Veridise. Auditing Report

Hardening Blockchain Security with Formal Methods

FOR

Phoenix DeFi Hub

Veridise Inc.

May 3, 2024

► **Prepared For:**

MoonBite
https://app.phoenix-hub.io

► **Prepared By:**

Ian Neal
Alberto Gonzalez

► **Contact Us:** contact@veridise.com

► **Version History:**

Jan. 23, 2024 Initial Draft
Jan. 31, 2024 Official Report.
May. 3, 2024 Official Report V2.

Contents

1 Executive Summary

From Jan. 3, 2024 to Jan. 18, 2024, MoonBite engaged Veridise to review the security of the Phoenix DeFi Hub contracts. The review covered the Rust code associated with the pool contracts and the staking logic. Veridise conducted the assessment over 4 person-weeks, with 2 engineers reviewing code over 2 weeks on commit `d65eef7` and commit `384c8cf`. The auditing strategy involved extensive manual auditing performed by Veridise engineers.

Code assessment. The Phoenix DeFi Hub developers provided the project's source code for review. To enhance the Veridise auditors' understanding of the code, the Phoenix DeFi Hub developers included high-level documentation, such as flow diagrams and written descriptions of the intended usage. It is also important to emphasize the clarity and well-structured nature of the code, which enabled auditors to focus on its security aspects.

The Veridise auditors made use of the test suite provided by the developers to enhance their understanding of the source code. While this test suite was very helpful in demonstrating some of the logic as well as the intended usage of the contracts, the Veridise auditors did note that the coverage of the tests could be improved. For example, some paths involving less common inputs, such as negative numbers or access control lacked sufficient coverage.

Summary of issues detected. The audit uncovered 35 issues, 9 of which are assessed to be of high or critical severity by the Veridise auditors. Specifically: V-PHX-VUL-002 identified a potential draining of the contract's funds; V-PHX-VUL-001 identified an incorrect access control that would have allowed changing critical pool configuration; and both V-PHX-VUL-003 and V-PHX-VUL-006 identified potential Denial of Service (DoS) patterns arising due to the Soroban contract's storage layout. The Veridise auditors also identified 4 medium-severity issues: V-PHX-VUL-013 identified the potential draining of the pool's funds when using tokens with callbacks; and both V-PHX-VUL-009 and V-PHX-VUL-012 identified issues with the user's slippage protection mechanism. In addition, the Veridise auditors uncovered 6 low-severity issues, 11 warnings, and 5 informational findings.

Recommendations. After auditing the protocol, the auditors had a few suggestions to improve the security of the Phoenix DeFi Hub contracts.

Simplify staking logic. The current staking logic involves tracking user bonds separately by pushing them to a vector. However, it utilizes global variables like "withdrawn amount" and "shares correction" for calculating staking rewards, making it challenging to discern each bond's contribution to these variables, when unbonding happens. This design introduces potential risks, as highlighted in V-PHX-VUL-007. We suggest simplifying the logic by maintaining a global accumulator of rewards, a user-specific accumulator, and the user-specific total tokens bonded. This simplified logic would allow the staking logic to calculate the user's rewards as the difference between the corresponding accumulators multiplied by the user's total tokens bonded while obviating the need to track individual bond amounts.

Improve the test suite. Although the codebase has a good test coverage, the auditors identified areas of improvements regarding the following, non-exhaustive list of aspects:

- Include access control tests for all privileged functionality.
- Include tests for user-flow functionality using uncommon inputs.
- Include tests for the staking logic that involve different bonds and unbonds by the same user, ensuring the correct computation of rewards.
- Include more tests for the slippage protection mechanism during swaps, ensuring the user is always protected in case of price movements.

Given the number of high and critical severity issues found by auditors, we would recommend a brief follow-up audit once the fixes have been made and reviewed.

Disclaimer. We hope that this report is informative but provide no warranty of any kind, explicit or implied. The contents of this report should not be construed as a complete guarantee that the system is secure in all dimensions. In no event shall Veridise or any of its employees be liable for any claim, damages or other liability, whether in an action of contract, tort or otherwise, arising from, out of or in connection with the results reported here.

2 Project Dashboard

Table 2.1: Application Summary.

Name	Version	Type	Platform
Phoenix DeFi Hub	`d65eef7` and `384c8cf`	Rust	Soroban

Table 2.2: Engagement Summary.

Dates	Method	Consultants Engaged	Level of Effort
Jan. 3 - Jan. 18, 2024	Manual	2	4 person-weeks

Table 2.3: Vulnerability Summary.

Name	Number	Fixed	Acknowledged
Critical-Severity Issues	2	2	2
High-Severity Issues	7	7	7
Medium-Severity Issues	4	3	4
Low-Severity Issues	6	5	6
Warning-Severity Issues	11	10	11
Informational-Severity Issues	5	5	5
TOTAL	35	32	35

Table 2.4: Category Breakdown.

Name	Number
Maintainability	8
Data Validation	7
Logic Error	7
Usability Issue	5
Access Control	2
Denial of Service	2
Centralization	1
Race Condition	1
TokenDecimals	1
Missing/Incorrect Events	1

3 Audit Goals and Scope

3.1 Audit Goals

The engagement was scoped to provide a security assessment of Phoenix DeFi Hub's smart contracts. In our audit, we sought to answer questions such as:

- Can funds be locked in the contract?
- Can users steal funds from the contract?
- Can non-trusted users change critical configurations in the contracts?
- Do the contracts make incorrect assumptions about the Soroban environment?
- Do the computations correctly take into account the operands' decimal precision?
- Are standard AMM invariants maintained?
- Are users appropriately protected from slippage?
- Is the staking rewards computation correct?
- Can users prevent other users from receiving their staking rewards?

3.2 Audit Methodology & Scope

Audit Methodology. Expert auditors from Veridise manually scrutinized the codebase, examining vulnerabilities and logic flaws. The evaluation included a comprehensive analysis of the provided test suite, ensuring coverage of diverse scenarios. The Phoenix DeFi Hub documentation was carefully reviewed to understand intended functionality and design considerations. Regular meetings with Phoenix DeFi Hub developers facilitated a dynamic exchange, clarifying queries and providing additional context. Our approach aimed not only to address specific concerns, such as those outlined in the questions, but also to uncover hidden vulnerabilities and enhance the overall security resilience of the smart contracts.

Scope. Veridise auditors reviewed on commits `d65eef7`and `384c8cf`. The code change affected the contracts under the *stable_pool* folder, which we reviewed after the new code was pushed.

3.3 Classification of Vulnerabilities

When Veridise auditors discover a possible security vulnerability, they must estimate its severity by weighing its potential impact against the likelihood that a problem will arise. Table 3.1 shows how our auditors weigh this information to estimate the severity of a given issue.

In this case, we judge the likelihood of a vulnerability as follows in Table 3.2:

In addition, we judge the impact of a vulnerability as follows in Table 3.3:

Table 3.1: Severity Breakdown.

