Automated Market Maker System: Cryptography and Implementation Review

Kaddex, Inc.

Version 2.0 – July 14, 2022

Prepared By
Eric Schorn
Aleks Kircanski
Kevin Henry
Parnian Alimi

Prepared For
Nicolas Ramsrud
Adrian Cardoso
Daniele De Vecchis
Giuseppe Pace
Gustavo Spelzon
Kate Oztunc

1 Executive Summary

Synopsis

During April through July of 2022, Kaddex Inc. engaged NCC Group's Cryptography Services team to perform a cryptography and implementation review of Kaddex's Automated Market Maker (AMM) System. This system consists of a suite of contracts that facilitate peer-to-peer market making and swapping of tokens on the Kadena blockchain. Full source code was provided, along with support via several video calls and also over a private Slack channel. This document covers the conclusion of both phase 1 and phase 2, which were delivered within 50-person days of total effort.

Scope

The primary materials utilized for this review include:

- Phase 1 targeted Kaddex code within commit `204b93e` of github.com/kaddex-org/wrapper-contract, with primary functional paths involving:
  - `kadenaswap/exchange.pact` – core module for the exchange.
  - `wrapper/wrapper.pact` – mainly provides for: A) boosted KDX rewards for liquidity provider fees, B) adding liquidity with a single side of the pair, and C) utility functions for the front-end to query LP stats.
  - `kadenaswap/tokens.pact` – liquidity tokens used by exchange.
  - `kadenaswap/gas-station/gas-station.pact` – creates and manages the gas station.
  - `wrapper/tokens/kdx.pact` – KDX token module, a modified fungible-v2 token.
  - `staking/staking.pact` – manages exchange fees, tracks users, and distributes rewards.
    - Including `wrapper/tokens/alchemist.pact` & `wrapper/tokens/skdx.pact`.
- Phase 2 targeted commit `381c9ed` then `8df27d0` of the above repository.
- Precedent Kadenaswap code at commit `e3958cd` of github.com/kadena-io/kadenaswap.
- Uniswap V2 code at commit `4dd5906` of github.com/Uniswap/v2-core.

The project methodology initially relied upon manual code review followed by dynamic experimentation within the REPL and interaction with testnet interfaces.

Limitations

Additional contract-level documentation is needed to reliably compare intended system behavior against implemented code, and making a significant investment in testing to ensure correct functionality is strongly recommended. As Chainweb/Pact is an emerging technology and the possibility of human error exists, it is advisable to get multiple independent opinions, including the implementation of a bug bounty. Nonetheless, NCC Group was able to achieve reasonable coverage of the in-scope material listed above.

Key Findings

While the in-scope code is clearly under rapid development with a variety of improvements ongoing, it appears to be thoughtfully architected and conservatively implemented. The review uncovered several findings to be resolved, including:

- **Unchecked constraints during liquidity removal** allows participants to withdraw more liquidity than they previously put in. As a consequence, they can take ownership of other users' tokens.
- **Missing checks on `time-delta` and `price-delta` prior to division** where a `price-delta` divided by a negative or excessively small `time-delta` may give a negative result or introduce arithmetic artifacts. Similarly, a negative `price-delta` calculated from a `final < initial` price scenario may result in a negative average price.

- **Weak input validation on `exchange.pact` API** where an attacker able to modify or inject malicious API traffic across a trust boundary may be able to exploit insufficiently validated input resulting in unexpected downstream behavior. Aggressive-deny input validation should be reviewed for the API of all contracts.
- **Insufficient testing strategy, methodology and implementation** requiring the development of a “broken until proven working” approach involving both per-function and user scenario tests, coverage reporting/analysis, and extensive coverage of unhappy paths. Finding `BG3` has been expanded upon in phase 2 and increased in severity.

Several additional informational findings are also documented. Extensive notes are included in the appendix titled Notes Involving Uniswap V2, Kadenaswap and several Kaddex modules.

Strategic Recommendations

NCC Group recommends addressing the findings from this engagement and prioritizing several themes during future development as follows:

- **Minimize the attack surface, minimize the functionality**: For example, some functions require the two token arguments to be provided in canonical order, while others tolerate any ordering by correcting mis-ordered function arguments internally. Remove the latter functionality and perform aggressive-deny validation. Eliminate multiple ways to do things, for example the `register-pair`, `register-pair-only`, and `add-liquidity` functions contain redundant functionality (only the latter two may be needed). Consider whether some code can be further refactored into a `utils` library, which may helpful to avoid implementing `min` 3 times.
- **Ring-fence core functionality from peripheral functionality**: Complementing the above theme, continue focusing on delineating security-critical, state-changing, self-contained core functionality from the less-critical supporting functionality, similar to the approach Uniswap V2 has taken with `core` versus `periphery`, and ensure directory and file organization reflects this.
- **Invest big in testing**: Develop test cases for every function starting with low-level helpers and working towards API-level functionality with robust coverage for both happy and unhappy paths. Implement continuous integration and system-level testing.
- **Invest in documentation**: Describe overall system intent, required behavior, correct/incorrect system operation, and tightly define valid stimulus. Testing should be able to connect requirements to implemented code, and will increase user confidence.

2 Dashboard

Finding Breakdown

Critical issues	1	🟪
High issues	1	🟥
Medium issues	1	🟧
Low issues	4	🟨🟨🟨🟨
Informational issues	7	⬜⬜⬜⬜⬜⬜⬜
Total issues	**14**	

Category Breakdown

Configuration	2	⬜⬜
Data Validation	4	🟧🟨⬜⬜
Error Reporting	1	🟥
Other	5	🟪🟨⬜⬜⬜
Patching	2	🟨🟨

🟪 Critical 🟥 High 🟧 Medium 🟨 Low ⬜ Informational

3 Table of Findings

For each finding, NCC Group uses a composite risk score that takes into account the severity of the risk, application's exposure and user population, technical difficulty of exploitation, and other factors.

Title	Status	ID	Risk
Token Heist Via Bogus Wrapper Liquidity Removals	Reported	E9F	Critical
Systemic Unexercised/Untested Functionality	Updated	BG3	High
Missing Checks on `time-delta` and `price-delta` Prior to Division	Reported	HH7	Medium
Outdated Pact Dependency	Reported	LVG	Low
Weak Input Validation on `exchange.pact` API	Reported	47B	Low
Penalty Calculation Zero Maturation Coefficient Edge Case	Reported	72V	Low
Unclear Governance May Allow Market Manipulation	New	6HY	Low
Discussion Item: Unicode Normalization	Reported	H36	Info
Unnecesary Locking Functionality Exposed	Reported	9A9	Info
Dead Code Lines for `pair-key`	Reported	A7K	Info
Unnecessary Privilege Restriction	Reported	H2F	Info
Lack of KDX Precision Flooring in Stake/Unstake Functions	Reported	YLQ	Info
Additional Staking Contract Redundancies and Observations	Reported	TAD	Info
Adding Liquidity Allows Negative Minimums	New	BXE	Info

4 Finding Details

Critical

Token Heist Via Bogus Wrapper Liquidity Removals

Overall Risk	Critical	**Finding ID**	NCC-E004218-E9F
Impact	High	**Category**	Other
Exploitability	High	**Status**	Reported

Impact

Participants on the network can withdraw more liquidity than they previously put in. As a consequence, they can take ownership of other users' tokens.

Description

The Kaddex Wrapper contract wraps the Kadenaswap AMM in order to expose additional features to investors, such as protection from IL (Impermanent Loss). The Wrapper contract is effectively a proxy contract sitting between investors and the AMM which holds investors' positions. Roughly, the workflow is as follows:

- In order to add liquidity to Kaddex, investors call the Wrapper's `add-liquidity` function, which accepts an investor's tokenA/tokenB funds and trades them for LP tokens in the Exchange contract (the Kadenaswap AMM). The minted LP tokens are held by the Wrapper and are not forwarded to the investor. The Wrapper's investor positions, their account information and each investor's holdings are described solely by the Wrapper tables/positions.
- When it is time to remove liquidity, investors call `remove-liquidity` on the Wrapper contract and the Wrapper contract surrenders a portion of the users' LP tokens it holds, in return for tokenA/tokenB liquidity. Finally, the Wrapper contract sends tokenA/tokenB funds to the user and the user positions in the Wrapper contract are updated.

An additional complication is that users can interact with the Exchange contract directly, using some of the same addresses that are used in their Wrapper positions. NCC Group consultants continue to look for issues that would stem from de-syncing these two views (Exchange's ledger and Wrapper's positions). This finding arose while searching for such data de-sync issues.

Consider the following snippets from the Wrapper's `remove-liquidity` function. The `liquidity` parameter is the LP token amount by which the investor's position is reduced. The question is: what prevents the investor from claiming an amount that is beyond their recorded position?

```
(defun remove-liquidity:object
  ( tokenA:module{fungible-v2}
    tokenB:module{fungible-v2}
    liquidity:decimal
    amountA-min:decimal
    amountB-min:decimal
    sender:string
    to:string
    to-guard:guard
    wants-kdx-rewards:bool
  )

    ;; ... SNIP ...
    ;; verify the liquidity position guard and invariants
    (install-capability (LIQUIDITY_POS_GUARD tokenA tokenB sender))
    (enforce (= tokenA token0) "sanity check: token0 is tokenA")
    (enforce (= tokenB token1) "sanity check: token1 is tokenB")
    (enforce (<= liquidity entry-total-liquidity)
      (format "remove-liquidity: Insufficient liquidity position ({} > {})" [liquidity, entry
      ↪ -total-liquidity]))
    (enforce (<= liquidity actual-total-liquidity) "sanity check: actual-total-liquidity
    ↪ covers the amount")
```

The last two highlighted lines aim to check the `liquidity` parameter passed by the investor. The first highlighted line ensures `liquidity` is less than `entry-total-liquidity`. However, this variable is defined as:

```
(let* (
;; .. SNIP ..
(entry-total-liquidity (at 'liquidity-tokens liquidity-account))
(actual-total-liquidity (tokens.get-balance pair-key liquidity-account-name))
```

The `liquidity-account` table is a generic table, unrelated to a specific user. The `entry-total-liquidity` value is an aggregated LP token amount held by all users of the Wrapper contract for that particular token pair. The second highlighted line checks that the Wrapper contract's LP token account in the Exchange actually has the funds that are requested. The `remove-liquidity` function, however, does not verify that that particular caller holds the funds. By the end of the `remove-liquidity` function, the investor's positions are decreased, but there is no check that this value stays positive. As such, an investor should be able to withdraw funds corresponding to other users' LP tokens.

Recommendation

Discussions with the Kaddex Team suggested the following constraints will be enforced:

```
0 < liquidity <= liquidity-position-share
0 < withdrawal-fraction <= 1.0
```

High

Systemic Unexercised/Untested Functionality

Overall Risk	High	**Finding ID**	NCC-E004218-BG3
Impact	High	**Category**	Error Reporting
Exploitability	Undetermined	**Status**	Updated

Impact

Avoidable bugs may result in the loss of user funds. Missing tests cannot help precisely specify intended (and unintended) functionality. Insufficient test coverage will reduce the agility and reliability of refactoring efforts. Poor testing will decrease the confidence of potential users.

Description

Safety and security oriented software must be considered broken until proven working. This requires a robust testing approach involving low-level per-function (unit) testing as well as higher-level user scenario (integration) testing. Both ‘happy’ paths that represent correct functional flow and ‘unhappy’ paths that represent incorrect input or exceptional circumstances should be exhaustively tested. A separate ‘golden model’ for the calculation of fees and rewards should generate self-checking test cases for the contract code (similar to what is currently done for `simulator.repl`).

Specific examples include:

- There appear to be specific constraints around token precision that are not tested.
- Users may interact with `wrapper.pact` and/or `exchange.pact` in some circumstances. This interaction is not tested.
- The KDX rewards and boosting process are not tested.
- The `withdraw-claim` function within `wrapper.pact` is wholly unused/unexercised. The `process-claim-request-if-necessary` function within `wrapper.pact` is referenced in `wrapper.repl` but currently commented out. Thus, neither function is tested.
- The `simulator.repl` test is minimalistic and does not consider extreme ratios, multiple pairs, or more than two users.
- There is (approximately) no per-function testing.
- There is very minimal testing of ‘unhappy’ paths.
- Code upgrade/update paths are not tested.

Many of the above functions involve potentially critical steps in the expected system usage flow and should be exhaustively tested. With minimal documentation, matching intended-versus-implemented functionality is error- prone. It is highly likely that testing would have caught finding "Token Heist Via Bogus Wrapper Liquidity Removals".

Recommendation

- Implement low-level per-function tests for the ‘happy’ path and all conceivable ‘unhappy’ paths.
- Implement scenario-level testing both individually and as a put-together series.
- Implement a golden model for value/fee/rewards calculation and generate self-testing cases.
- Measure statement and path coverage (by hand if necessary)

Medium

Missing Checks on `time-delta` and `price-delta` Prior to Division

Overall Risk	Medium	**Finding ID**	NCC-E004218-HH7
Impact	Medium	**Category**	Data Validation
Exploitability	Low	**Status**	Reported

Impact

A price-delta divided by a negative or excessively small time-delta may give a negative result or introduce arithmetic artifacts. Similarly, a negative price-delta may result in a negative average price.

