needed and can be removed safely. Since the assets amount will be converted to decimal types in lines 476-486, this redundant precision adjustment will lead to incorrect math computations. Furthermore, the `amp` parameter should be calculated as `compute_current_amp(&math_config, &env)?.unwrap_or_else(|| 0u64.into())`, instead of current `compute_current_amp(&math_config, &env)?.checked_mul(n_coins.into()).unwrap_or_else(|| 0u64.into())` (lines 466-468).

```
436   pub fn query_on_join_pool(
437     deps: Deps,
438     env: Env,
439     assets_in: Option<Vec<Asset>>,
440     _mint_amount: Option<Uint128>,
441     _slippage_tolerance: Option<Decimal>,
442   ) -> StdResult<AfterJoinResponse> {
443     ...
```

```
    // Adjust for precision
    for (deposit, pool) in assets_collection.iter_mut() {
      // We cannot put a zero amount into an empty pool.
      if deposit.amount.is_zero() && pool.is_zero() {
        return Ok(return_join_failure(
          "Cannot deposit zero into an empty pool".to_string(),
        ));
      }

      // Adjusting to the greatest precision
      let coin_precision = get_precision(deps.storage, &deposit.info)?;
      deposit.amount = adjust_precision(
        deposit.amount,
        coin_precision,
        math_config.greatest_precision,
      )?;
      *pool = adjust_precision(*pool, coin_precision, math_config.greatest_precision)?;
    }

    // Compute amp parameter
    let n_coins = config.assets.len() as u8;
    let amp = compute_current_amp(&math_config, &env)?
      .checked_mul(n_coins.into())
      .unwrap_or_else(|| 0u64.into());

    // If AMP value is invalid, then return a `Failure` response
    if amp == 0u64 {
      return Ok(return_join_failure("Invalid amp value".to_string()));
    }

    // Convert to Decimal types
    let assets_collection = assets_collection
      .iter()
      .cloned()
      .map(|(asset, pool)| {
        let coin_precision = get_precision(deps.storage, &asset.info)?;
        Ok((
          asset.to_decimal_asset(coin_precision)?,
          Decimal256::with_precision(pool, coin_precision)?,
        ))
      })
      .collect::<StdResult<Vec<(DecimalAsset, Decimal256)>>>()?;

    ...
  }
```

Listing 3.7: `stable_5pool/src/contract.rs::query_on_join_pool()`

Note the incorrect implementation logic issue also exists in the `imbalanced_withdraw()` function of the same contract. Specifically, the `amp` parameter calculation logic is not correct (lines 1048-1050) and the precision adjust logic for the `burn_amount` should be removed (lines 1063-1067).

```
fn imbalanced_withdraw(
  deps: Deps,
  env: &Env,
  config: &Config,
  math_config: &MathConfig,
  provided_amount: Uint128,
  assets: &[Asset],
) -> Result<Uint128, ContractError> {
  ...

  let n_coins = config.assets.len() as u8;

  let amp = compute_current_amp(math_config, env)?
    .checked_mul(n_coins.into())
    .unwrap();

  ...

  let burn_amount = total_share
    .checked_multiply_ratio(
      init_d
        .to_uint128_with_precision(18u8)?
        .checked_sub(after_fee_d.to_uint128_with_precision(18u8)?)?,
      init_d.to_uint128_with_precision(18u8)?,
    )?
    .checked_add(Uint128::from(1u8))?; // In case of rounding errors - make it unfavorable for the "attacker"

  let burn_amount = adjust_precision(
    burn_amount,
    math_config.greatest_precision,
    LP_TOKEN_PRECISION,
  )?;
  ...
}
```

Listing 3.8: `stable_5pool/src/contract.rs::query_on_join_pool()`

Recommendation Remove the redundant codes from the above mentioned functions.

Status The issue has been fixed by this commit: `f991a9b`.