	Somewhat Bad	Bad	Very Bad	Protocol Breaking
Not Likely	Info	Warning	Low	Medium
Likely	Warning	Low	Medium	High
Very Likely	Low	Medium	High	Critical

Table 3.2: Likelihood Breakdown

Not Likely	A small set of users must make a specific mistake
Likely	Requires a complex series of steps by almost any user(s) - OR - Requires a small set of users to perform an action
Very Likely	Can be easily performed by almost anyone

Table 3.3: Impact Breakdown

Somewhat Bad	Inconveniences a small number of users and can be fixed by the user
Bad	Affects a large number of people and can be fixed by the user - OR - Affects a very small number of people and requires aid to fix
Very Bad	Affects a large number of people and requires aid to fix - OR - Disrupts the intended behavior of the protocol for a small group of users through no fault of their own
Protocol Breaking	Disrupts the intended behavior of the protocol for a large group of users through no fault of their own

4 Vulnerability Report

In this section, we describe the vulnerabilities found during our audit. For each issue found, we log the type of the issue, its severity, location in the code base, and its current status (i.e., acknowledged, fixed, etc.). Table 4.1 summarizes the issues discovered:

Table 4.1: Summary of Discovered Vulnerabilities.

ID	Description	Severity	Status
V-PHX-VUL-001	Incorrect access control when updating pool c	Critical	Fixed
V-PHX-VUL-002	Pool contract can be drained due to negative r	Critical	Fixed
V-PHX-VUL-003	Unbounded instance storage	High	Fixed
V-PHX-VUL-004	Factory can be made to deploy malicious pool	High	Fixed
V-PHX-VUL-005	Deployment of pools can be front-runned	High	Fixed
V-PHX-VUL-006	The usage of assert_max_spread assume pool	High	Fixed
V-PHX-VUL-007	Any user can DoS important functionality of t	High	Fixed
V-PHX-VUL-008	Unbound breaks the reward distribution	High	Fixed
V-PHX-VUL-009	Incorrect return_amount in stable pool	High	Fixed
V-PHX-VUL-010	Split deposit should target the new pool ratio	Medium	Fixed
V-PHX-VUL-011	Soroban Storage DoS Pattern in Factory contra	Medium	Acknowledged
V-PHX-VUL-012	The do_swap function only allows belief price	Medium	Fixed
V-PHX-VUL-013	Multihop swaps do not allow to express belief	Medium	Fixed
V-PHX-VUL-014	Incorrect assignment of total_fee_bps	Low	Fixed
V-PHX-VUL-015	Invalid value returned by total_comission_am	Low	Fixed
V-PHX-VUL-016	User can accidentally swap the wrong asset	Low	Fixed
V-PHX-VUL-017	Soroban storage limitation discourages staking	Low	Acknowledged
V-PHX-VUL-018	Incorrect computation of return amount	Low	Fixed
V-PHX-VUL-019	Missing max_spread validation	Low	Fixed
V-PHX-VUL-020	Incorrect decimals assertion	Warning	Fixed
V-PHX-VUL-021	Incorrect event topic when providing liquidity	Warning	Fixed
V-PHX-VUL-022	Max referral fee cannot be changed	Warning	Fixed
V-PHX-VUL-023	Missing, incomplete, or redundant basis-point	Warning	Fixed
V-PHX-VUL-024	Deployer redundancies	Warning	Fixed
V-PHX-VUL-025	LP token's metadata is too simple	Warning	Fixed
V-PHX-VUL-026	LP token's decimals should be constant	Warning	Fixed
V-PHX-VUL-027	Compilation errors	Warning	Fixed
V-PHX-VUL-028	Static tolerance value may lead to high imprec	Warning	Fixed
V-PHX-VUL-029	Incorrect decimal handling in the decimal pac	Warning	Acknowledged
V-PHX-VUL-030	Curve combinations become increasingly expe	Warning	Fixed
V-PHX-VUL-031	Documentation and naming issues	Info	Fixed
V-PHX-VUL-032	Unused code and data types	Info	Fixed
V-PHX-VUL-033	Lack of validation on total_shares	Info	Fixed
V-PHX-VUL-034	Unchecked assumptions of get_deposit_amou	Info	Fixed
V-PHX-VUL-035	Unnecessary referral unwrapping	Info	Fixed

4.1 Detailed Description of Issues

4.1.1 V-PHX-VUL-001: Incorrect access control when updating pool configuration

Severity	Critical	**Commit**	d65eef7
Type	Access Control	**Status**	Fixed
File(s)	`contracts/pool/src/contract.rs`		
Location(s)	update_config		
Confirmed Fix At	https://github.com/Phoenix-Protocol-Group/phoenix-contracts/pull/188		

The `update_config` function in the pool contract employs access control by checking if the `sender` matches the admin. However, the critical flaw here is that `sender` is derived from a parameter passed to the function, rather than validating that the admin saved in the storage has authorized the contract invocation.

```
if sender != utils::get_admin(&env) {
   panic!("Pool: UpdateConfig: Unauthorized");
}
```

Snippet 4.1: Code snippet from the `update_config` function in the `contracts/pool/src/contract.rs` file. Validation that `sender` is equal to `admin`.

Impact This issue will allow any user to manipulate the configuration of the pool by passing the `admin` address as the `sender` parameter.

Recommendation Implement an access control pattern similar to the one found in the `upgrade` function:

```
let admin: Address = utils::get_admin(&env);
admin.require_auth();
```

Snippet 4.2: Code snippet from the `upgrade` function. It validates that the `admin` has authorized the current invocation.

Developer Response The recommendation to fetch the admin address and require its authentication has been implemented.

4.1.2 V-PHX-VUL-002: Pool contract can be drained due to negative referral fee

Severity	Critical	**Commit**	d65eef7
Type	Data Validation	**Status**	Fixed
File(s)	`contracts/pool/src/contract.rs`		
Location(s)	do_swap		
Confirmed Fix At	`https://github.com/Phoenix-Protocol-Group/phoenix-contracts/pull/191`		

The `swap` function's execution flow currently includes a validation check in the `do_swap` function to ensure that the `referral_fee` does not exceed the maximum fee allowed. However, there is a significant oversight as there is no check to prevent the `referral_fee` from being negative. The way the `referral_fee` is applied within the `compute_swap` function illustrates the problem:

```
let referral_fee_amount: i128 = return_amount * Decimal::bps(referral_fee);

let return_amount: i128 = return_amount - referral_fee_amount;

ComputeSwap {
    return_amount,
    spread_amount,
    commission_amount,
    referral_fee_amount,
}
```

Snippet 4.3: Code snippet from the `compute_swap` function.

In the above snippet, a negative `referral_fee` results in a negative `referral_fee_amount`, which when subtracted from `return_amount`, increases the `return_amount`.

Impact This issue allows the potential misuse of the `referral_fee` to improperly inflate the `return_amount`, essentially enabling an attacker to drain resources from the pool.