Description

The `get-average-price` function, implemented in `wrapper.pact` as shown below, returns a result critical to the correct operation of the system.

```
1064  (defun get-average-price:decimal
1065      ( initial:object{exchange.observation}
1066        final:object{exchange.observation}
1067      )
1068      "Utility function for calculating the TWAP between two oracle observations performed."
1069      (let*
1070        ( (time-delta (diff-time (at 'timestamp final) (at 'timestamp initial)))
1071          (price-delta (- (at 'price final) (at 'price initial)))
1072        )
1073        (exchange.truncate (get-base-token) (/ price-delta time-delta))
1074      )
1075    )
```

There is no check on the whether the `time-delta` is negative or an unreasonably small value prior to the division operation highlighted above. If the initial timestamp matches the final timestamp, the division should cause the transaction to fail (as desired). If the initial timestamp were larger than the final timestamp for some reason, the average price returned would be negative. If the initial timestamp were exceedingly close to the final timestamp, unexpected rounding effects may be introduced.

A similar situation involves `price-delta` where a final price that is smaller than the initial price will result in a negative numerator. The function will then give a negative result.

Note that an arguably-similar non-zero check is performed within the `UniswapV2Pair.sol` contract.[1]

Recommendation

Validate (enforce) that the `time-delta` is larger than a reasonable minimum based on normal operating conditions. The `price-delta` should be an absolute value of the price difference.

Location

Near line 1073 of wrapper/wrapper.pact

1. https://github.com/Uniswap/v2-core/blob/4dd59067c76dea4a0e8e4bfdda41877a6b16dedc/contracts/UniswapV2Pair.sol#L77

Low

Outdated Pact Dependency

Overall Risk	Low	**Finding ID**	NCC-E004218-LVG
Impact	Medium	**Category**	Patching
Exploitability	Undetermined	**Status**	Reported

Impact

Testing the AMM behavior against an outdated Pact version may not surface differences stemming from the much more recent Pact version currently deployed on Kadena Chainweb nodes.

Description

The `exchange.pact` source file begins by specifying a minimum version of Pact as shown below.

```
13 (enforce-pact-version "3.7")
14
15 (namespace (read-msg 'ns))
16
17 (module exchange GOVERNANCE
18 ...
```

Pact version 3.7 was first released in December 2020[2]. The version of Pact deployed on the current Kadena Chainweb node is 4.2.0.1,[3] which is significantly newer.

Pact version 4.0 (a new major version) and onwards includes a large number of fixes and enhancements that can impact contract behavior. Relevant changes may pertain to gas consumption, changes in arithmetic precision, and new native operations. Should the code already take advantage of these, it may no longer be runnable on Pact version 3.7 as currently specified.

Recommendation

Adapt the code to enforce a minimum version of 4.2. Ensure all dependencies and tools are updated to the latest specific versions[4] recommended for production deployment and consistent with deployed Kadena Chainweb nodes. Add a periodic gating milestone to the development process that involves reviewing all dependencies for outdated or vulnerable versions.

Location

Line 13 of wrapper-contract/kadenaswap/exchange.pact

2. https://github.com/kadena-io/pact/releases/tag/v3.7.0
3. https://github.com/kadena-io/chainweb-node/blob/328785eb7e194f9707ec60b0af31c889aac93afe/chainweb.cabal#L374
4. https://cabal.readthedocs.io/en/3.4/cabal-project.html#cfg-flag---reject-unconstrained-dependencies

Low

Weak Input Validation on `exchange.pact` API

Overall Risk	Low	**Finding ID**	NCC-E004218-47B
Impact	Medium	**Category**	Data Validation
Exploitability	Low	**Status**	Reported

Impact

An attacker able to modify or inject malicious API traffic across a trust boundary may be able to exploit insufficiently validated input resulting in unexpected downstream behavior.

Description

The `add-liquidity` function implemented in `exchange.pact` is an API[5] that may be considered a trust boundary. This motivates the additional property-level checks implemented near line 31. As shown below, validation of 'unit precision' is performed on input amounts per lines 535 and 536. However, the initially received decimal amounts are not validated to be positive values (hence zero or negative values may enter the logic).

```
522 (defun add-liquidity:object
523   ( tokenA:module{fungible-v2}
524     tokenB:module{fungible-v2}
525     amountADesired:decimal
526     amountBDesired:decimal
527     amountAMin:decimal
528     amountBMin:decimal
529     sender:string
530     to:string
531     to-guard:guard
532   )
533   "Adds liquidity to an existing pair. The `to` account specified will receive the..."
534   (enforce-contract-unlocked)
535   (tokenA::enforce-unit amountADesired) ;; enforce the informed amounts are in the corr...
536   (tokenB::enforce-unit amountBDesired)
537   (with-capability (MUTEX) ;; obtain the mutex lock
538     (obtain-pair-lock (get-pair-key tokenA tokenB)))
539   (let*
540     ( (p (get-pair tokenA tokenB))
541       (reserveA (reserve-for p tokenA))
542       (reserveB (reserve-for p tokenB))
543       ;; calculate the actual amounts of liquidity that will be added to keep the rese...
544       (amounts ...
```

The API function currently (initially) accepts zero and negative values, and proceeds to perform calculations on them starting with line 544. While downstream code may currently happen to handle this correctly, best practices suggest early and aggressive input validation – particularly to protect against future code/logic updates.

Similarly, the `liquidity` parameter[6] of the `remove-liquidity` function implemented in the same source file would benefit from an additional check.

5. https://github.com/kaddex-org/wrapper-contract/blob/204b93e2bf845b680f4e7bcfd24ca88e07532ad4/kadenaswap/exchange.pact#L8
6. https://github.com/kaddex-org/wrapper-contract/blob/204b93e2bf845b680f4e7bcfd24ca88e07532ad4/kadenaswap/exchange.pact#L649

Recommendation

In addition to precision and other checks, ensure input amounts are positive and reject otherwise.

Location

Near line 535 and 649 of wrapper-contract/kadenaswap/exchange.pact

Low

Penalty Calculation Zero Maturation Coefficient Edge Case

Overall Risk	Low	**Finding ID**	NCC-E004218-72V
Impact	Undetermined	**Category**	Other
Exploitability	Undetermined	**Status**	Reported

Impact

In the case when the maturation coefficient is set to zero, the penalty calculation function may exhibit behavior different than if the maturation coefficient were non-zero. This may result result in logical issues down the line.

Description

Consider the staking module's function that penalizes stakers who claim their rewards before the maturation period ends:

```
(defun calculate-penalty:decimal
   (seconds:decimal
    amount:decimal)
   "Given a reward amount and seconds passed since effective-start, calculate \
   \ reward penalty. Applies the reward penalty curve as described above \
   \ defconst MATURATION_COEFFICIENT."
   (enforce (>= seconds 0.0) ;; Enforce that some time has passed from effective-start.
     (format "Violation of causality ({} seconds staked)" [seconds]))
   (if (> seconds MATURATION_PERIOD) 0.0 ;; No penalty after maturation.
     (if (= MATURATION_COEFFICIENT 0.0) amount ;; No penalties if MATURATION_COEFFICIENT is 0.
       (let*
         ( (maturation-fraction (/ seconds MATURATION_PERIOD))
           (penalty-base (- 1.0 maturation-fraction))
           (penalty-exp (^ penalty-base MATURATION_COEFFICIENT)))
         (floor (* amount penalty-exp) (kdx.precision))))))
```

In the first highlighted line, if the maturation period has passed, the function returns zero, meaning that no penalty is applied. The second highlighted line concerns the hard-coded `MATURATION_COEFFICIENT` parameter, which determines the shape of the penalty curve and is currently set to 0.66. If this coefficient is zero, the comment indicates that no penalties should be applied. In that case, regardless of the passed number of seconds, the penalty calculation formula would return the full amount, meaning that the staker would always be penalized with the full amount. After discussing with the Kaddex Team, it appears that the error is in the code and not in the comment.

In addition, if the maturation coefficient is zero (the second highlighted line), `calculation-penalty` does not floor the result up to `kdx.precision`. The rounding, however, does happen in the last line of the snippet. This appears to be an inconsistency, as if the `amount` input is not rounded and the maturation coefficient is zero, the function output will not be rounded either.

Recommendation

If the logic that handles a zero maturation coefficient is for debugging purposes, consider removing it to simplify the code. If in practice the coefficient can be zero, correct either the code or the comment. Finally, in that case, it may make sense to round the amount before returning the amount.

Low

Unclear Governance May Allow Market Manipulation

Overall Risk	Low	**Finding ID**	NCC-E004218-6HY
Impact	Medium	**Category**	Patching
Exploitability	Undetermined	**Status**	New

Impact

Privileged parties may be able to modify configuration parameters or critical business logic in the contracts for unfair advantage.

Description

The Uniswap V2 contracts deployed on Ethereum are essentially stand-alone and immutable[7]. As a result, participants are protected from post-deployment changes that may be to their disadvantage. Proposals for future enhancements or changes are incorporated into a separate and distinct set of next-generation contracts (e.g., Uniswap V3).

However, the Kadena Chainweb/Pact system provides a fundamentally different mechanism[8] from Ethereum for managing contract upgrades. Both the `wrapper.pact` and `exchange.pact`, and various token contracts, support a governance model that boils down to a `keyset-ref-guard 'kaddex-ns-admin` check – in other words, administrative keys.

Centralizing authority over contracts in this way nullifies many of the benefits of deploying the contract to a decentralized network, and may make the system less attractive to participants. Participants must trust Kaddex to be both safe against attackers who may take control of administrative accounts and trustworthy enough to not take advantage of their powers.

Additionally, the contracts support `OPS` privileges[9] that are used by external off-chain systems (within Kaddex) which are necessary for correct system functionality. This presents another component that participants must trust.

Recommendation

This is a low-severity finding to surface concern. Articulate the governance[10] process for bug-fixes, the steps taken to protect high-value keys, and the commitments behind external off-chain systems.

Location

The `wrapper.pact`, `exchange.pact`, and the various token contracts.

7. https://github.com/Uniswap/v2-core/blob/master/contracts/UniswapV2Pair.sol
8. https://medium.com/kadena-io/pact-solving-smart-contract-governance-and-upgradeability-976aac3bbb31
9. https://github.com/kaddex-org/wrapper-contract/blob/main/wrapper/wrapper.pact#L38
10. https://docs.kaddex.com/kaddex-features/governance/proposals-and-voting-power-calculation#governance

Info

Discussion Item: Unicode Normalization

Overall Risk	Informational	**Finding ID**	NCC-E004218-H36
Impact	Medium	**Category**	Data Validation
Exploitability	Undetermined	**Status**	Reported

Impact

This (currently) strictly-informational finding is intended to provide documentation for an early item of discussion.

Different Unicode encodings of the same string identifier may lead to spoofing attacks based on user misunderstandings, as well as interoperability problems potentially impacting consensus between interacting implementations/components.

Description

When entering the same visual/logical string into a UI, differently encoded Unicode strings may arise from malicious intent or simply the broad range of participating devices, operating systems, locales, languages and applications involved. Endpoints handling string identifiers, such as coin names, user names or secrets, should validate and/or normalize them before use to A) ensure consistent handling and B) maximize interoperability.

Divergent string encoding typically involves characters with accents or other modifiers that have multiple correct Unicode encodings. For example, the Á (a-acute) glyph can be encoded as a single character U+00C1 (the "composed" form) or as two separate characters U+0041 then U+0301 (the "decomposed" form). In some cases, the order of a glyph's combining elements is significant and in other cases different orders must be considered equivalent[11]. At the extreme, the Unicode character U+FDFA can be correctly encoded as either a single code point or a sequence of up to 18 code points[12]. An identifier may appear visually identical but in fact be physically distinct, such as "Álpha Token" and "Álpha Token". Normalization is the common[13][14][15] process of standardizing string representation such that if two strings are canonically equivalent and are normalized to the same normal form, their byte representations will be the same. Only then can string comparison, ordering, deduplication and cryptographic operations be relied upon.

There are a variety of Normalization methods available[16] with NFKC being most appropriate[17] in this circumstance (although note that BIP-39 uses NFKD[18]). Similarly, section 5.1.1.2 of the NIST Special Publication 800-63B Digital Identity Guidelines[19] document gives guidance related to the use of Unicode in Memorized Secret Verifiers as follows:

> If Unicode characters are accepted in memorized secrets, the verifier SHOULD apply the Normalization Process for Stabilized Strings using either the NFKC or NFKD normalization defined in Section 12.1 of Unicode Standard Annex 15. This

11. http://unicode.org/reports/tr15/tr15-22.html
12. https://www.compart.com/en/unicode/U+FDFA
13. https://docs.oracle.com/javase/tutorial/i18n/text/normalizerapi.html
14. https://blog.golang.org/normalization
15. https://docs.rs/unicode-normalization/0.1.13/unicode_normalization/
16. http://unicode.org/reports/tr15/#Norm_Forms
17. See question 2 of https://unicode.org/faq/normalization.html
18. https://en.bitcoin.it/wiki/BIP_0039
19. https://pages.nist.gov/800-63-3/sp800-63b.html

> process is applied before hashing the byte string representing the memorized secret. Subscribers choosing memorized secrets containing Unicode characters SHOULD be advised that some characters may be represented differently by some endpoints, which can affect their ability to authenticate successfully.