3.7 Trust Issue of Admin Keys

- ID: PVE-007
- Severity: Medium
- Likelihood: Low
- Impact: High
- Target: `Multiple contracts`
- Category: Security Features [6]
- CWE subcategory: CWE-287 [3]

Description

In `Dexter` protocol, there is a privileged account, i.e., `owner`. This account plays a critical role in governing and regulating the system-wide operations (e.g., update general settings for `vault`, update `DEX` pool configuration, create `DEX` pool instance, update stable pool's math configuration, update general settings for `generator`, create rewarding pool, set proxy for rewarding pool, deactivate rewarding pool,etc.). Our analysis shows that this privileged account needs to be scrutinized. In the following, we use the `vault/src/contract.rs` contract as an example and show the representative functions potentially affected by the privileges of the `owner` account.

```
    pub fn execute_update_config(
        deps: DepsMut,
        _env: Env,
        info: MessageInfo,
        lp_token_code_id: Option<u64>,
        fee_collector: Option<String>,
        generator_address: Option<String>,
    ) -> Result<Response, ContractError> {
        let mut config: Config = CONFIG.load(deps.storage)?;

        // permission check
        if info.sender.clone() != config.owner {
            return Err(ContractError::Unauthorized {});
        }

        // Update fee collector
        if let Some(fee_collector) = fee_collector {
            config.fee_collector = Some(addr_validate_to_lower(deps.api, fee_collector.
                as_str())?);
        }

        // Set generator only if its not set
        if !config.generator_address.is_some() {
            if let Some(generator_address) = generator_address {
                config.generator_address = Some(addr_validate_to_lower(
                    deps.api,
                    generator_address.as_str(),
                )?);
            }
```

```
        }

        // Update LP token code id
        if let Some(lp_token_code_id) = lp_token_code_id {
            config.lp_token_code_id = lp_token_code_id;
        }

        CONFIG.save(deps.storage, &config)?;
        Ok(Response::new().add_attribute("action", "update_config"))
    }
```

Listing 3.9: vault/src/contract.rs::execute_update_config()

```
    pub fn execute_update_pool_config(
        deps: DepsMut,
        info: MessageInfo,
        pool_type: PoolType,
        is_disabled: Option<bool>,
        new_fee_info: Option<FeeInfo>,
    ) -> Result<Response, ContractError> {
        let config = CONFIG.load(deps.storage)?;
        let mut pool_config = REGISTERY
            .load(deps.storage, pool_type.to_string())
            .map_err(|_| ContractError::PoolConfigNotFound {})?;

        // permission check :: If developer address is set then only developer can call
            this function
        if pool_config.fee_info.developer_addr.is_some() {
            if info.sender.clone() != pool_config.fee_info.developer_addr.clone().unwrap
                () {
                return Err(ContractError::Unauthorized {});
            }
        }
        // permission check :: If developer address is not set then only owner can call
            this function
        else {
            if info.sender.clone() != config.owner {
                return Err(ContractError::Unauthorized {});
            }
        }

        // Disable or enable pool instances creation
        if let Some(is_disabled) = is_disabled {
            pool_config.is_disabled = is_disabled;
        }

        // Update fee info
        if let Some(new_fee_info) = new_fee_info {
            if !new_fee_info.valid_fee_info() {
                return Err(ContractError::InvalidFeeInfo {});
            }
            pool_config.fee_info = new_fee_info;
        }
```

```
        // Save pool config
        REGISTERY.save(
            deps.storage,
            pool_config.pool_type.to_string(),
            &pool_config,
        )?;

        Ok(Response::new().add_attribute("action", "update_pool_config"))
    }
```

Listing 3.10: vault/src/contract.rs::execute_update_pool_config()

We understand the need of the privileged functions for proper Dexter operations, but at the same time the extra power to the owner may also be a counter-party risk to the Dexter users. Therefore, we list this concern as an issue here from the audit perspective and highly recommend making these privileges explicit or raising necessary awareness among protocol users.

Recommendation Make the list of extra privileges granted to owner explicit to Dexter users.

Status This issue has been confirmed.