Recommendation It is crucial to implement a validation check to ensure that the `referral_fee` is not negative.

Developer Response The development team has disabled the referral feature and the issue no longer exists.

4.1.3 V-PHX-VUL-003: Unbounded instance storage

Severity	High	**Commit**	d65eef7
Type	Denial of Service	**Status**	Fixed
File(s)	`contracts/factory/src/storage.rs`		
Location(s)	save_lp_vec, save_lp_vec_with_tuple_as_key		
Confirmed Fix At	`https://github.com/Phoenix-Protocol-Group/phoenix-contracts/pull/189/files`		

In the factory contract, the methods for storing information about the liquidity pools (LPs) created, namely `save_lp_vec` and `save_lp_vec_with_tuple_as_key`, currently use instance storage. These methods respectively append new pools to a vector of existing pools and use a tuple of the tokens forming the pool as a key for storage. The use of instance storage is problematic due to its limited capacity, which is not suitable for storing unbounded data.

Reference

Impact The primary issue with using instance storage in this context is its capacity limitation and the incremental cost of increasing the instance storage. As the number of pools increases, the storage limit will eventually be reached, at which point the factory contract will be unable to deploy additional pools.

Recommendation To address this scalability issue, it is recommended to switch from instance storage to persistent storage for storing the vectors of LPs and the tuples used as keys. Persistent storage does not have the same capacity limitations as instance storage, making it a more suitable choice for data that is expected to grow indefinitely.

Developer Response The development team has implemented the recommendation to use persistent storage for both `lp_vec` and `lp_vec_tuple_as_key`.

4.1.4 V-PHX-VUL-004: Factory can be made to deploy malicious pools

Severity	High	**Commit**	d65eef7
Type	Centralization	**Status**	Fixed
File(s)	`contracts/factory/src/contract.rs`, `contracts/pool/src/contract.rs`		
Location(s)	create_liquidity_pool, initialize		
Confirmed Fix At	`https://github.com/Phoenix-Protocol-Group/phoenix-contracts/pull/194`		

The `create_liquidity_pool` function in the `factory` contract utilizes the `deploy_lp_contract` function to deploy new pool contracts. This deployment process involves using a combination of `tokenA` and `tokenB` addresses to generate a salt and the `lp_wasm_hash` provided by the user to locate the contract code to be deployed.

The function allows `lp_wasm_hash`, which specifies the contract code to be deployed, to be provided by the user. This design permits a malicious user to input any `lp_wasm_hash`, potentially leading the factory to deploy arbitrary and potentially malicious contract code. This issue is mirrored in the `initialize` function of the `LiquidityPool` contract, where `token_wasm_hash` and `stake_wasm_hash` are similarly provided by user input, creating a risk that arbitrary contracts could be initialized.

Impact Given that users expect the contract code deployed by the factory to be consistent and secure, the ability to deploy arbitrary code undermines the trust and integrity of the entire platform.

Recommendation To mitigate these risks, it is crucial to restrict the source of the wasm hash values to a trusted and verifiable origin. The contract should store the hashes for `lp_wasm`, `token_wasm`, and `stake_wasm` within its own configuration storage, rather than accepting them via user inputs.

Developer Response Following the recommendation, the contract has been updated to store the wasm hash values for the `pool`, `stake` and `token` contracts within its own configuration.

4.1.5 V-PHX-VUL-005: Deployment of pools can be front-runned

Severity	High	**Commit**	d65eef7
Type	Race Condition	**Status**	Fixed
File(s)	`contracts/factory/src/contract.rs`		
Location(s)	create_liquidity_pool		
Confirmed Fix At	`https://github.com/Phoenix-Protocol-Group/phoenix-contracts/pull/190`		

The pool contract diverges from a permission-less model as it incorporates a privileged account, commonly referred to as the admin. This admin possesses the authority to modify crucial configurations within the pool contract. Notably, the admin is empowered to upgrade the contract's WebAssembly (wasm) through the dedicated `upgrade` function. Thus, the identity of the party deploying a pool is a critical aspect of the system.

It's important to note that the identifier (address) of the pool contract relies on the `deployer` account (linked to the factory contract) and the addresses of the associated token pair, namely `tokenA` and `tokenB`. Consequently, the system is designed to allow only one pool to exist for a given pair of tokens. However, a concern arises as any user has the capability to deploy a pool for any token pair, irrespective of ownership or legitimate authorization.

Impact Since each pair of tokens can only have one pool, a malicious user could exploit this by creating pools for specific tokens, like `tokenB` and `USDC`. This action allows the malicious actor to hinder the genuine creators of `tokenB` from utilizing the system.

Recommendation Only whitelisted accounts should be able to use the factory contract to create pools.

Developer Response The recommendation was implemented. The contract has a vector of whitelisted accounts for pool deployments.

4.1.6 V-PHX-VUL-006: The usage of assert_max_spread assume pool tokens have the same amount of decimals

Severity	High	**Commit**	d65eef7
Type	TokenDecimals	**Status**	Fixed
File(s)	`contracts/pool/src/contract.rs`		
Location(s)	do_swap, assert_max_spread		
Confirmed Fix At	`https://github.com/Phoenix-Protocol-Group/phoenix-contracts/pull/280`		

The `do_swap` function allows users to specify a `belief_price` for slippage protection during swaps, ensuring they do not execute unfavorable transactions. Initially, `belief_price` is converted to a `Decimal` representing a percentage:

```
1 let belief_price = belief_price.map(Decimal::percent);
```

Snippet 4.4: Code snippet from the `do_swap` function. The code transforms `belief_price` to a `Decimal`.

Given this setup, a `belief_price` of `100` translates to a `Decimal` numerator of `1e18`, indicating an expected 1:1 swap ratio.

In the `assert_max_spread` function, the `expected_return` is calculated based on the `belief_price` :

```
1 let expected_return = belief_price.map(|price| offer_amount * price);
```

Snippet 4.5: Code snippet from the `assert_max_spread` function. The code computes the `expected_return` from the swap operation.

However, if `belief_price` equates to `1e18`, it results in: `offer_amount * 1e18 / 1e18`, which simplifies to `offer_amount`. This poses a problem if the tokens involved in `offer_amount` and `expected_return` have different decimal precisions.

Impact Depending on which token has more and less decimal precision, the execution will panic due to the `spread` being too much, or it will continue at the possibility of the user making a bad swap.

Recommendation The `belief_price` parameter should take into account the tokens decimal precision.

Developer Response The development team has changed the implementation from using `belief_price` to using a `min_amount` parameter, effectively addressing this issue.