Best practices require Unicode normalization to prevent the proliferation of visually and logically similar strings with different encodings. This can be considered a form of input validation.

Recommendation

This topic requires more investigation within the specific Kaddex context before a firm recommendation can be made. Generally, it is recommended to perform NKFC Unicode normalization on all string identifiers immediately upon receipt using functionality such as that found in the 'unicode-transforms' package[20].

20. https://hackage.haskell.org/package/unicode-transforms

Info

Unnecesary Locking Functionality Exposed

Overall Risk	Informational	**Finding ID**	NCC-E004218-9A9
Impact	Low	**Category**	Other
Exploitability	None	**Status**	Reported

Impact

The staking contract exposes functionality that allows operators to lock an arbitrary user's funds for unlimited amounts of time. This may undermine public confidence in the staking module.

Description

The following function allows administrators to add arbitrary locks to chosen accounts:

```
(defun lock-stake (account:string amount:decimal until:time)
  "Operator-only function to add a stake lock to a given account."
  (with-capability (OPS)
  (with-read stake-table account
    { 'locks := locks }
    (update stake-table account { 'locks: (+ locks  ;; add to locks, which is a list of
    ↳ stake-locks
      [ { 'amount: amount, 'until: until } ]) }))))
```

As per discussions with the Kaddex team, the functionality was implemented to add vesting schedules for early token holders. There is no planned use case and, at the same time, it appears to give operators a degree of control that is unnecessary.

Recommendation

Since there is no planned usage for `lock-stake` and `onboard-with-lock` functions, they may be deleted/modified.

Info

Dead Code Lines for `pair-key`

Overall Risk	Informational	**Finding ID**	NCC-E004218-A7K
Impact	None	**Category**	Configuration
Exploitability	None	**Status**	Reported

Impact
Dead code may reflect an unfinished implementation or incomplete refactoring.

Description
The `let` statement defining `pair-key` on lines 524[21] and 765[22] of `wrapper.pact` is unecessary. This value is not subsequently used within the enclosing function.

Recommendation
Remove the unnecessary lines involving `pair-key`.

Location
wrapper-contract/wrapper/wrapper.pact

21. https://github.com/kaddex-org/wrapper-contract/blob/204b93e2bf845b680f4e7bcfd24ca88e07532ad4/wrapper/wrapper.pact#L524
22. https://github.com/kaddex-org/wrapper-contract/blob/204b93e2bf845b680f4e7bcfd24ca88e07532ad4/wrapper/wrapper.pact#L765

Info

Unnecessary Privilege Restriction

Overall Risk	Informational	**Finding ID**	NCC-E004218-H2F
Impact	None	**Category**	Configuration
Exploitability	High	**Status**	Reported

Impact

An expectation that retrieving all pending requests requires the OPS keyset is easily bypassed.

Description

The `get-all-pending-requests` and `get-user-pending-requests` functions as implemented in `wrapper.pact` are shown below. For the former function, line 1245 places the read logic behind a `with-capability` clause[23] which effectively requires the `'kaddex-ops-keyset` keyset[24]. Thus, the ability to get all pending requests could be expected to be constrained to the OPS user.

```
1243  ;; TODO: not sure which of the below functions we will want to use for the data to
      ↪ display on the frontend
1244  (defun get-all-pending-requests:[string] ()
1245    (with-capability (OPS) (at 'requests (read pending-requests ALL_PENDING_REQUESTS_KEY))))
1246
1247  (defun get-user-pending-requests:[string]
1248    ( account:string )
1249    (at 'requests (read pending-requests account)))
```

However, line 135 defines the constant used above as the empty string.[25]

```
(defconst ALL_PENDING_REQUESTS_KEY "")
```

As a result, if any user calls the `get-user-pending-requests` function with an empty string, the code will return **all** pending requests without requiring the OPS keyset. This was confirmed via Shadena with the transaction `(kaddex.wrapper.get-user-pending-requests "")`.

Note that the comment on line 1243 suggests these functions may be under review.

Recommendation

Consider whether the functionality can be simplified to target only the latter function.

23. https://pact-language.readthedocs.io/en/stable/pact-functions.html#with-capability
24. https://github.com/kaddex-org/wrapper-contract/blob/204b93e2bf845b680f4e7bcfd24ca88e07532ad4/wrapper/wrapper.pact#L167
25. https://github.com/kaddex-org/wrapper-contract/blob/204b93e2bf845b680f4e7bcfd24ca88e07532ad4/wrapper/wrapper.pact#L135

Info

Lack of KDX Precision Flooring in Stake/ Unstake Functions

Overall Risk	Informational	**Finding ID**	NCC-E004218-YLQ
Impact	Low	**Category**	Other
Exploitability	None	**Status**	Reported

Impact

External systems' calls to `stake` and `unstake` may fail due to precision issues.

Description

Kaddex contracts juggle with different decimal number precision specifications. On the one hand, there is the native Pact precision (up to 256 decimal digits) inside the Kaddex contracts themselves. In addition, there are wrapped/fungible token precision specifications. All of the relevant fungible token contracts in the codebase hard-code their precision to 12 digits, however, it is easily imaginable that other future wrapped tokens will have different precisions. Finally, while addition and subtraction preserve precision, operations such as multiplication and division do not. As such, as amounts travel between different fungible tokens and Kaddex contracts, rounding needs to be carefully applied. The most obvious consequence of an oversight in this regard is that a fungible token would reject a particular transfer, potentially resulting in a stall in the staking contract operation.

As an informational illustration, consider a scenario where Kaddex onboards a fungible token with precision different than the otherwise uniform precision in other tokens. This is allowed as per the fungible token interface, as each particular token specifies their own precision. When the `sweep-one` function is processing the token pair that involves that particular token, the `remove-liquidity` function is called. The token amount that is sent to `pair-account` and, subsequently, to the shared token amount, is expressed in that token's original precision. The same holds for `sweep-one`'s `amount0` and `amount1` values. The `swap-to-kdx` function needs to truncate the output amount during the conversion and this is correctly done in the exchange contract. NCC Group consultants validated other token travel trails and did not identify a precision inconsistency.

It is conceivable that external automated systems may access the staking and unstaking interface. A problem could arise if these external systems use number systems with precision different than 12 digits, such as double precision floats. In such a case, a call to the staking/unstaking interface would fail since the KDX contract would not be able to enforce the minimal precision (the `enforce-unit` function).

Recommendation

One may make an argument for rounding up to `kdx.precision` before wrapping/ unwrapping actions in staking/unstaking operations. This could make the code more tolerant to external systems not strictly following the specification.

Info

Additional Staking Contract Redundancies and Observations

Overall Risk	Informational	**Finding ID**	NCC-E004218-TAD
Impact	None	**Category**	Other
Exploitability	None	**Status**	Reported

Impact

This is a purely informational finding which contains observations on how the staking contract code could be improved.

Description

Rollup function unused variables

```
(defun rollup:decimal (account:string)
   "Realize a given account's deserved reward amount into the rollover field of \
   \ their stake record, resetting start-cumulative to the current revenue-per-kdx \
   \ number. Should be done before any unstake or reward claim."
   (with-capability (ROLLUP account)
   (with-read stake-table account
     { 'amount := amount
     , 'rollover := rollover
     , 'effective-start := effective-start
     , 'start-cumulative := start-cumulative }
     (let*
       ( (now (at 'block-time (chain-data)))
         (seconds (diff-time now effective-start))
```

The `seconds` and `now` variables computed inside the `let*` statement are not used by the `rollup` function. The new rollover is computed using the old cumulative and the amount (i.e. it does not depend on the time). It is assumed this is a vestige from previous code versions.

Subtraction by zero in `swap-to-kdx`

The final amount in `swap-to-kdx` is extracted as the last element in the array returned by `exchange.swap-exact-in`:

```
         (at 'amount (at (- (length path) 0) ;; Extract amount out from swap result
          (exchange.swap-exact-in
            amount-in 0.0 ;; TODO: Supply a minimum amount out
            (unroll-path token-in path) from to to-guard))))))
```

The subtraction by zero can be removed, as it does not change the index that is being accessed.

`sweep-some` redundancies

The `sweep-some` function sweeps fees for a list of token pairs. For each pair, the LP token fees are first converted to tokens and then to KDX. The `total-out` in the second line in the snippet below is a sum of all KDX earnings.

```
            (drain-result (map (drain-account) accounts))
            (total-out (fold (+) 0.0 drain-result))
            ;; Enforce that the KDX balance of KDX_BANK after the swapping is
            ;; equal to the KDX balance before + the swap outputs.
            (balance-after (kdx.get-balance KDX_BANK))
            (effective-out (- balance-after balance-before)))
          (enforce (= total-out effective-out) (format "Accounting mismatch (expected {} out,
          ↪ received {})" [total-out effective-out]))
          (if (= (floor total-out (kdx.precision)) 0.0)
```

The `total-out` cannot have more granular precision than `kdx.precision` and as such the flooring in the last line appears redundant. Similarly, the first highlighted line appears to enforce an invariant which has to hold. The line can be considered a debug statement/ invariant as it is not clear how it can be violated and as such may be up for deletion.

`read-waiting` implementation issue

The `read-waiting` function aims to return stakers that have pending stake:

```
## `read-waiting`

The method appears to aim to return those stakers that have pending stake:

~~~lisp
  (defun read-waiting:[string] ()
    "Utility function to read the list of stakers waiting to be added to the pool. \
    \ In production, will only be called on /local due to gas costs."
    (map (at 'account) (select stake-table (where 'pending-add (< 0.0))))
  )
```

It is unclear how `pending-add` can become negative, which makes this a possible oversight.

`revenue-per-kdx` can spike during low stake levels

The `sweep-some` function updates the `revenue-per-kdx` variable as follows:

```
              (update state-table STATE_KEY
                { 'revenue-per-kdx: (+ prev-revenue (/ total-out staked-kdx)) })
              total-out)))))))
```

The `total-out` and `staked-kdx` values are independent. In a single-staker scenario (e.g. right at staking contract creation or during other unforeseen irregular conditions), the staker can make their stake as small as the KDX precision allows. This will result in the `revenue-per-kdx` growing to a huge number. NCC Group consultants have not identified a way to exploit this to gain advantage. Additionally, this condition could happen only in irregular low stake conditions and, as such, does not appear to represent a threat.

Recommendation

Consider fixing the first four items in the finding. For the fifth item, unless the `revenue-per-kdx` behavior is clearly unintended from the Kaddex team's perspective, it does not appear that it warrants any fix.

Info

Adding Liquidity Allows Negative Minimums

Overall Risk	Informational	**Finding ID**	NCC-E004218-BXE
Impact	None	**Category**	Data Validation
Exploitability	None	**Status**	New

Impact

Missing input validation may allow mistaken or malicious values to interfere with downstream logic resulting in undesirable behavior.

Description

The `exchange.add-liquidity` and `wrapper.add-liquidity` functions both have essentially the same signature, defined as follows, where the 5th and 6th parameters represent an acceptable minimum amount of tokens to be consumed.