3.8 Revisited Logic in in router::query_simulate_multihop()

- ID: PVE-008
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: router
- Category: Business Logic [8]
- CWE subcategory: CWE-841 [5]

Description

In the Dexter protocol, the router contract allows multihop swaps and provides a query function, i.e., query_simulate_multihop() for users to simulate the multihop swaps result. While reviewing its logic, we notice the current implementation needs to be revisited.

In the following, we show the related code snippet of the query_simulate_multihop() routine. If two or more swaps use the same pool in the given parameter multiswap_request, then the simulated result will not be correct as for the simulation to happen correctly the state of the pool needs to change between queries, which is not possible in a query operation.

```
fn query_simulate_multihop(
  deps: Deps,
  _env: Env,
  multiswap_request: Vec<HopSwapRequest>,
  swap_type: SwapType,
```

```
    amount: Uint128,
) -> StdResult<SimulateMultiHopResponse> {
    let config = CONFIG.load(deps.storage)?;
    let mut simulated_trades: Vec<SimulatedTrade> = vec![];
    let mut fee_response: Vec<Asset> = vec![];

    // Error - If invalid request
    if multiswap_request.len() == 0 {
      return_swap_sim_failure(vec![], "Multiswap request cannot be empty".to_string());
    }

    match swap_type {
      // If we are giving in, we need to simulate the trades in the order of the hops
      SwapType::GiveIn {} => {
        ...
      }
      SwapType::GiveOut {} => {
        ...
      }
      SwapType::Custom(_) => {
        return Ok(return_swap_sim_failure(
          vec![],
          "SwapType not supported".to_string(),
        ))
      }
    }

    Ok(SimulateMultiHopResponse {
      swap_operations: simulated_trades,
      fee: fee_response,
      response: ResponseType::Success {},
    })
}
```

Listing 3.11: `router/src/contract.rs::query_simulate_multihop()`

Recommendation Add additional checks to make sure the input parameter `multiswap_request` of the `query_simulate_multihop()` function will not use the same pool.

Status The issue has been fixed by this commit: `b52178f`.

4 | Conclusion

In this audit, we have analyzed the design and implementation of the `Dexter` protocol, which is a `DEX` implemented as a generalized state transition executor where the transition's math computes are queried from the respective pool contracts. `Dexter` enables a decentralized, non-custodial aggregated liquidity and exchange rate discovery among different tokens on `Persistence`. The current code base is well structured and neatly organized. Those identified issues are promptly confirmed and addressed.

Meanwhile, we need to emphasize that smart contracts as a whole are still in an early, but exciting stage of development. To improve this report, we greatly appreciate any constructive feedbacks or suggestions, on our methodology, audit findings, or potential gaps in scope/coverage.

References

[1] MITRE. CWE-1126: Declaration of Variable with Unnecessarily Wide Scope. https://cwe.mitre.org/data/definitions/1126.html.

[2] MITRE. CWE-197: Numeric Truncation Error. https://cwe.mitre.org/data/definitions/197.html.

[3] MITRE. CWE-287: Improper Authentication. https://cwe.mitre.org/data/definitions/287.html.

[4] MITRE. CWE-837: Improper Enforcement of a Single, Unique Action. https://cwe.mitre.org/data/definitions/837.html.

[5] MITRE. CWE-841: Improper Enforcement of Behavioral Workflow. https://cwe.mitre.org/data/definitions/841.html.

[6] MITRE. CWE CATEGORY: 7PK - Security Features. https://cwe.mitre.org/data/definitions/254.html.

[7] MITRE. CWE CATEGORY: Bad Coding Practices. https://cwe.mitre.org/data/definitions/1006.html.

[8] MITRE. CWE CATEGORY: Business Logic Errors. https://cwe.mitre.org/data/definitions/840.html.

[9] MITRE. CWE CATEGORY: Numeric Errors. https://cwe.mitre.org/data/definitions/189.html.

[10] MITRE. CWE VIEW: Development Concepts. https://cwe.mitre.org/data/definitions/699.html.

[11] OWASP. Risk Rating Methodology. https://www.owasp.org/index.php/OWASP_Risk_Rating_Methodology.

[12] PeckShield. PeckShield Inc. https://www.peckshield.com.