4.1.7 V-PHX-VUL-007: Any user can DoS important functionality of the stake contract

Severity	High	**Commit**	384c8cf
Type	Access Control	**Status**	Fixed
File(s)	`contracts/stake/src/contract.rs`		
Location(s)	create_distribution_flow		
Confirmed Fix At	`https://github.com/Phoenix-Protocol-Group/phoenix-contracts/pull/249`		

In the `stake` contract, some key functions like `bond` and `distribute_rewards` iterate over all distributions recorded in the contract using a `for` loop. This loop accesses a list of distributions, growing unbounded depending on how many distributions exist. However, the process for creating new distributions, handled by the `create_distribution_flow` function, lacks access control, allowing any user to create an unlimited number of distributions.

Impact This lack of control expose the contract to a potential Denial of Service (DoS) attack, where a malicious user could create an excessive number of distributions.

Recommendation The ability to create distributions should be restricted to authorized roles.

Developer Response Access control has been added to `create_distribution_flow` such that only the contract's owner or manager can invoke the function.

4.1.8 V-PHX-VUL-008: Unbound breaks the reward distribution

Severity	High	**Commit**	384c8cf
Type	Logic Error	**Status**	Fixed
File(s)	`contracts/stake/src/contract.rs`		
Location(s)	unbound		
Confirmed Fix At	`https://github.com/Phoenix-Protocol-Group/phoenix-contracts/pull/270`		

When calculating withdrawable rewards, the `stake` contract multiplies the global accumulator `distribution.shares_per_point` by the user's `total_stake`. It then adjusts this amount by subtracting the `shares_correction` (to account for deposits made before the reward period) and `adjustment.withdrawn_rewards` (to exclude previously claimed rewards). However, there's an oversight: when a user unbonds part of their stake, the calculations still include the `shares_correction` and `withdrawn_rewards` for that stake's part.

Impact This flaw disrupts accurate reward distribution. Specifically, even after a stake is unbonded, its corrections continue to be considered, leading to lower reward calculations or completely break the distribution.

Recommendation Adjust the contract to stop counting `shares_correction` and `withdrawn_rewards` from stakes that are no longer active.

Developer Response Following the recommendation, after unbounding the `shares_correction` and `withdrawn_rewards` are updated accordingly.

4.1.9 V-PHX-VUL-009: Incorrect return_amount in stable pool

Severity	High	**Commit**	384c8cf
Type	Logic Error	**Status**	Fixed
File(s)	`contracts/stable_pool/src/contract.rs`		
Location(s)	compute_swap		
Confirmed Fix At	`https://github.com/Phoenix-Protocol-Group/phoenix-contracts/pull/233`		

The `compute_swap` function in the stable pool contract is intended to calculate the results of a swap, including the ask token amount (return_amount), swap commission, and spread. However, an issue arises as this function does not subtract the `commission_amount` from the `return_amount`.

Impact This discrepancy affects the `do_swap` function, where the `return_amount` calculated by `compute_swap` is transferred directly to the user without deducting the `commission_amount`. This results in users receiving more tokens than they should.

Recommendation Modify the `compute_swap` function in the stable pool contract to subtract the `commission_amount` from the `return_amount` before returning it.

Developer Response The developers have updated the `compute_swap` function to correctly deduct the `commission_amount` from the `return_amount`.

4.1.10 V-PHX-VUL-010: Split deposit should target the new pool ratio

Severity	Medium	**Commit**	d65eef7
Type	Logic Error	**Status**	Fixed
File(s)	`pool/scr/contract.rs`		
Location(s)	split_deposit_based_on_pool_ratio		
Confirmed Fix At	`https://github.com/Phoenix-Protocol-Group/phoenix-contracts/pull/289`		

The function `split_deposit_based_on_pool_ratio` tries to match the amount deposited to the ratio of balances held by the pool. However, the target ratio is based on the pool ratio before the swap occurs.

Impact After the swap occurs, the ratio changes, and so the user is rewarded fewer LP shares and ends up losing equity even on a fee-less pool.

Recommendation Use of a precise formula for computing `split_deposit_based_on_pool_ratio` rather than the current binary search method. This would not only make the calculation more accurate, but much more efficient.

Developer Response The single-side liquidity feature was removed from the code.

4.1.11 V-PHX-VUL-011: Soroban Storage DoS Pattern in Factory contract

Severity	Medium	**Commit**	d65eef7
Type	Denial of Service	**Status**	Acknowledged
File(s)	`contracts/factory/src/contract.rs`		
Location(s)	create_liquidity_pool		
Confirmed Fix At			

Soroban transactions include something called the `Footprint`. The footprint of a transaction declares the Ledger Keys that the transaction will read or write during its execution. At the beginning of the execution, Soroban loads to the Host Environment every ledger entry associated with the declared ledger keys on the `Footprint`.

On the factory contract we can find the following code in the `create_liquidity_pool` function:

```
let mut lp_vec = get_lp_vec(&env);

lp_vec.push_back(lp_contract_address.clone());

save_lp_vec(&env, lp_vec);
```

Snippet 4.6: Code snippet from the `create_liquidity_pool` function. It loads a vector containing all the created pools, appends an element, and then saves it.

Given the above, then we know that in order to execute the function `create_liquidity_pool` the footprint of the transaction must include the Ledger Key associated with this vector. Then, Soroban will load the corresponding Ledger Entry. But this Ledger Entry is unbounded in nature, it grows every time we create a new pool.

Impact The creation of a pool will get more expensive due to the necessity of loading the `lp_vec` which grows in size after every execution. At some point, the creation of new pools will be impossible due to network limits.

The current per-transaction limit of reading bytes is of 130 KB. The vector increases 40 bytes each time a new address is appended. Hence, it will take 3250 deployed pools to reach the limit.

Reference

Recommendation Let an off-chain system to re-create the vector of created pools.

Developer Response Acknowledged, won't fix. Contract will not handle more than 100 pools.

4.1.12 V-PHX-VUL-012: The do_swap function only allows belief prices down to 1%.

Severity	Medium	**Commit**	384c8cf
Type	Usability Issue	**Status**	Fixed
File(s)	`contracts/pool/src/contract.rs`		
Location(s)	do_swap		
Confirmed Fix At	`https://github.com/Phoenix-Protocol-Group/phoenix-contracts/pull/280`		

The `do_swap` function allows users to specify a `belief_price` for slippage protection, ensuring they don't execute unfavorable swaps. Initially, `belief_price` is converted into a `Decimal` percentage:

```
let belief_price = belief_price.map(Decimal::percent);
```

Snippet 4.7: Code snippet from the `do_swap` function. The code transforms `belief_price` to a `Decimal`.

This transformation implies that the smallest `belief_price` a user can specify is `1`, resulting in a `Decimal` with a numerator of `1e16`. In the `assert_max_spread` function, the `expected_return` from the swap operation is calculated:

```
let expected_return = belief_price.map(|price| offer_amount * price);
```

Snippet 4.8: Code snippet from the `assert_max_spread` function. The code computes the `expected_return` from the swap operation.

With `belief_price` set to `1e16`, this results in `offer_amount * 1e16 / 1e18`, equating to 1% of `offer_amount`. This calculation becomes problematic for token pairs where the price relationship is less than 1%, such as `USDC` to `ETH`, where the ratio is about 0.03% (`1 ETH / 2600 USDC`).