```
(defun add-liquidity:object
    ( tokenA:module{fungible-v2}
      tokenB:module{fungible-v2}
      amountA-desired:decimal
      amountB-desired:decimal
      amountA-min:decimal
      amountB-min:decimal
      sender:string
      to:string
      to-guard:guard
    )
```

Experiments indicate that calling these functions with negative minimums is successful. A user may misunderstand amount conventions and mistakenly provide a negative number that is not respected. Further, allowing invalid data to enter the system may increase the risk to downstream logic.

Recommendation

Note that this finding is marked informational as (effectively) ignoring negative values could be considered correct. Consider tightening the input validation checks to disallow negative (and potentially zero) values as these could never be in the user's best interest.

5 Notes Involving Uniswap V2, Kadenaswap and several Kaddex modules

This informal section contains notes and observations generated during the project. While all security issues have been reported in the preceding findings, the content below is relevant for discerning intended functionality, the continutity of phase 1 → phase 2 understanding, and for stimulating interim discussion topics. The content is not intended to be comprehensive.

1 – Scope

The primary materials used for phase 1 of the review include the following:

- The target Kaddex code within commit `204b93e` of github.com/kaddex-org/wrapper-contract, with primary functional paths involving:
  - `kadenaswap/exchange.pact` – core module for the exchange.
  - `kadenaswap/tokens.pact` – liquidity tokens used by exchange.
  - `kadenaswap/gas-station/gas-station.pact` – creates and manages the gas station.
  - `wrapper/wrapper.pact` – mainly provides for: A) boosted KDX rewards for liquidity provider fees, B) adding liquidity with a single side of the pair, and C) utility functions for the front-end to query LP stats.
  - `wrapper/tokens/kdx.pact` – KDX token module, a modified fungible-v2 token.
  - `staking/staking.pact` – manages exchange fees, tracks of users, and distributes rewards.
    - Including `wrapper/tokens/alchemist.pact & wrapper/tokens/skdx.pact`
- Precedent Kadenaswap code at commit `e3958cd` of github.com/kadena-io/kadenaswap.
- Uniswap V2 code at commit `4dd5906` of github.com/Uniswap/v2-core.
- The Pact Smart-Contract Language (whitepaper).
- The Pact Language Reference at https://pact-language.readthedocs.io/en/stable/.
- The above code built and dynamically exercised through the REPL and local server interfaces.

The above list was updated as the project progressed. Note that front-end code is not currently in scope. The project methodology primarily relies upon manual code review supported by dynamic interaction with testnet interfaces and REPL test cases. Note that the specific commits listed above are generally the latest available.

2 – General Observations

- The code contains a significant number of `TODO` s which all seem sensible and are assumed to be in plan. The recently added code comments have been helpful.
- The overall contract(s) re-entrant lock/mutex may be unnecessary as Pact precludes reentrancy. However, this could be considered a defense-in-depth risk mitigation for a well-known and high-impact exploit vector (as Pact's guarantee pertains to the same code executed twice rather than a different portion inadvertently executed).
- In some places input validation is via 'aggressive deny', such as when tokens are required to be supplied in canonical order. However, in other places the code attempts to 'fix and continue' such as when token mis-ordering is detected then corrected. The latter more tolerant approach adds unnecessary code complexity and risk.
- There is some redundancy in helper functions, such as `max` implemented multiple times. The contracts should be restructured into their simplest most-DRY form.

2.1 - Uniswap V2 Notes

Uniswap V2 aims to create an exchange market between pairs of tokens. There are three primary types of participants: the operator, liquidity providers, and token traders.

The operator has very limited presence. Consider this to be the original developers, their governance, the overall deployment source, ecosystem and a potential sink of a 0.05% trading fee in the future.

The liquidity providers supply the pair of tokens to an automated, pair-specific, exchange contract in return for part ownership in the form of a (third) liquidity token. The purpose of this contract is to allow each token of the pair to be traded for the other token at a floating rate and for a fee. At any time, liquidity providers can redeem their liquidity token to reclaim their portion of the pair of tokens at a floating rate along with accumulated fees.

Traders send one type of token to the above contract and receive the other, with the floating exchange rate based on the relative supply of each token in the pair held.

The contracts are split into a core group and a periphery group. The former group actually holds the assets, is intentionally minimalistic in functionality, and is security-critical. The periphery group is intended to provide additional convenience functionality and is not security-critical. Only the former core group is described below. Furthermore, the intriguing `uint112` type and arithmetic operators are not described here as they will not apply to Kaddex.

3 – The `UniswapV2Factory` Contract

The `UniswapV2Factory.sol` factory contract is responsible for creating unique pair-specific exchange/trading contracts, and is also able to turn on the 0.05% protocol charge (and set the destination). It is deployed and operated by the 'operator' participant.

Note that the Uniswap V2 documentation[26] has misalignments with the latest contract code commit. Specifically, the `getPair()` and `allPairs()` functions are missing and the two fee-oriented functions are now prefixed with `set` in the code (and are state changing rather than `view` only).[27]

3.1 – Contract State Variables, Events, and Constructor

The constructor below is provided with an unvalidated `feeToSetter` address when called during deployment, and all other state variables are initialized empty.

```
address public feeTo;
address public feeToSetter;

mapping(address => mapping(address => address)) public getPair;
address[] public allPairs;

event PairCreated(address indexed token0, address indexed token1, address pair, uint);
```

26. https://docs.uniswap.org/protocol/V2/reference/smart-contracts/factory
27. https://github.com/Uniswap/v2-core/blob/4dd59067c76dea4a0e8e4bfdda41877a6b16dedc/contracts/UniswapV2Factory.sol

```

constructor(address _feeToSetter) public {
    feeToSetter = _feeToSetter;
}
```

As will be seen in the functionality below, the `feeToSetter` address is the only participant allowed to change the `feeTo` and `feeToSetter` state variables. In a sense, this is similar to the contract owner. If/when the `feeTo` is set (and is non-zero), that address state variable indicates the recipient of fees. If a mistake is made when setting the `feeToSetter` address state variable, it will be unrecoverable.

A multi-level mapping is created: consider this as a pair of ERC-20 contract addresses that index into an array element containing a pair-specific exchange contract address. A list of all covered pairs is also initialized empty, and an event defined for the deployment of pair-specific contracts.

The `PairCreated` event is emitted each time a pair is created via `createPair()` where A) `token0` is guaranteed to be strictly less than `token1` by sort order, and B) the final `uint` log value reflects the cumulative number of pairs deployed.

3.2 – The `allPairsLength()` (helper) Function

The `allPairsLength()` function returns the number of pair-specific contracts currently covered.

```
function allPairsLength() external view returns (uint) {
    return allPairs.length;
}
```

This external, read-only and publicly-accessible function should match the last integer on the `PairCreated` event. It should be much cheaper than iterating over the `getPair` mapping.

3.3 – The `createPair()` Function

The `createPair()` function deploys a pair-specific `UniswapV2Pair.sol` contract to initiate a pair-specific exchange. This is a central function that all downstream pair-specific activity depends upon.

```
function createPair(address tokenA, address tokenB) external returns (address pair) {
    require(tokenA != tokenB, 'UniswapV2: IDENTICAL_ADDRESSES');
    (address token0, address token1) = tokenA < tokenB ? (tokenA, tokenB) : (tokenB,
    ↪ tokenA);
    require(token0 != address(0), 'UniswapV2: ZERO_ADDRESS');
    require(getPair[token0][token1] == address(0), 'UniswapV2: PAIR_EXISTS'); // single
    ↪ check is sufficient
    bytes memory bytecode = type(UniswapV2Pair).creationCode;
    bytes32 salt = keccak256(abi.encodePacked(token0, token1));
    assembly {
        pair := create2(0, add(bytecode, 32), mload(bytecode), salt)
    }
    IUniswapV2Pair(pair).initialize(token0, token1);
    getPair[token0][token1] = pair;
    getPair[token1][token0] = pair; // populate mapping in the reverse direction
    allPairs.push(pair);
    emit PairCreated(token0, token1, pair, allPairs.length);
}
```

This external, state-changing and publicly-accessible function is provided with addresses for tokenA and tokenB. The code ensures the addresses are not identical, adjusts the sorted ordering if necessary, checks that the address are non-zero and that the pair has not already been seen. The code then hashes the token pair for salt, utilizes the `create2` assembly code to create, deploy and initialize the pair-specific exchange contract. Finally, both directions of the `getPair` mapping are populated, the pair is pushed onto the `allPairs` list and an event is emitted.

3.4 – The `setFeeTo()` Function

The `setFeeTo()` function is the gatekeeper/setter of the value of the `feeTo` address state variable.

```
function setFeeTo(address _feeTo) external {
    require(msg.sender == feeToSetter, 'UniswapV2: FORBIDDEN');
    feeTo = _feeTo;
}
```

As seen above, only the `feeToSetter` address is able to successfully utilize this external state-changing function. When called, it is provided with the `feeTo` address which represents the destination of the 0.05% trading fee. When this is set (or is initialized) to zero, the fee is considered to be off. See Protocol Charge Calculation.

Note that the Uniswap V2 documentation indicates this is a non state-changing `view` function and uses a slightly different name (without the `set` prefix).

3.5 – The `setFeeToSetter()` Function

The `setFeeToSetter()` function is the gatekeeper/setter to the value of `feeToSetter` address state variable.

```
function setFeeToSetter(address _feeToSetter) external {
    require(msg.sender == feeToSetter, 'UniswapV2: FORBIDDEN');
    feeToSetter = _feeToSetter;
}
```

As seen above, only the `feeToSetter` address is able to successfully utilize this external state-changing function. When called, it is provided with a new `feeToSetter` address which effectively represents the address of the 'new operator'. See Protocol Charge Calculation.

Note that the Uniswap V2 documentation indicates this is a non state-changing `view` function and uses a slightly different name (without the `set` prefix).

4 – The `UniswapV2Pair` Contract

The `UniswapV2Pair.sol` contract is created by the factory contract described above, and implements a pair-specific token exchange to provide the primary functionality to both liquidity providers and traders. There are two functional responsibilities: A) automated market trading, and B) keeping track of token balances. This contract also exposes data which can be used to build decentralized price oracles. Note that this contract inherits from `UniswapV2ERC20.sol`, which provides the the ERC-20 functions for the liquidity tokens.

4.1 – Contract State Variables, Events, and Constructor (inherits from `UniswapV2ERC20`)

The no-argument constructor below is called during deployment and simply sets the `factory` address state variable to the message sender. All other state variables are initialized empty.

```
uint public constant MINIMUM_LIQUIDITY = 10**3;
bytes4 private constant SELECTOR = bytes4(keccak256(bytes('transfer(address,uint256)')));

address public factory;
address public token0;
address public token1;

uint112 private reserve0;           // uses single storage slot, accessible via getReserves
uint112 private reserve1;           // uses single storage slot, accessible via getReserves
uint32  private blockTimestampLast; // uses single storage slot, accessible via getReserves

uint public price0CumulativeLast;
uint public price1CumulativeLast;
uint public kLast; // reserve0 * reserve1, as of immediately after the most recent
↪ liquidity event

uint private unlocked = 1;
modifier lock() {
    require(unlocked == 1, 'UniswapV2: LOCKED');
    unlocked = 0;
    _;
    unlocked = 1;
}
...
event Mint(address indexed sender, uint amount0, uint amount1);
event Burn(address indexed sender, uint amount0, uint amount1, address indexed to);
event Swap(address indexed sender, uint amount0In, uint amount1In, uint amount0Out, uint
↪ amount1Out, address indexed to);
event Sync(uint112 reserve0, uint112 reserve1);

constructor() public {
    factory = msg.sender;
}
```

The MINIMUM_LIQUIDITY constant represents a minimum number of liquidity tokens that always exist (owned by account zero) that helps avoid cases of division by zero. The SELECTOR constant represents the ABI selector for the ERC-20 transfer function which is used to transfer ERC-20 tokens.

The `factory` address points to the factory contract that created this pool, while the two token addresses point to the contracts for the two types of ERC-20 tokens that can be exchanged by this pool.

The two private reserve integers reflect exchange reserves for each token, and it is generally assumed that the two represent the same amount of value. The third `blockTimestampLast` integer will track the last block in which an exchange occurred for rate calculation purposes.

The two price integers hold the cumulative costs for each token in terms of the other. The `kLast` integer is used to keep multiples of the two reserves constant during trades, and will be further described below.

The lock-related integer and modifier support reentrancy prevention.

The `Mint` event is emitted each time liquidity tokens are created via `mint`. The `Burn`, `Swap`, and `Sync` events are analogous.

4.2 – The `getReserves()` Function

The `getReserves()` function is a simple getter for the two reserves and timestamp contract state variables.

```
function getReserves() public view returns (uint112 _reserve0, uint112 _reserve1, uint32
↳ _blockTimestampLast) {
    _reserve0 = reserve0;
    _reserve1 = reserve1;
    _blockTimestampLast = blockTimestampLast;
}
```

This external, read-only, and publicly-accessible function returns the reserves of the token pair which is used to price trades and distribute liquidity. The function also returns the mod 2^{32} timestamp of the last block during which an interaction occurred for the pair.

4.3 – The `_safeTransfer()` Function

The `_safeTransfer()` function transfers ERC-20 tokens from the exchange to another address.

```
    function _safeTransfer(address token, address to, uint value) private {
        (bool success, bytes memory data) = token.call(abi.encodeWithSelector(SELECTOR, to,
        ↳ value));
        require(success && (data.length == 0 || abi.decode(data, (bool))), 'UniswapV2:
        ↳ TRANSFER_FAILED');
    }
```

This internal state-changing function receives arguments for the token address, the destination address and the value. It uses the abi SELECTOR when calling into the token. The function reverts if the external call returns false or if it ends normally but reports a failure.

4.4 – The `initialize()` Function

The `initialize()` function is called by the factory contract to initialize the exchange's token state address variables.

```
// called once by the factory at time of deployment
function initialize(address _token0, address _token1) external {
    require(msg.sender == factory, 'UniswapV2: FORBIDDEN'); // sufficient check
    token0 = _token0;
    token1 = _token1;
}
```

This external state-changing function can only be called by the factory and receives two token addresses that it stores in the contract state. Nothing *here* prevents this function from being called multiple times, thus presenting a risk that the underlying tokens could change during operation which could potentially impact liquidity providers.

4.5 – The `_update()` Function

The `_update()` function is called by the exchange's `mint()`, `burn()`, `swap()`, and `skim()` functions to update the contract state variables for reserves and price accumulators.