Impact Users will not have protection for price fluctuations for token pairs with a price relation smaller than 1%.

Recommendation Allow a greater range of `belief_price`.

Developer Response The development team has changed the implementation from using `belief_price` to using a `min_amount` parameter, effectively addressing this issue.

4.1.13 V-PHX-VUL-013: Multihop swaps do not allow to express belief price for all the swap operations

Severity	Medium	**Commit**	d65eef7
Type	Usability Issue	**Status**	Fixed
File(s)	`contracts/multihop/src/contract.rs`		
Location(s)	swap		
Confirmed Fix At	`https://github.com/Phoenix-Protocol-Group/phoenix-contracts/pull/234`		

The `multihop` contract includes a `swap` function designed for executing complex swaps across multiple pools, such as swapping from `tokenA` to `tokenC` via an intermediate `tokenB`. This function, however, currently only accepts a single `belief_price` for the entire sequence of swaps, rather than individual `belief_prices` for each leg of the swap sequence.

Impact Users risk financial loss from price fluctuations because they can't set distinct `belief_prices` for each swap segment.

Recommendation Modify the `swap` function to accept a vector of `belief_prices`, allowing one for each swap operation.

Developer Response The recommendation was adopted, enabling individual `belief_prices` for each swap.

Update: The `belief_price` was removed.

4.1.14 V-PHX-VUL-014: Incorrect assignment of total_fee_bps

Severity	Low	**Commit**	d65eef7
Type	Logic Error	**Status**	Fixed
File(s)	`contracts/pool/src/contract.rs`		
Location(s)	query_pool_info_for_factory		
Confirmed Fix At	`https://github.com/Phoenix-Protocol-Group/phoenix-contracts/pull/235`		

The `query_pool_info_for_factory` function in the pool contract retrieves information structured as `LiquidityPoolInfo`, which includes fields such as `pool_address`, `pool_response`, and `total_fee_bps`. The `total_fee_bps` is intended to represent the transaction fee percentage for each swap within the pool. However, there is a misassignment where `total_fee_bps` is erroneously set to `max_allowed_spread_bps` from the pool configuration.

Impact This misalignment can misinform third-party integrations and users, leading to incorrect assumptions about transaction costs, potentially affecting decision-making processes related to swaps and investments in the pool.

Recommendation Correct the assignment within the `query_pool_info_for_factory` function.

Developer Response Recommendation was implemented.

4.1.15 V-PHX-VUL-015: Invalid value returned by total_comission_amount.

Severity	Low	**Commit**	d65eef7
Type	Logic Error	**Status**	Fixed
File(s)	`contracts/multihop/src/contract.rs`		
Location(s)	simulate_swap, simulate_reverse_swap		
Confirmed Fix At	`https://github.com/Phoenix-Protocol-Group/phoenix-contracts/pull/236`		

The `simulate_swap` function in the `multihop` contract allows users to simulate sequential swap operations across different liquidity pools. During the simulation, the function calculates the outcomes for each swap, adjusting the input for the next based on the output of the previous swap.

However, the function incorrectly accumulates `total_commission_amount` by summing the commission amounts of each swap, despite these commissions being denominated in different tokens. This results in a meaningless total, as it mixes values across different token denominations.

Impact This aggregation leads to an invalid total commission amount, causing confusion and potentially misleading users about the cost implications of their intended transactions.

Recommendation To resolve this, the function should store commission amounts in a vector, similar to how spread amounts are handled. Each entry in the vector would represent the commission for a corresponding swap, maintaining clarity about the costs in their respective tokens.

Developer Response The developers have updated the function to store commission amounts in a vector format.

4.1.16 V-PHX-VUL-016: User can accidentally swap the wrong asset

Severity	Low	**Commit**	384c8cf
Type	Data Validation	**Status**	Fixed
File(s)	`contracts/pool/src/contract.rs`		
Location(s)	do_swap, simulate_reverse_swap, simulate_swap, split_deposit_based_on_pool_ratio		
Confirmed Fix At	https://github.com/Phoenix-Protocol-Group/phoenix-contracts/pull/237		

In the liquidity pool operations such as `do_swap`, `simulate_swap`, `simulate_reverse_swap`, and `split_deposit_based_on_pool_ratio`, there's a common check pattern to determine if the `offer_asset` (or `ask_asset` in the case of `simulate_reverse_swap`) matches `config.token_a` or `config.token_b`. The code then assigns pool balances based on this condition. However, this approach has a significant oversight: if `offer_asset` is neither `config.token_a` nor `config.token_b`, the function defaults to treating it as if it were `config.token_b`.

```
let (pool_balance_sell, pool_balance_buy) = if offer_asset == config.token_a {
    (pool_balance_a, pool_balance_b)
} else {
    (pool_balance_b, pool_balance_a)
};
```

Snippet 4.9: Check for `config.token_a` or `config.token_b`, taken from `do_swap`.

Impact A user could accidentally swap their balance of `token_a` or `token_b` if they try to swap `token_c`. This could happen if the user erroneously invokes a swap on the wrong liquidity pool.

Recommendation Abort the transaction if the user specifies an asset that is not `token_a` nor `token_b` in each of these functions.

Developer Response Explicit checks that `asset` is either `token_a` or `token_b` are implemented.

4.1.17 V-PHX-VUL-017: Soroban storage limitation discourages staking

Severity	Low	**Commit**	384c8cf
Type	Usability Issue	**Status**	Acknowledged
File(s)	`contracts/stake/src/storage.rs`		
Location(s)	BondingInfo		
Confirmed Fix At			

Similar to the V-PHX-VUL-011 issue, the `stakes` vector in a user's `BondingInfo` grows with each subsequent `bond` operation that a user performs. Therefore, each `bond` operation will become more and more expensive for the user to perform and could eventually exceed the transaction storage limit.

Impact This impacts the usability of the staking contract and discourages users from having many stakes in the contract.

Recommendation Since the `stake_timestamp` of each stake is not checked or used after a stake is created, and the `WithdrawAdjustment` tracks adjustments based on the time the user performs a `bond`, the `BondingInfo` could solely track a user's `total_stake` and omit storing all of the user's individual `stake` operations.

Developer Response Acknowledged.

4.1.18 V-PHX-VUL-018: Incorrect computation of return amount

Severity	Low	**Commit**	d65eef7
Type	Logic Error	**Status**	Fixed
File(s)	`contracts/pool/src/contract.rs`		
Location(s)	do_swap		
Confirmed Fix At	`https://github.com/Phoenix-Protocol-Group/phoenix-contracts/pull/238`		

In the function call to `assert_max_spread`, the `return_amount` is currently calculated by adding `compute_swap.return_amount` and `compute_swap.commission_amount`. This calculation, however, overlooks the `compute_swap.referral_fee_amount`.

```
assert_max_spread(
    &env,
    belief_price,
    max_spread,
    offer_amount,
    compute_swap.return_amount + compute_swap.commission_amount,
    compute_swap.spread_amount,
);
```

Snippet 4.10: `assert_max_spread` invocation.

Impact Neglecting the `referral_fee_amount` leads to an underestimation of the total return amount.

Recommendation Update the `return_amount` argument to be `compute_swap.return_amount + compute_swap.commission_amount + compute_swap.referral_fee_amount`.

Developer Response Recommendation was implemented.

4.1.19 V-PHX-VUL-019: Missing max_spread validation

Severity	Low	**Commit**	d65eef7
Type	Data Validation	**Status**	Fixed
File(s)	`contracts/pool/src/contract.rs`		
Location(s)	do_swap		
Confirmed Fix At	`https://github.com/Phoenix-Protocol-Group/phoenix-contracts/pull/239`		

The `do_swap` function does not currently ensure that `max_spread` is within acceptable limits; it neither confirms that it is non-negative nor that it does not exceed `config.max_allowed_spread_bps`.

```
1 let max_spread = Decimal::bps(max_spread.map_or_else(|| config.max_allowed_spread_bps
      , |x| x));
```

Snippet 4.11: `max_spread` initialization in `do_swap`.

Impact `assert_max_spread` may not catch spread errors if `max_spread` is greater than `config.max_allowed_spread_bps`, and `assert_max_spread` will always fail if `max_spread` is negative.

Recommendation Validate that `max_spread` is not greater than `config.max_allowed_spread_bps` and validate that is not negative.

Developer Response The function now asserts that `max_spread` is greater than or equal to 0 and less than or equal to `config.max_allowed_spread_bps`.

4.1.20 V-PHX-VUL-020: Incorrect decimals assertion

Severity	Warning	**Commit**	d65eef7
Type	Data Validation	**Status**	Fixed
File(s)	`packages/decimal/src/lib.rs`		
Location(s)	from_str		
Confirmed Fix At	https://github.com/Phoenix-Protocol-Group/phoenix-contracts/pull/197		

The function `from_str` in the `DECIMAL` crate is used to convert a string to a decimal type. To handle the fractional part of the number, the function has the following logic:

```
if let Some(fractional_part) = parts_iter.next() {
   let fractional: i128 = fractional_part.parse().expect("Error parsing fractional");
   let exp = Self::DECIMAL_PLACES - fractional_part.len() as i32;
   assert!(exp <= Self::DECIMAL_PLACES, "Too many fractional digits");
   let fractional_factor = 10i128.pow(exp as u32);
   atomics += fractional * fractional_factor;
}
```

Snippet 4.12: Code snippet from the `from_str` function. It handles the fractional part of a string number.

As it can be noted, the `assert!` is a tautology since `exp` will always be greater or equal than `DECIMAL_PLACES` given that `fractional_part.len()` is a positive number. Given the this, then it is possible for a negative `exp` to pass the `assert!` which later will be casted to a `u32` number.

Impact Currently, there are no immediate consequences to this situation. The limitation arises because the largest string permissible as input can consist of only 36 digits in its fractional part. Exceeding this limit and invoking **`fractional_part.parse()`** would result in a panic. Conversely, the smallest accepted value for **`exp`** is -18. When this is cast to **`u32`**, it transforms into a substantial number, causing the subsequent line with **`pow`** to panic due to an overflow

Recommendation The assert should be changed to `assert!(exp >= 0)`.

Developer Response Recommendation was implemented.

4.1.21 V-PHX-VUL-021: Incorrect event topic when providing liquidity

Severity	Warning	**Commit**	d65eef7
Type	Missing/Incorrect Eve	**Status**	Fixed
File(s)	`contracts/pool/src/contract.rs`		
Location(s)	provide_liquidity		
Confirmed Fix At	`https://github.com/Phoenix-Protocol-Group/phoenix-contracts/pull/195`		

At the end of the function `provide_liquidity` the code publish some events:

```
env.events()
    .publish(("provide_liquidity", "sender"), sender);
env.events()
    .publish(("provide_liquidity", "token_a"), &config.token_a);
env.events()
    .publish(("provide_liquidity", "token_a-amount"), amounts.0);
env.events()
    .publish(("provide_liquidity", "token_a"), &config.token_b);
env.events()
    .publish(("provide_liquidity", "token_b-amount"), amounts.1);
```

Snippet 4.13: Events published in the `provide_liquidity` function.

The 4th event's topic is wrong, it should be `token_b` instead of `token_a`.

Impact Event will be published with the wrong topic.

Recommendation Change `token_a` to `token_b`.

Developer Response Recommendation was implemented.

4.1.22 V-PHX-VUL-022: Max referral fee cannot be changed

Severity	Warning	**Commit**	d65eef7
Type	Maintainability	**Status**	Fixed
File(s)	`contracts/pool/src/contract.rs`		
Location(s)	update_config		
Confirmed Fix At	`https://github.com/Phoenix-Protocol-Group/phoenix-contracts/pull/196/`		

The pool contract has a function named `update_config` which allows the admin of the contract to change some parameters of the current pool's configuration. The current logic of this function does not consider the possibility to change the `max_referral_bps` variable.

Impact Once `max_referral_bps` is set during the initialization of the pool contract, then it cannot be changed.

Recommendation Allow the `admin` of a pool to change the `max_referral_bps` variable during `update_confing`.

Developer Response Recommendation was implemented. The parameter of `max_referral_bps` has been added to the `update_config` function.

4.1.23 V-PHX-VUL-023: Missing, incomplete, or redundant basis-point range checks

Severity	Warning	**Commit**	d65eef7
Type	Data Validation	**Status**	Fixed
File(s)	`contracts/pool/src/contract.rs`		
Location(s)	initialize, update_config		
Confirmed Fix At	`https://github.com/Phoenix-Protocol-Group/phoenix-contracts/pull/199`		

The configuration values `max_allowed_spread_bps`, `max_allowed_slippage_bps`, `max_referral_bps`, and `total_fee_bps` are assumed to be between 0 and 10,000 bps. However, these checks are not explicitly performed, are only partially performed (e.g., `validate_fee_bps` checks that a basis-point value is 10,000 but not 0), or redundantly performed (`swap_fee_bps` is checked twice in initialize).

Impact These inconsistent checks could result in inconsistent configurations, e.g. a negative `max_allowed_spread_bps`.

Recommendation Add a function similar to `validate_fee_bps` that asserts that the supplied basis-points values are between 0 and 10,000 and use this to validate all basis-points configuration values. We also recommend using named constant values, such as `MAX_TOTAL_FEE_BPS`, rather than directly checking against literals.

Developer Response Macro was added to the contract logic.