```
// update reserves and, on the first call per block, price accumulators
function _update(uint balance0, uint balance1, uint112 _reserve0, uint112 _reserve1)
↪ private {
    require(balance0 <= uint112(-1) && balance1 <= uint112(-1), 'UniswapV2: OVERFLOW');
    uint32 blockTimestamp = uint32(block.timestamp % 2**32);
    uint32 timeElapsed = blockTimestamp - blockTimestampLast; // overflow is desired
    if (timeElapsed > 0 && _reserve0 != 0 && _reserve1 != 0) {
        // * never overflows, and + overflow is desired
        price0CumulativeLast += uint(UQ112x112.encode(_reserve1).uqdiv(_reserve0)) *
        ↪ timeElapsed;
        price1CumulativeLast += uint(UQ112x112.encode(_reserve0).uqdiv(_reserve1)) *
        ↪ timeElapsed;
    }
    reserve0 = uint112(balance0);
    reserve1 = uint112(balance1);
    blockTimestampLast = blockTimestamp;
    emit Sync(reserve0, reserve1);
}
```

This private, internal, and state-changing function is called every time tokens are deposited or withdrawn, and receives both token balances and both token reserves. First the magnitude of the `uint112` balances is checked then the `block.timestamp` is truncated to 32 bits. The `timeElapsed` value is calculated (and should never overflow as it is only ever set just down below). With non-zero timestamp and reserve values, new cumulative prices are calculated. The reserves and last timestamp are set and the `Sync` event is emitted. Note that balances are not checked against zero.

4.6 – The `_mintFee()` Function

The `_mintFee()` function is called by the exchange's `mint()` and `burn()` functions to handle the protocol fee.

```
// if fee is on, mint liquidity equivalent to 1/6th of the growth in sqrt(k)
function _mintFee(uint112 _reserve0, uint112 _reserve1) private returns (bool feeOn) {
    address feeTo = IUniswapV2Factory(factory).feeTo();
    feeOn = feeTo != address(0);
    uint _kLast = kLast; // gas savings
    if (feeOn) {
        if (_kLast != 0) {
           uint rootK = Math.sqrt(uint(_reserve0).mul(_reserve1));
           uint rootKLast = Math.sqrt(_kLast);
           if (rootK > rootKLast) {
               uint numerator = totalSupply.mul(rootK.sub(rootKLast));
               uint denominator = rootK.mul(5).add(rootKLast);
               uint liquidity = numerator / denominator;
               if (liquidity > 0) _mint(feeTo, liquidity);
           }
        }
    } else if (_kLast != 0) {
        kLast = 0;
    }
}
```

This private, internal, and state-changing function is called when liquidity is added or removed from the exchange, and receives both (new) token reserves. First the `feeTo` value is retrieved and if zero, no action is performed other than forcing `kLast` to zero. If there is a non-zero `feeTo` address set, the remainder of the logic calculates the 0.05% fee and mints it. The function returns a boolean indication of whether the `feeTo` state variable is enabled.

4.7 – The `mint()` Function

The `_mint()` function is called when a liquidity provider adds liquidity to the trading pair.

```
// this low-level function should be called from a contract which performs important
↳ safety checks
function mint(address to) external lock returns (uint liquidity) {
    (uint112 _reserve0, uint112 _reserve1,) = getReserves(); // gas savings
    uint balance0 = IERC20(token0).balanceOf(address(this));
    uint balance1 = IERC20(token1).balanceOf(address(this));
    uint amount0 = balance0.sub(_reserve0);
    uint amount1 = balance1.sub(_reserve1);

    bool feeOn = _mintFee(_reserve0, _reserve1);
    uint _totalSupply = totalSupply; // gas savings, must be defined here since
    ↳ totalSupply can update in _mintFee
    if (_totalSupply == 0) {
        liquidity = Math.sqrt(amount0.mul(amount1)).sub(MINIMUM_LIQUIDITY);
       _mint(address(0), MINIMUM_LIQUIDITY); // permanently lock the first
       ↳ MINIMUM_LIQUIDITY tokens
    } else {
        liquidity = Math.min(amount0.mul(_totalSupply) / _reserve0,
        ↳ amount1.mul(_totalSupply) / _reserve1);
    }
    require(liquidity > 0, 'UniswapV2: INSUFFICIENT_LIQUIDITY_MINTED');
    _mint(to, liquidity);

    _update(balance0, balance1, _reserve0, _reserve1);
    if (feeOn) kLast = uint(reserve0).mul(reserve1); // reserve0 and reserve1 are up-to-
    ↪ date
    emit Mint(msg.sender, amount0, amount1);
}
```

This external, state-changing, publicly-accessible, and 'locking' function mints liquidity tokens (and does not involve minting of the ERC-20 token/trading pairs). The function receives a recipient address, retrieves both the reserves and balances of each token in the pair, and then calculates proposed amounts by subtracting. After processing the protocol fee and determining the total supply, the liquidity-to-mint amount is calculated (after MINIMUM_LIQUIDITY is reserved). The non-zero liquidity is minted, the contract balances/ reserves are updated, and a `Mint` event is emitted.

Note that the preceding code comment indicating the need for important safety checks is not referring to overall system security but rather the potential for user mistakes.

4.8 – The `burn()` Function

The `burn()` function is called when liquidity is withdrawn and the appropriate liquidity tokens need to be burned.

```
// this low-level function should be called from a contract which performs important
↪ safety checks
function burn(address to) external lock returns (uint amount0, uint amount1) {
    (uint112 _reserve0, uint112 _reserve1,) = getReserves(); // gas savings
    address _token0 = token0;                                // gas savings
    address _token1 = token1;                                // gas savings
    uint balance0 = IERC20(_token0).balanceOf(address(this));
    uint balance1 = IERC20(_token1).balanceOf(address(this));
    uint liquidity = balanceOf[address(this)];

    bool feeOn = _mintFee(_reserve0, _reserve1);
    uint _totalSupply = totalSupply; // gas savings, must be defined here since
    ↪ totalSupply can update in _mintFee
    amount0 = liquidity.mul(balance0) / _totalSupply; // using balances ensures pro-rata
    ↪ distribution
    amount1 = liquidity.mul(balance1) / _totalSupply; // using balances ensures pro-rata
    ↪ distribution
    require(amount0 > 0 && amount1 > 0, 'UniswapV2: INSUFFICIENT_LIQUIDITY_BURNED');
    _burn(address(this), liquidity);
    _safeTransfer(_token0, to, amount0);
    _safeTransfer(_token1, to, amount1);
    balance0 = IERC20(_token0).balanceOf(address(this));
    balance1 = IERC20(_token1).balanceOf(address(this));

    _update(balance0, balance1, _reserve0, _reserve1);
    if (feeOn) kLast = uint(reserve0).mul(reserve1); // reserve0 and reserve1 are up-to-
    ↪ date
    emit Burn(msg.sender, amount0, amount1, to);
}
```

This external, state-changing, publicly-accessible, and 'locking' function burns liquidity tokens when liquidity is removed from the pair (and does not involving the burning of either of the ERC-20 tokens). The function is very similar to `mint()` as it receives a recipient address and then retrieves both the reserves and balances of each token in the pair (along with liquidity balance). After processing the protocol fee, the liquidity-to-burn and corresponding token amounts are calculated. The liquidity is burned, ERC-20 tokens are transferred, the contract balances/reserves are updated, and a `Burn` event is emitted.

Note that the preceding code comment indicating the need for important safety checks is not referring to overall system security but rather the potential for user mistakes.

4.9 – The `swap()` Function

The `swap()` function is called to swap tokens – all swaps happen in this central function called by another smart contract.

```
// this low-level function should be called from a contract which performs important safety checks
function swap(uint amount0Out, uint amount1Out, address to, bytes calldata data) external lock {
    require(amount0Out > 0 || amount1Out > 0, 'UniswapV2: INSUFFICIENT_OUTPUT_AMOUNT');
    (uint112 _reserve0, uint112 _reserve1,) = getReserves(); // gas savings
    require(amount0Out < _reserve0 && amount1Out < _reserve1, 'UniswapV2: INSUFFICIENT_LIQUIDITY');

    uint balance0;
    uint balance1;
    { // scope for _token{0,1}, avoids stack too deep errors
    address _token0 = token0;
    address _token1 = token1;
    require(to != _token0 && to != _token1, 'UniswapV2: INVALID_TO');
    if (amount0Out > 0) _safeTransfer(_token0, to, amount0Out); // optimistically transfer tokens
    if (amount1Out > 0) _safeTransfer(_token1, to, amount1Out); // optimistically transfer tokens
    if (data.length > 0) IUniswapV2Callee(to).uniswapV2Call(msg.sender, amount0Out, amount1Out, data);
    balance0 = IERC20(_token0).balanceOf(address(this));
    balance1 = IERC20(_token1).balanceOf(address(this));
    }
    uint amount0In = balance0 > _reserve0 - amount0Out ? balance0 - (_reserve0 - amount0Out) : 0;
    uint amount1In = balance1 > _reserve1 - amount1Out ? balance1 - (_reserve1 - amount1Out) : 0;
    require(amount0In > 0 || amount1In > 0, 'UniswapV2: INSUFFICIENT_INPUT_AMOUNT');
    { // scope for reserve{0,1}Adjusted, avoids stack too deep errors
    uint balance0Adjusted = balance0.mul(1000).sub(amount0In.mul(3));
    uint balance1Adjusted = balance1.mul(1000).sub(amount1In.mul(3));
    require(balance0Adjusted.mul(balance1Adjusted) >= uint(_reserve0).mul(_reserve1).mul(1000**2), 'UniswapV2: K');
    }

    _update(balance0, balance1, _reserve0, _reserve1);
    emit Swap(msg.sender, amount0In, amount1In, amount0Out, amount1Out, to);
}
```

This external, state-changing, publicly-accessible, and 'locking' function performs a token swap for tokens that were (generally) previously transferred to the exchange contract. The function receives indication of the desired output token amounts, a recipient address, and a bytestring. The desired output amounts must not both be non-zero and each less than the corresponding reserve, and the recipient must not be one of the token addresses. The non-zero tokens are optimistically transferred, and balances updated. The necessary input amounts are calculated and are required to be positive. Adjusted balances are calculated, alongside the 0.3% fee, and balances/reserves are updated.

4.10 – The `skim()` Function

The `skim()` function is one of two recovery mechanisms that allows a more graceful handling of situations where total supplies are greater than the `uint112` allows.

```
    // force balances to match reserves
    function skim(address to) external lock {
        address _token0 = token0; // gas savings
        address _token1 = token1; // gas savings
        _safeTransfer(_token0, to, IERC20(_token0).balanceOf(address(this)).sub(reserve0));
        _safeTransfer(_token1, to, IERC20(_token1).balanceOf(address(this)).sub(reserve1));
    }
```

This external, state-changing, publicly-accessible, and ‘locking’ function allows a user to withdraw the difference between the current balance and `uint112.MAX` should that be greater than zero. This is not relevant to Kaddex.

4.11 – The `sync()` Function

The `sync()` function is the second of two recovery mechanisms that protects against token implementations that can update the pair’s balance.

```
// force reserves to match balances
function sync() external lock {
    _update(IERC20(token0).balanceOf(address(this)),
    ↪ IERC20(token1).balanceOf(address(this)), reserve0, reserve1);
}
```

This external, state-changing, publicly-accessible, and ‘locking’ function resets the reserves of the token contracts to match the corresponding balance.

5 – The `UniswapV2ERC20` Contract

The `UniswapV2ERC20.sol` contract implements the ERC-20 liquidity token. The purpose of a standard like ERC-20 is to allow interoperable tokens with widely reviewed functionality. This particular contract does contain some divergence that will be highlighted below.

5.1 – Contract State Variables, Events, and Constructor

The no-argument constructor below is called during deployment and sets the `chained` state and calculates a 32-byte DOMAIN_SEPARATOR. All constants are effectively hardcoded and other state variables are initialized as empty.

```
string public constant name = 'Uniswap V2';
string public constant symbol = 'UNI-V2';
uint8 public constant decimals = 18;
```

```
uint  public totalSupply;
mapping(address => uint) public balanceOf;
mapping(address => mapping(address => uint)) public allowance;

bytes32 public DOMAIN_SEPARATOR;
// keccak256("Permit(address owner,address spender,uint256 value,uint256 nonce,uint256
↪ deadline)");
bytes32 public constant PERMIT_TYPEHASH = 0x6e71edae12b1b97f4d1f60370fef10105fa2faae0126114a
↪ 169c64845d6126c9;
mapping(address => uint) public nonces;

event Approval(address indexed owner, address indexed spender, uint value);
event Transfer(address indexed from, address indexed to, uint value);

constructor() public {
    uint chainId;
    assembly {
        chainId := chainid
    }
    DOMAIN_SEPARATOR = keccak256(
        abi.encode(
           keccak256('EIP712Domain(string name,string version,uint256 chainId,address
           ↪ verifyingContract)'),
           keccak256(bytes(name)),
           keccak256(bytes('1')),
           chainId,
           address(this)
        )
    );
}
```

The name, symbol, and decimals shown above correspond to ERC-20 definitions but are hardcoded. The `totalSupply` and initial mapping of account balances are empty. Nonces are tracked to prevent replay attacks.

5.2 – The `_mint()` Function

The `_mint()` function is called to create specific amount tokens, credit this amount to a specific account, and update the total supply.