4.1.24 V-PHX-VUL-024: Deployer redundancies

Severity	Warning	**Commit**	d65eef7
Type	Maintainability	**Status**	Fixed
File(s)	See issue description		
Location(s)	See issue description		
Confirmed Fix At	`https://github.com/Phoenix-Protocol-Group/phoenix-contracts/pull/198`		

The functions:

- `deploy_stake_contract` (contracts/pool/src/storage.rs)
- `deploy_lp_contract` (contracts/factory/src/utils.rs)

Perform a check on `deployer` that will always be false:

```
let deployer = e.current_contract_address();

if deployer != e.current_contract_address() {
    deployer.require_auth();
}
```

Snippet 4.14: Redundant deployer check; the `if` statement will never be taken.

This could be simplified by removing the creation of the `deployer` value and instead using the `with_current_address` function.

```
e.deployer()
  .with_current_address(salt)
    .deploy(stake_wasm_hash)
```

Snippet 4.15: Reduced stake contract deployment in `deploy_stake_contract`.

Impact The current implementations are more verbose and lead to dead code.

Recommendation Simplify the logic in the function.

Developer Response The recommendation was implemented.

4.1.25 V-PHX-VUL-025: LP token's metadata is too simple

Severity	Warning	**Commit**	d65eef7
Type	Maintainability	**Status**	Fixed
File(s)	`contracts/pool/src/contract.rs`		
Location(s)	initialize		
Confirmed Fix At	`https://github.com/Phoenix-Protocol-Group/phoenix-contracts/pull/243`		

In the `initialize` function of the pool contract, the code deploys the LP token contract and initializes it. However, the metadata for the LP token, specifically its name and symbol, is hardcoded to "Pool Share Token" and "POOL" respectively, regardless of the reserve tokens of the pool.

```
token_contract::Client::new(&env, &share_token_address).initialize(
      // admin
      &env.current_contract_address(),
      // number of decimals on the share token
      &share_token_decimals,
      // name
      &"Pool Share Token".into_val(&env),
      // symbol
      &"POOL".into_val(&env),
);
```

Snippet 4.16: Code snippet from the `initialize` function. It initializes the lp token contract.

Impact This results in all LP tokens having identical metadata, which could cause confusion as they do not reflect the specific assets they represent.

Recommendation Modify the initialization parameters to include information about `tokenA` and `tokenB` in the name and symbol of the LP token to better distinguish between different pools.

Developer Response The token name and symbol can now be customized when the token is created.

4.1.26 V-PHX-VUL-026: LP token's decimals should be constant

Severity	Warning	**Commit**	d65eef7
Type	Maintainability	**Status**	Fixed
File(s)	`contracts/factory/src/contract.rs`		
Location(s)	create_liquidity_pool		
Confirmed Fix At	`https://github.com/Phoenix-Protocol-Group/phoenix-contracts/pull/241`		

When deploying a pool via `create_liquidity_pool`, the code initializes the pool using various parameters, including the amount of decimals for the LP token. Currently, this decimal value is determined by user input rather than being hard-coded.

Impact Allowing the decimal count to vary introduces the potential for inconsistency among LP tokens, posing risks for third-party integrations that expect a standard format.

Recommendation It is advisable to set the LP token decimal count as a constant value to ensure uniformity across all tokens.

Developer Response Share token decimals is now part of the Factory configuration instead of given by input.

4.1.27 V-PHX-VUL-027: Compilation errors

Severity	Warning	**Commit**	384c8cf
Type	Maintainability	**Status**	Fixed
File(s)	`contracts/pool_stable/*`		
Location(s)	contracts/pool_stable/		
Confirmed Fix At	`https://github.com/Phoenix-Protocol-Group/phoenix-contracts/issues/224`		

The specified commit does not compile due to the changes made in `pool_stable`. This can be reproduced by running `make test`.

Impact There are errors in the code that need to be corrected.

Recommendation Fix the compilation issues and ensure the tests run successfully.

Developer Response Fixes were implemented so that the main branch now compiles successfully.

4.1.28 V-PHX-VUL-028: Static tolerance value may lead to high imprecision for small pools

Severity	Warning	**Commit**	384c8cf
Type	Usability Issue	**Status**	Fixed
File(s)	`contracts/pool/src/contract.rs`		
Location(s)	split_deposit_based_on_pool_ratio		
Confirmed Fix At	`https://github.com/Phoenix-Protocol-Group/phoenix-contracts/pull/289`		

The `tolerance` value, which is used as the smallest difference in deposit that the pool cares about, is constant (`500`).

Impact For small pools or pools with a large imbalance between tokens, this tolerance may lead to a large percentage of error in deposit amounts.

Recommendation While we recommend rewriting this function entirely (see V-PHX-VUL-010), if the `tolerance` logic is kept, we recommend making it either configurable or based on the size of the underlying balances in the liquidity pool.

Developer Response Acknowledge. Feature was removed.

4.1.29 V-PHX-VUL-029: Incorrect decimal handling in the decimal package

Severity	Warning	**Commit**	384c8cf
Type	Logic Error	**Status**	Acknowledged
File(s)	packages/decimal/src/lib.rs		
Location(s)	to_i128_with_precision		
Confirmed Fix At			

The `to_i128_with_precision` function is used to transform a value of type `decimal` to an `i128` given a `target_precision`. The logic looks like:

```
pub fn to_i128_with_precision(&self, precision: impl Into<i32>) -> i128 {
    let value = self.atomics();
    let precision = precision.into();

    let divisor = 10i128.pow((self.decimal_places() - precision) as u32);
    value / divisor
}
```

Snippet 4.17: Code of the `to_i128_with_precision` function in the `decimal` package.

As it can be seen, the code does not take into account the case where `precision > self.decimal_places()`. Currently, `decimal_places` returns `18`.

Impact Following the flow of execution, we identified that the `calc_y` function invokes the `to_i128_with_precision`. The `calc_y` function is invoked via `compute_swap`:

```
let new_ask_pool = calc_y(
    amp as u128,
    Decimal::from_atomics(offer_pool + offer_amount, 6),
    &[
      Decimal::from_atomics(offer_pool, 6),
      Decimal::from_atomics(ask_pool, 6),
     ],
     6,
);
```

Snippet 4.18: Code snippet from the `compute_swap` function. It calls the `calc_y` function with `6` as target precision.

The current code, uses `6` as target precision, which makes the call to `to_i128_with_precision` safe. But, the developers acknowledge that they will change to use the decimals of the token instead of the hard coded `6`.

Recommendation Handle the case where `precision > self.decimal_places()`.

Developer Response Acknowledged, won't fix. No tokens with more than 18 decimals will be used.

4.1.30 V-PHX-VUL-030: Curve combinations become increasingly expensive

Severity	Warning	**Commit**	384c8cf
Type	Usability Issue	**Status**	Fixed
File(s)	`contracts/stake/src/contracts.rs`		
Location(s)	fund_distribution		
Confirmed Fix At	`https://github.com/Phoenix-Protocol-Group/phoenix-contracts/pull/283`		

When a distribution is funded in the staking contract, a new curve is created and combined with the previous curve.

```
// now combine old distribution with the new schedule
let new_reward_curve = previous_reward_curve.combine(&env, &new_reward_distribution);
save_reward_curve(&env, token_address.clone(), &new_reward_curve);
```

Snippet 4.19: Curve combination in `fund_distribution`.

This combination of curves will become prohibitively expensive over time, as old points in the curve are maintained. This both increases storage costs and sorting overhead, as each combination of two pairwise functions is at least $O(n\hat{}2)$ (as the `curve` package uses the bubble sorting algorithm), where n is the number of points in the curve.

Impact Updating or modifying the reward curve will become increasing expensive over time.

Recommendation We recommend removing old points from the curve that are no longer necessary; for example, if the previous curve's `x_max` is smaller than current timestamp, then you can use directly the new curve without combine it with the previous one.

Developer Response Recommendation is implemented and the maximum complexity for a curve is set to 10.

4.1.31 V-PHX-VUL-031: Documentation and naming issues

Severity	Info	**Commit**	d65eef7
Type	Maintainability	**Status**	Fixed
File(s)	See issue description		
Location(s)	See issue description		
Confirmed Fix At	`https://github.com/Phoenix-Protocol-Group/phoenix-contracts/pull/200`		

Here we list a few issues with the documentation and naming issues throughout the codebase:

- contracts/pool/src/contract.rs
  - Line 35: Comment references `token_wasm_hash` as a parameter, but `token_wash_hash` is actually provided as a member of `token_init_info`.
- contracts/pool/src/storage.rs
  - Line 298: comment states that the amount must be within 1%, but the threshold is configurable
- packages/curve/src/lib.rs
  - Line 2: wynd-contracts URL is out-of-date, use `https://github.com/wynddao/wynddao/` instead
  - Line 44: "Constan" → "Constant"
- packages/decimal/src/lib.rs
  - Lines 133-141: These comments fails rust's doctests.
  - Lines 197-201: Dead code in the comments should be removed to avoid confusion.
- packages/phoenix/src/utils.rs
  - Line 21: Function `assert_approx_ratio` does not actually perform a panic or assert, subverting a user's expectations.

Impact These inconsistencies can result in maintainability issues or future bugs caused by developer misconceptions.

Recommendation Update the documentation.

Developer Response Documentation was updated.

4.1.32 V-PHX-VUL-032: Unused code and data types

Severity	Info	**Commit**	384c8cf
Type	Maintainability	**Status**	Fixed
File(s)	See issue description		
Location(s)	See issue description		
Confirmed Fix At	`https://github.com/Phoenix-Protocol-Group/phoenix-contracts/pull/242`		

Here we list locations of unused code paths, data type definitions, and the like:

- `contracts/multihop/src/contract.rs`
  - Line 55: `admin` is set but not used.
- `contracts/stake/src/contract.rs`
  - Line 118: `max_distributions` is never used.
  - Line 205: `manager` is never used.
- `contracts/stake/src/utils.rs`
  - Line 5: `OptionUint` is unused.
- `packages/decimal/src/lib.rs`
  - Line 180: The case `Err(Error::Overflow)` will never happen since the error is not returned by `checked_from_ratio`.
  - Line 274: Same as Line 180. The project is configured so that overflow is handled by the native rust code rather than a package-specific error.
- `packages/phoenix/src/error.rs`: This file is empty.

Impact These unused items can become outdated easily and make overall code base maintenance more difficult.

Recommendation Remove unused code.

Developer Response Unused code was removed.

4.1.33 V-PHX-VUL-033: Lack of validation on total_shares

Severity	Info	**Commit**	384c8cf
Type	Data Validation	**Status**	Fixed
File(s)	contracts/pool/src/contract.rs		
Location(s)	withdraw_liquidity		
Confirmed Fix At	https://github.com/Phoenix-Protocol-Group/phoenix-contracts/pull/245		

In `withdraw_liquidity`, the number of `total_shares` is checked to ensure it is non-zero before computing `share_ratio` to avoid a divide-by-zero error:

```
let mut share_ratio = Decimal::zero();
let total_shares = utils::get_total_shares(&env);
if total_shares != 0i128 {
    share_ratio = Decimal::from_ratio(share_amount, total_shares);
}
```

Snippet 4.20: `total_shares` check and `share_ratio` computation.

However, if `total_shares` is zero, then there is no way that any liquidity can be withdrawn (as there isn't any liquidity in the pool). So, if `total_shares` is zero here and the transfer of share tokens has occurred, then there is an error in the protocol.

Impact Performing the zero check on `total_shares` without issuing an error if `total_shares` is zero misses an opportunity to report a serious protocol error if it has occurred.

Recommendation Ensure that `total_shares` is not zero and issue a panic otherwise.

Developer Response The recommendation was implemented.

4.1.34 V-PHX-VUL-034: Unchecked assumptions of get_deposit_amounts arguments

Severity	Info	**Commit**	384c8cf
Type	Data Validation	**Status**	Fixed
File(s)	`contracts/pool/src/storage.rs`		
Location(s)	get_deposit_amounts		
Confirmed Fix At	`https://github.com/Phoenix-Protocol-Group/phoenix-contracts/pull/246`		

Based on the tests and usage in `provide_liquidity`, `get_deposit_amounts` assumes that:

- `desired_a` and `desired_b` are assumed to be `> 0`
- `min_a` and `min_b` are `>= 0` if they are not `None`

However, these assumptions are not explicitly checked in the function.

Impact Implicit constraints may be accidentally violated in future versions of the code.

Recommendation Add checks to the arguments passed to `get_deposit_amounts`.

Developer Response Checks for `desired_a` and `desired_b` parameters are implemented.

4.1.35 V-PHX-VUL-035: Unnecessary referral unwrapping

Severity	Info	**Commit**	384c8cf
Type	Maintainability	**Status**	Fixed
File(s)	`contracts/multihop/src/contract.rs`		
Location(s)	swap		
Confirmed Fix At	`https://github.com/Phoenix-Protocol-Group/phoenix-contracts/pull/240`		

The following code:

```
if let Some(referral) = referral.clone() {
    next_offer_amount = lp_client.swap(
        &recipient,
        &Some(referral),
        &op.offer_asset,
        &next_offer_amount,
        &max_belief_price,
        &max_spread_bps,
    );
} else {
    next_offer_amount = lp_client.swap(
        &recipient,
        &None,
        &op.offer_asset,
        &next_offer_amount,
        &max_belief_price,
        &max_spread_bps,
    );
}
```

Snippet 4.21: Code snipped from `swap`.

Unnecessarily unwraps the optional `referral` value.

Impact The redundant code hurts the maintainability of the function.

Recommendation Rewrite the logic as follows:

```
next_offer_amount = lp_client.swap(
                            &recipient,
                            &referral,
                            &op.offer_asset,
                            &next_offer_amount,
                            &max_belief_price,
                            &max_spread_bps,
                        );
```

Snippet 4.22: Suggested rewrite.

Developer Response The referral mechanism was deleted.