```
    function _mint(address to, uint value) internal {
        totalSupply = totalSupply.add(value);
        balanceOf[to] = balanceOf[to].add(value);
        emit Transfer(address(0), to, value);
    }
```

This internal state-changing function receives a recipient account address and an amount of tokens to mint. The total supply is updated upwards, the recipient's balance is credited, and an event is emitted.

5.3 – The `_burn()` Function

The `_burn()` function is called to destroy a specific amount of tokens, debit this amount to a specific account, and update the total supply.

```
function _burn(address from, uint value) internal {
    balanceOf[from] = balanceOf[from].sub(value);
    totalSupply = totalSupply.sub(value);
    emit Transfer(from, address(0), value);
}
```

This internal state-changing function receives a recipient account address and an unvalidated amount of tokens to burn. The total supply is updated downwards, the recipient's balance is credited, and an event is emitted. There is no validation here to prevent the subtraction from rolling over, though in some cases a SafeMath library is used. This is not relevant to Kaddex.

5.4 – The `_approve()` Function

The `_approve()` function is called by the public `approve()` API function, and creates an allowance for a `spender` over the caller's token.

```
function _approve(address owner, address spender, uint value) private {
    allowance[owner][spender] = value;
    emit Approval(owner, spender, value);
}
```

This internal state-changing function receives an owner account address, a spender account address and an amount of tokens to reserve as an allowance. The allowance mapping is set as appropriate, and an event is emitted.

5.5 – The `_transfer()` Function

The `_transfer()` function is called by the public `transfer()` API function, and transfers tokens from source address `from` to sink address `to`.

```
function _transfer(address from, address to, uint value) private {
    balanceOf[from] = balanceOf[from].sub(value);
    balanceOf[to] = balanceOf[to].add(value);
    emit Transfer(from, to, value);
}
```

This internal state-changing function receives a source address, a sink address, and an amount of tokens to transfer. The transfer amount is subtracted from the source mapping entry and added to the sink mapping entry. An event is emitted.

5.6 – The `approve()` Function

The public `approve()` API function is a gatekeeper for the internal `_approve()` function described earlier.

```
function approve(address spender, uint value) external returns (bool) {
    _approve(msg.sender, spender, value);
    return true;
}
```

This function simply extracts the message sender address and uses that to specify the `owner` when calling the internal `_approve` function.

5.7 – The `transfer()` Function

The public `transfer()` API function is a gatekeeper for the internal `_transfer()` function described earlier.

```
function transfer(address to, uint value) external returns (bool) {
    _transfer(msg.sender, to, value);
    return true;
}
```

This function simply extracts the message sender address and uses that to specify the `owner` when calling the internal `_approve` function.

5.8 – The `transferFrom()` Function

The public `transferFrom()` API function moves an approved amount of tokens from source address `from` to sink address `to` using the allowance mechanism.

```
function transferFrom(address from, address to, uint value) external returns (bool) {
    if (allowance[from][msg.sender] != uint(-1)) {
        allowance[from][msg.sender] = allowance[from][msg.sender].sub(value);
    }
    _transfer(from, to, value);
    return true;
}
```

This function receives a source address, a sink address, and an amount. If the allowance mapping from source to message sender is not set to its maximum value, it is reduced by amount. The transfer function is then called.

5.9 – The `permit()` Function

The `permit()` function is similar to the approve function except it allows for modifying the allowance by a signed message rather than strictly by the message sender.

```
function permit(address owner, address spender, uint value, uint deadline, uint8 v, bytes32
↪ r, bytes32 s) external {
    require(deadline >= block.timestamp, 'UniswapV2: EXPIRED');
    bytes32 digest = keccak256(
        abi.encodePacked(
            '\x19\x01',
            DOMAIN_SEPARATOR,
            keccak256(abi.encode(PERMIT_TYPEHASH, owner, spender, value, nonces[owner]++,
            ↪ deadline))
        )
    );
    address recoveredAddress = ecrecover(digest, v, r, s);
    require(recoveredAddress != address(0) && recoveredAddress == owner, 'UniswapV2:
    ↪ INVALID_SIGNATURE');
    _approve(owner, spender, value);
}
```

This function receives an owner and spender address, token amount, deadline, and three scalar values for the signature. A digest of the expected message is calculated, and `ecrecover` is used to determine the address that signed it with `v`, `r`, and `s`. If the signature is valid, the `_approve()` function is called with the `owner` parameter (rather than message sender).

6 – Kaddex `exchange.pact` (Relative to Kadenaswap and Uniswap V2)

The functions implemented in `exchange.pact` are listed below, with the primary API bolded. New functions that are not present in Kadenaswap are annotated with `n`, while relatively simple read-only helper functions are annotated `r-h`. Note that most functions, **but not all**, have their return type annotated in the code. It would likely be beneficial to enforce this over all functions.

1. **add-liquidity**
2. add-tracked-path (`n`)
3. burn
4. canonicalize (`r-h`)
5. chunk-list-pairs (`n`, `r-h`)
6. compute-in
7. compute-out
8. create-fee-account (`n`, `r-h`)
9. create-pair-account (`r-h`)
10. **create-pair**
11. dump-observations (`n`, debug)
12. enforce-contract-unlocked (`n`, `r-h`)
13. get-observation-key (`n`, `r-h`)
14. get-oracle-time-cumulative-price (`n`)
15. get-pair-by-key (`n`, `r-h`)
16. get-pair-key (`r-h`)
17. get-pair (`r-h`)
18. get-pairs (`n`, `r-h`)
19. get-spot-price-for (`n`)
20. init
21. is-canonical (`r-h`)
22. is-leg0 (`r-h`)
23. is-path-tracked (`n`)
24. leg-for (`r-h`)
25. maybe-observe (`n`)
26. mint-fee (`n`)
27. mint-fee-manual (`n`)
28. mint
29. observe-compound-path (`n`)
30. observe-direct (`n`)
31. oracle-add-tracked-path (`n`, `r-h`)
32. pair-exists (`r-h`)
33. quote (`r-h`)
34. **remove-liquidity**
35. reserve-for (`r-h`)
36. rotate-fee-guard (`n`)
37. set-contract-lock (`n`)
38. **swap**
39. swap-alloc
40. **swap-exact-in**
41. **swap-exact-out**
42. swap-pair
43. truncate (`r-h`)
44. update-k (`n`)
45. update-reserves

The `exchange.pact` module makes use of `tokens.pact` to handle liquidity tokens, along with a number of other helper and interface files.

The primary `exchange.pact` API is described below.

6.1 – The `create-pair` Function

This function is called by the administrator function when initiating coverage for a new pair of distinct tokens. As described in the comments below, this involves creating a new pair record, a pair-specific liquidity token, and new empty accounts for each leg of the pair.

Once established, liquidity providers may utilize (for example) `add-liquidity` and then traders may utilize (for example) `swap-exact-out`.

```
997  (defun create-pair:object{pair}
998    ( token0:module{fungible-v2}
999      token1:module{fungible-v2}
1000     hint:string
1001     )
1002   " Create new pair for legs TOKEN0 and TOKEN1. This creates a new \
1003   \ pair record, a liquidity token named after the canonical pair key \
1004   \ in the 'tokens' module, and new empty accounts in each leg token. \
1005   \ If account key value is already taken in leg tokens, transaction \
1006   \ will fail, which is why HINT exists (which should normally be \"\"), \
1007   \ to further seed the hash function creating the account id."
1008   (enforce-contract-unlocked)
1009   (let* ((key (get-pair-key token0 token1))
1010          (canon (is-canonical token0 token1))
1011          (ctoken0 (if canon token0 token1))
1012          (ctoken1 (if canon token1 token0))
1013          (a (create-pair-account key hint))
1014          (g (create-module-guard key))
1015          (f (create-fee-account key hint))
1016          (fg (create-module-guard key)) ;; TODO: replace this with something the other
              ↳contracts can use?
1017          (p { 'leg0: { 'token: ctoken0, 'reserve: 0.0 }
1018             , 'leg1: { 'token: ctoken1, 'reserve: 0.0 }
1019             , 'account: a
1020             , 'guard: g
1021             , 'fee-account: f
1022             , 'fee-guard: fg
1023             , 'last-k: 0.0
1024             , 'locked: false
1025             })
1026          )
1027     ;; create the table entry and all the token accounts
1028     (with-capability (CREATE_PAIR ctoken0 ctoken1 key a)
1029       (insert pairs key p)
1030       (token0::create-account a g)
1031       (token1::create-account a g)
1032       (tokens.create-account key a g)
1033       (tokens.create-account key f fg)
1034       { "key": key
1035       , "account": a
1036       }))
1037   )
```

The above function receives arguments indicating the two specific tokens of interest along with a hint string, then ensures that the overall contract is in the unlocked state.

Lines 1009-1026 calculate the information necessary to subsequently populate the records below. The key is essentially a string concatenation of the canonically ordered token names. The tokens are canonically re-ordered if necessary. A pair account and module guard is created, followed by creation of a fee account and module guard. A pair record `p` is then populated.

Finally, state is changed starting on line 1029 as the pair record is inserted, each token's `create-account` is called with pair-account info, and the liquidity token's `create-account` is called with both the pair-account and fee-account (and guards). The function returns a pair of key:account information to its caller.

6.2 – The `add-liquidity` Function

This function is called by a liquidity provider when contributing liquidity to an established token-pair. Much of the logic involves calculating correct proportions of tokens; note that the target token-pair may have an initial balance of zero or non-zero. Once a token-pair has liquidity, the liquidity can either be removed or the tokens can be traded.

```
(defun add-liquidity:object
  ( tokenA:module{fungible-v2}
    tokenB:module{fungible-v2}
    amountADesired:decimal
    amountBDesired:decimal
    amountAMin:decimal
    amountBMin:decimal
    sender:string
    to:string
    to-guard:guard
  )
  "Adds liquidity to an existing pair. The `to` account specified will receive the
  ↪ liquidity tokens."
  (enforce-contract-unlocked)
  (tokenA::enforce-unit amountADesired) ;; enforce the informed amounts are in the correct
  ↪ precision
  (tokenB::enforce-unit amountBDesired)
  (with-capability (MUTEX) ;; obtain the mutex lock
    (obtain-pair-lock (get-pair-key tokenA tokenB)))
  (let*
    ( (p (get-pair tokenA tokenB))
      (reserveA (reserve-for p tokenA))
      (reserveB (reserve-for p tokenB))
      ;; calculate the actual amounts of liquidity that will be added to keep the reserve
      ↪ ratios
      (amounts
        (if (and (= reserveA 0.0) (= reserveB 0.0))
         [amountADesired amountBDesired]
         (let ((amountBOptimal (quote amountADesired reserveA reserveB)))
           (if (<= amountBOptimal amountBDesired)
             (let ((x (enforce (>= amountBOptimal amountBMin)
                        "add-liquidity: insufficient B amount")))
               [amountADesired amountBOptimal])
             (let ((amountAOptimal (quote amountBDesired reserveB reserveA)))
               (enforce (<= amountAOptimal amountADesired)
                 "add-liquidity: optimal A less than desired")
               (enforce (>= amountAOptimal amountAMin)
                 "add-liquidity: insufficient A amount")
               [amountAOptimal amountBDesired])))))
      (amountA (truncate tokenA (at 0 amounts)))
      (amountB (truncate tokenB (at 1 amounts)))
      (pair-account (at 'account p))
    )
    (with-capability (UPDATING) ;; if necessary, mint exchange fees
      (mint-fee p))
    ;; transfer the tokens from the user to the pair
    (tokenA::transfer sender pair-account amountA)
    (tokenB::transfer sender pair-account amountB)
    ;; mint the liquidity tokens to the user
    (let* ;; first we calculate the actual amounts transferred by calling `get-balance`
      ( (token0:module{fungible-v2} (at 'token (at 'leg0 p)))
        (token1:module{fungible-v2} (at 'token (at 'leg1 p)))
        (balance0 (token0::get-balance pair-account))
        (balance1 (token1::get-balance pair-account))
        (reserve0 (at 'reserve (at 'leg0 p)))
        (reserve1 (at 'reserve (at 'leg1 p)))
        (amount0 (- balance0 reserve0))
        (amount1 (- balance1 reserve1))
        (key (get-pair-key tokenA tokenB))
        ;; given the liquidity tokens' total supply, calculate the amount of liquidity we
        ↪ need to mint
        (totalSupply (tokens.total-supply key))
        (lock-account (create-lock-account "")) ;; TODO: add a hint parameter to this
        ↪ function as well
        (liquidity (tokens.truncate key
         (if (= totalSupply 0.0) ;; in this case, we need to mint MINIMUM_LIQUIDITY
           (with-capability (ISSUING)
             (mint key lock-account (at 'guard p) MINIMUM_LIQUIDITY)
             (- (sqrt (* amount0 amount1)) MINIMUM_LIQUIDITY))
           (let ((l0 (/ (* amount0 totalSupply) reserve0))
                 (l1 (/ (* amount1 totalSupply) reserve1))
                )
             ;; here we take the minimum between l0 and l1
             (if (<= l0 l1) l0 l1)))))
      )
      ;; mint the liquidity for the user
      (enforce (> liquidity 0.0) "mint: insufficient liquidity minted")
      (with-capability (ISSUING)
        (mint key to to-guard liquidity))
      ;; update pair reserves and last-k value
      (with-capability (UPDATING)
        (update-reserves p key balance0 balance1)
        (update-k key))
      ;; release the pair lock
      (with-capability (MUTEX)
        (release-pair-lock (get-pair-key tokenA tokenB)))
      ;; return the information to the user
      { "liquidity": liquidity
      , "supply": (tokens.total-supply key)
      , "amount0": amount0
      , "amount1": amount1
      }
    )
  )
)
```

The above function receives arguments indicating the two specific tokens of interest, pairs of amounts desired and minimum acceptable amounts, the sender, and the recipient. The code first ensures the contract is in the unlocked state and validates the precision of the desired amounts (finding "Weak Input Validation on `exchange.pact` API" suggests also validating for positive values). A pair-specific mutex/lock is then acquired to prevent reentrancy.

Lines 539-561 calculate the actual amount of liquidity to be added to each token in order to keep the reserve ratios constant (prior to truncation). If both token reserves are empty, this ratio is simply set to the provided desired amounts. If not, the logic starts with the amount of token A desired, and sees if the calculated amount of token B is above its requested minimum. Failing this, the A-B roles are reversed and the calculation will rerun. Minimal amounts are enforced. The tokens are then transferred on lines 565-566.

Lines 568-578 calculate the actual amounts sent above for use in the subsequent liquidity calculations. This is done by retrieving balances and reserves, then subtracting. If the total supply is 0, then the minimum liquidity is minted to the lock account and subtracted away. Otherwise, the minimum of `amount0 * totalSupply / reserve0` and `amount1 * totalSupply / reserve1` is returned as liquidity. The results of these calculations can be seen in an example scenario tested on lines 162-223 of `exchange.repl`.

Finally, line 592-607 mints the (positive) liquidity to the recipient, updates reserves and k, releases the mutex pair-lock, and returns the calculated liquidity, total supply, and token amounts info to the function caller.

6.3 – The `remove-liquidity` Function

This function is called by a liquidity provider when extracting liquidity from an established token-pair. As with `add-liquidity` much of the logic involves calculating correct proportions of tokens.

```
646 (defun remove-liquidity:object
647   ( tokenA:module{fungible-v2}
648     tokenB:module{fungible-v2}
649     liquidity:decimal
650     amountAMin:decimal
651     amountBMin:decimal
652     sender:string
653     to:string
654     to-guard:guard
655   )
656   "Removes liquidity from an existing pair. The `to` account specified will receive the
    ↪ tokens."
657   (enforce-contract-unlocked)
658   (with-capability (MUTEX) ;; obtain the pair lock
659     (obtain-pair-lock (get-pair-key tokenA tokenB)))
660   (let* ( (p (get-pair tokenA tokenB))
661           (pair-account (at 'account p))
662           (pair-key (get-pair-key tokenA tokenB))
663         )
664     ;; if necessary, mint fee tokens
665     (with-capability (UPDATING)
666       (mint-fee p))
667     ;; transfer liquidity tokens from the sender to the pair for burning
668     (tokens.transfer pair-key sender pair-account liquidity)
669     (let* ;; calculate current reserves and withdrawal amount
670       ( (token0:module{fungible-v2} (at 'token (at 'leg0 p)))
671         (token1:module{fungible-v2} (at 'token (at 'leg1 p)))
672         (balance0 (token0::get-balance pair-account))
673         (balance1 (token1::get-balance pair-account))
674         (total-supply (tokens.total-supply pair-key))
675         (amount0 (truncate token0 (/ (* liquidity balance0) total-supply)))
676         (amount1 (truncate token1 (/ (* liquidity balance1) total-supply)))
677         (canon (is-canonical tokenA tokenB))
678       )
679       ;; enforce values are sensible
680       (enforce (and (> amount0 0.0) (> amount1 0.0))
681         "remove-liquidity: insufficient liquidity burned")
682       (enforce (>= (if canon amount0 amount1) amountAMin)
683         "remove-liquidity: insufficient A amount")
684       (enforce (>= (if canon amount1 amount0) amountBMin)
685         "remove-liquidity: insufficient B amount")
686       ;; burn the liquidity tokens received from the user
687       (with-capability (ISSUING)
688         (burn pair-key pair-account liquidity))
689       ;; transfer both tokens to the user
690       (install-capability (token0::TRANSFER pair-account to amount0))
691       (token0::transfer-create pair-account to to-guard amount0)
692       (install-capability (token1::TRANSFER pair-account to amount1))
693       (token1::transfer-create pair-account to to-guard amount1)
694       ;; update the reserves with the new balances and the last-k value
695       (with-capability (UPDATING)
696         (update-reserves p pair-key
697          (token0::get-balance pair-account)
698          (token1::get-balance pair-account))
699         (update-k pair-key))
700       ;; release the pair lock
701       (with-capability (MUTEX)
702         (release-pair-lock (get-pair-key tokenA tokenB)))
703       ;; return the withdrawn amounts
704       { 'amount0: amount0
705       , 'amount1: amount1
706       }
707     )
708   )
709 )
```

The above function receives arguments indicating the two specific tokens of interest, an amount of liquidity to 'reclaim', the minimum acceptable token amounts, a sender, and a recipient. The code first ensures the contract is in the unlocked state and acquires a pair-specific mutex/lock to prevent reentrancy.

Lines 660-663 retrieves the pair account and key. The `mint-fee` function is called on line 666 to update the fee account, followed by the liquidity token reclamation on line 668. Lines 670-679 retrieves the token balances, total supply, and calculates the proportional (proposed) amounts of each tokens; these amounts are validated against positive values and desired minimums on lines 680-686.

The liquidity tokens are then burnt on line 688, token amounts are transferred on lines 691/693, the reserves and k are updated, the pair-specific mutex/lock is released, and the amount values are reported to the function caller.

6.4 – The `swap-exact-in` Function

This function performs a series of swaps along a specified path such that an exact amount of input token results in at least a minimum amount of output tokens.

```
728 (defun swap-exact-in
729   ( amountIn:decimal
730     amountOutMin:decimal
731       path:[module{fungible-v2}]
732       sender:string
733       to:string
734       to-guard:guard
735     )
736     "Swaps exactly `amountIn` using the token path `path`. Sends from `sender` and `to`
        ↳ receives the result tokens. Ensures that the final output amount is at least
        ↳ `amountOutMin`"
737     (enforce-contract-unlocked)
738     (enforce (>= (length path) 2) "swap-exact-in: invalid path")
739     ;; fold over tail of path with dummy first value to compute outputs
740     ;; assembles allocs in reverse order
741     (let*
742       ( (p0 (get-pair (at 0 path) (at 1 path)))
743         (allocs
744           (fold (compute-out)
745             [ { 'token-out: (at 0 path)
746               , 'token-in: (at 1 path)
747               , 'out: amountIn
748               , 'in: 0.0
749               , 'idx: 0
750               , 'pair: p0
751               , 'path: path
752               }]
753             (drop 1 path)))
754       )
755       (enforce (>= (at 'out (at 0 allocs)) amountOutMin)
756         (format "swap-exact-in: insufficient output amount {}" [(at 'out (at 0 allocs))]))
757       ;; initial dummy is correct for initial transfer
758       (with-capability (SWAPPING)
759         (swap-pair sender to to-guard (reverse allocs)))
760     )
761   )
```

The above function receives arguments indicating the exact amount of input tokens, minimum acceptable amount of output tokens, a path consisting of a list of tokens, a sender, and a recipient. The code first ensures the contract is in the unlocked state and the path length is at least two (it does not validate that the path is acyclic).

Lines 742-755 assembles an `allocs` list of steps in reverse order using the `compute-out` function to calculate interim token values. The final output amount is validated against the minimum acceptable output amount. The `swap-pair` function is utilized to perform the actual swaps (in forward order as `allocs` is reversed).

6.5 – The `swap-exact-out` Function

This function performs a series of swaps along a specified path such that an exact amount of output tokens consumes no more than a maximum amount of input tokens.

```
792   (defun swap-exact-out
793     ( amountOut:decimal
794       amountInMax:decimal
795       path:[module{fungible-v2}]
796       sender:string
797       to:string
798       to-guard:guard
799     )
800   "Swaps enough tokens to get exactly `amountOut` using the token path `path`. Sends from
    ↳ `sender` and `to` receives the result tokens. Ensures that the initial input amount is
    ↳ at most `amountInMax`"
801   (enforce-contract-unlocked)
802   (enforce (>= (length path) 2) "swap-exact-out: invalid path")
803   ;; fold over tail of reverse path with dummy first value to compute inputs
804   ;; assembles allocs in forward order
805   (let*
806     ( (rpath (reverse path))
807       (path-len (length path))
808       (pz (get-pair (at 0 rpath) (at 1 rpath)))
809       (e:[module{fungible-v2}] [])
810       (allocs
811         (fold (compute-in)
812          [ { 'token-out: (at 1 rpath)
813            , 'token-in: (at 0 rpath)
814            , 'out: 0.0
815            , 'in: amountOut
816            , 'idx: path-len
817            , 'pair: pz
818            , 'path: e
819            }]
820          (drop 1 rpath)))
821       (allocs1 ;; drop dummy at end, prepend dummy for initial transfer
822         (+ [  { 'token-out: (at 0 path)
823              , 'token-in: (at 1 path)
824              , 'out: (at 'in (at 0 allocs))
825              , 'in: 0.0
826              , 'idx: 0
827              , 'pair: (at 'pair (at 0 allocs))
828              , 'path: path
829           } ]
830           (take (- path-len 1) allocs)))
831     )
832     (enforce (<= (at 'out (at 0 allocs1)) amountInMax)
833       (format "swap-exact-out: excessive input amount {}" [(at 'out (at 0 allocs1))]))
834     (with-capability (SWAPPING)
835       (swap-pair sender to to-guard allocs1))
836   )
837 )
```

The above function receives arguments indicating the exact amount of output tokens, maximum acceptable amount of input tokens, a path consisting of a list of tokens, a sender, and a recipient. The code first ensures the contract is in the unlocked state and the path length is at least two (it also does not validate the path is acyclic).

Lines 806-821 assembles an `allocs` list of steps in forward order using the `compute-in` function to calculate interim token values. The `allocs1` list is trivially derived from `allocs` by dropping the last item and prepending an initial item. The required input value is validated against the maximum acceptable amount. The `swap-pair` function is utilized to perform the actual swaps in forward order.

7 – Kaddex `wrapper.pact`

The functions implemented in `wrapper.pact` are listed below, with the primary API bolded. Relatively simple read-only helper functions are annotated `r-h`. Note that almost all functions have their return type annotated in the code, but a very few do not. It would likely be beneficial to enforce this over all functions. Several functions are marked for clean-up or removal.

1. **add-liquidity**
2. add-liquidity-one-sided
3. bank-guard (`r-h`)
4. burn-guard (`r-h`)
5. compute-elapsed-time (r-h`, d)
6. compute-remaining-time (`r-h`)
7. dump-liquidity (`r-h`)
8. dump-positions (`r-h`)
9. enforce-contract-unlocked (`r-h`)
10. get-all-pending-requests (`r-h`)
11. get-amount-out (`r-h` , d)
12. get-average-price (`r-h`)
13. get-base-token-extended (`r-h`)
14. get-base-token (`r-h`)
15. get-liquidity-account (`r-h`)
16. get-liquidity-position-key (`r-h`)
17. get-liquidity-position (`r-h`)
18. get-one-sided-liquidity-swap-amount
19. get-other-side-token-amount-after-swap
20. get-pair-account (`r-h`)
21. get-token-amounts-for-liquidity (`r-h`)
22. get-total-supply-considering-fees
23. get-user-pending-requests (`r-h`)
24. get-user-position-stats
25. init
26. is-base-path
27. is-reward-request-claimable (`r-h`)
28. liquidity-guard (`r-h`)
29. max (`r-h`)
30. min (`r-h`)
31. min-int (`r-h`)
32. mint-guard (`r-h`)
33. pair-guard (`r-h`)
34. pair-registered (`r-h`)
35. process-claim-request-if-necessary
36. register-pair
37. **register-pair-only**
38. remove-elem-from-list (`r-h`)
39. **remove-liquidity**
40. set-contract-lock
41. set-fee-multiplier-for-pair
42. swap-fees-for-base-and-bank
43. swap-for-base
44. tokens-equal (`r-h`)
45. update-positions-for-new-multiplier
46. update-single-position-for-new-multiplier
47. **withdraw-claim**
48. **withdraw-settled-fees-without-booster**

The `wrapper.pact` module makes use of `exchange.pact` to handle trading activity, along with a number of other helper and interface files.

The primary `wrapper.pact` API is described below. **Note that the source code is not excerpted for brevity**.

7.1 – The `register-pair-only` Function

This function 'initiates' coverage for a pair of tokens by creating accounts and setting up state. Liquidity is added separately (though these two operations are combined in `register-pair`).

The function receives arguments for two tokens, the token paths, and a hint string. The code first ensures the contract is in the unlocked state and requires the `OPS` capability. Lines 314-326 ensure/adjust correct ordering of the tokens and confirms the validity of the paths. Next, the pair account name is derived and the token's `create-account` function called, followed by the same scheme for liquidity token account. An entry to `trading-pairs` is inserted followed by another into `liquidity-accounts` .

7.2 – The `add-liquidity` Function

Similar to its namesake in `exchange.pact`, this function adds user liquidity to an existing pair of tokens.

The function receives arguments for two tokens, the two desired amounts, the minimum acceptable amounts, a sender, and a receiver. The code first ensures the contract is in the unlocked state and that the sender is not empty. Lines 901-913 derives the pair-key and retrieves the pair record from the `exchange` module, determines the total (initial) liquidity amount, followed by retrieving the liquidity accounts, its name and guard, and the liquidity position key. The `exchange.add-liquidity` function is then called with the calculated parameters. Finally, the reserve amounts for both tokens are retrieved.

Lines 914-927 first binds the results of the prior call to `exchange.add-liquidity` function to `add-liquidity-result`, reads (with defaults) the target liquidity position records. If no prior liquidity positions exist and the initial supply is zero, then an initial record is inserted minus a MINIMUM_LIQUIDITY fee). If no prior liquidity positions exist and the initial supply is not zero, then an initial record is inserted without subtracting the fee. The difference in the two prior cases is in the `tokenA-pooled` field and its tokenB sibling. If a prior liquidity position for the user does exist, then the two cases of A) starting at zero (if previously withdrawn) or, B) starting at non-zero. The liquidity position is then updated.

Finally, the liquidity account for the token pair is updated, and the (bind) `add-liquidity-result` returned from the call to `exchange.add-liquidity` is returned to this function’s caller.

7.3 – The `remove-liquidity` Function

Similar to its namesake in `exchange.pact`, this function removes user liquidity from an existing pair of tokens. It also handles/initiates KDX rewards.

The function receives arguments for two tokens, the amount of liquidity to remove, the minimum acceptable token amounts, a sender, a receiver and a boolean flag indicating KDX rewards. The code first ensures the contract is in the unlocked state and that the sender is not empty. Lines 619-635 retrieves the pair’s record from the `exchange` module, derives the pair-key, retrieves the liquidity account and name, retrieves liquidity position information, calculates the withdrawal fraction, and sets up pair-account values.

Following the above, lines 636-643 enforce a number of invariants. Lines 646-664 calculates two pooled values, performs a liquidity removal from the exchange contract, determines reserves and withdrawn amounts, calculates fee values and the remaining total liquidity.

If the initial KDX rewards boolean flag as true and fees are sufficient, lines 667-714 enforces several more invariants, transfers base adjusted amounts to the user, then constructs and writes a reward-claim-request record before writing the pending-request tables. Alternatively, lines 715-721 simply transfer the base tokens.

This function finishes by updating the user’s liquidity position and the token pair account balance.

7.4 – The `withdraw-claim` and `withdraw-settled-fees-without-booster` functions

These functions are related to the final steps in user/usage flow but are unexercised in the code repository. A full description here is prevented by project time pressure. They were included in the review.

8 – Kaddex `staking.pact`

The staking contract allows participants to stake native Kaddex AMM token (KDX) and earn rewards. Staking participants earn a portion of the Kaddex AMM fee, proportional to the amount they staked into the staking contract. The amount of staked KDX and staking/unstaking events do not influence the AMM's operation and as such, in its operation, the staking contract is adjacent to the AMM. The actual amount that's being rewarded to an account is roughly the amount the user staked times the "cumulative" delta. The cumulative is a monotonically increasing quantity, which grows with the staking contract's yield.

The staking contract has access to a portion of the AMM fees, expressed in AMM's LP tokens. The staking conrtact's operators are in charge of scooping the fee and making it available to the staking contract's participants. This happens in `sweep-some` and `sweep-one` functions. The `sweep-one` function removes the liquidity from the `fee-account` into staking contract's account pairs inside the wrapped tokens, then transfers them to accounts from which they will be swapped into KDX. This is done for each token pair that the staking contract keeps track of. From the perspective of a single token, there can exist multiple staking contract's accounts on that token, as a token can participate in more than one pairs. Those accounts are drained in the `sweep-some` function, converted into KDX and finally moved to the special `KDX_BANK` account on the KDX contract. It is from the `KDX_BANK` account that participants finally receive their rewards.

List of Functions

The functions implemented in `staking.pact` are listed below:

1. calculate-new-start
2. calculate-out (`r-h`)
3. calculate-penalty
4. calculate-reward
5. calculate-unlocked-stake
6. **claim**
7. enforce-contract-unlocked
8. fee-guard
9. get-kdx-guard
10. get-pair-record (`r-h`)
11. get-path (`r-h`)
12. get-pool-state (`r-h`)
13. get-stake-record (`r-h`)
14. get-token-record (`r-h`)
15. include-batch
16. include-one
17. init
18. **inspect-staker**
19. kdx-guard
20. lock-stake
21. max (`r-h`)
22. min (`r-h`)
23. onboard-with-lock
24. read-unlocked-stake (`r-h`)
25. read-waiting (`r-h`)
26. record-burn
27. **register-pair**
28. register-token-if-unregistered
29. **rollup**
30. set-contract-lock
31. **stake**
32. swap-to-kdx
33. **sweep-all**
34. sweep-one
35. sweep-some
36. token-guard
37. unroll-path (`r-h`)
38. **unstake**
39. wrap-guard

Notes below contain brief descriptions of some of the functionalities offered by some of the listed functions.

8.1 – Staking Account Locking Functions

The staking contract keeps track of active staking accounts in `stake-table`. Each staking account can potentially contain a list of locks. Each lock consists of an expiry date and the amount that is locked.

The locking functionality was implemented solely to support vesting schedule during the initial token sales. Multiple locks per account are included to support non-trivial vesting schedules.

Even though locks aren't going to be used in the future, the staking contract has to support them, at least up to some point. For example, even if support for adding accounts with locks is removed, functions such as `calculate-unlocked-stake` still have to exist, as the locking information is inside the table, which makes these functions relevant for the review.

Calculating Unlocked Stake

Each time tokens are unstaked, it is necessary to validate that the amount in question is not locked. This is done using two functions, `calculate-unlocked-stake` and `read-unlocked-stake`.

```
(defun calculate-unlocked-stake:decimal (amount:decimal locks:[object{stake-lock}])
  "Given a stake amount and a list of stake locks, calculate the unlocked amount  at the
  ↪current block time."
  (let*
    ( (now (at 'block-time (chain-data)))
      (is-lock-active (lambda (lock:object{stake-lock}) (< now (at 'until lock))))
      (active-locks (filter (is-lock-active) locks)) ;; filter input locks by validity
      (locked-amount (fold (+) 0.0 (map (at 'amount) active-locks)))
    )
    (if (> locked-amount amount) 0.0
      (- amount locked-amount))))
```

The function filters for active locks and subtracts the `locked-amount` from the supplied parameter `amount`. This function is wrapped by `read-unlocked-stake`, which simply calculates the ongoing unlocked amount for a given account.

```
(defun read-unlocked-stake:decimal (account:string)
  "Given an account name, fetch and calculate the unlocked stake amount at the current
  ↪block time."
  (with-read stake-table account
    { 'amount := amount
    , 'locks := locks }
    (calculate-unlocked-stake amount locks)))
```

For the purposes of locking actual funds, `lock-stake` and `onboard-with-lock` functions are exposed. It is worth noting that the `lock-stake` function does not perform sanity checks on the `until` parameter:

```
(defun lock-stake (account:string amount:decimal until:time)
  "Operator-only function to add a stake lock to a given account."
  (with-capability (OPS)
  (with-read stake-table account
    { 'locks := locks }
    (update stake-table account { 'locks: (+ locks  ;; add to locks, which is a list of
      ↳ stake-locks
      [ { 'amount: amount, 'until: until } ]) }))))
```

From the discussions with the development team, these functions appear redundant at this stage and they may be considered for removal.

8.2 – Token Pair Registration

The purpose of the staking contract is to extract a portion of the fees the AMM generates and distributes them to participants who enrolled in the staking protocol. As such, the staking contract needs to keep track of which token pools/pairs exist in the exchange. The staking contract does not fetch all the pools automatically, rather, this is supported as a manual process in which the operators register token pairs:

```
  (defun register-pair
    (token0:module{fungible-v2}
     token1:module{fungible-v2}
     hint:string)
    "Operator function to register a trading pair with the staking contract. \
    \ Registers the component tokens if unregistered, and takes ownership of the \
    \ exchange feeTo LP token accounts. Creates a pair-record row for the pair."
    (with-capability (OPS)
      (let*
        ( (p (exchange.get-pair token0 token1))
          (key (exchange.get-pair-key token0 token1))
          (token0-name (format "{}" [token0]))
          (token1-name (format "{}" [token1]))
          (pair-account-name (hash (format "{}_{}_{}" [token0 token1 hint])))
          (fee-account (at 'fee-account p)))
        ;; Create token accounts (per-token place for remove-liq consolidation)
        (register-token-if-unregistered token0 hint)
        (register-token-if-unregistered token1 hint)
        ;; Create pair accounts (per-pair 2x accounts to receive funds
        ;; immediately from remove-liquidity, which only takes single recipient)
        (token0::create-account pair-account-name (token-guard))
        (token1::create-account pair-account-name (token-guard))
        ;; Take ownership of feeTo account if not owned already.
        (exchange.rotate-fee-guard key (fee-guard))
        ;; Create pair record.
        (insert pair-table key
          { 'key: key
          , 'fee-account: fee-account
          , 'fee-guard: (fee-guard)
          , 'pair-account: pair-account-name
          , 'pair-guard: (token-guard)}))))
```

At the end of the execution, the `register-pair` function inserts a row in the `pair-table` which makes the pair in question known to the staking contract. Before that, it takes over the LP token's contract `fee-account` in order to be able to control accrued fees in the LP contract for that pair. It also creates accounts for the staking contract in the actual wrapped tokens; these are necessary since the staking contract needs a placeholder for tokens that the LP tokens yield via `remove-liquidity` on the LP fees. Finally, the tokens themselves need to be registered:

```
 (defun register-token-if-unregistered (token:module{fungible-v2} hint:string)
   "Internal function to register a given fungible-v2 token within the staking \
   \ contract. Not called directly; register-pair calls this function. If a \
   \ previously unknown token is passed to this function, a corresponding token-record \
   \ row is created, and two accounts are created with the same name: one with \
   \ the given token, and another with KDX. The token account is used to receive \
   \ remove-liquidity outputs when sweeping fees, and the KDX account is used \
   \ when swapping the collected token amounts into KDX."
   ;; require-capability to avoid operators calling this function on its own
   (require-capability (OPS))
   (let
     ( (token-name (format "{}" [token]))
       ;; Generate an opaque account name given the token name and hint string.
       (account-name (hash (format "{}_{}" [token hint]))))
     (with-default-read token-table token-name
       { 'account: "" }
       { 'account := current-account }
       (if (= current-account "") ;; Continue only if the token isn't already registered
        (let ((x 0)) ;; throwaway let for many-statement if clause
          (token::create-account account-name (token-guard)) ;; Create token account
          (if (!= token kdx) ;; If token isn't KDX, create KDX account with same name/guard.
            (kdx.create-account account-name (token-guard)) {})
          (write token-table token-name ;; Create the token record.
            { 'account: account-name, 'token: token, 'guard: (token-guard) }))
        {}))))
```

This function optionally writes to `token-table` which holds the account to which the tokens will be sent inside the `sweep-one` function. An account inside the KDX wrapped token is created; this is where the KDX that is swapped for the tokens will ultimately land, before it is transferred to `KDX_BANK` and sent to stakers.

8.3 – Staking and Unstaking Functions

The staking interface converts the KDX into sKDX and expands to pool in a staggered way:

```
(defun stake
   (from:string
    amount:decimal)
   "Request to add a certain amount of KDX to the stake pool. This function \
   \ immediately wraps the provided KDX to sKDX, and adds the requested amount \
   \ to the stake record's pending-add field. This amount isn't yet included \
   \ into the stake pool, see include-some and include-batch for when this queued \
   \ amount is actually included into the pool. \
   \ The queue system is to mitigate unfairness, as fee sweeping is a discrete \
   \ event, and letting people into the pool at any arbitrary period would let \
   \ them unfairly partake in fees accrued between the last fee sweep and their \
   \ entry time. With the queue, the operator includes waiting participants into \
   \ the pool right after a fee sweep."
   (with-capability (STAKE from amount)
     (enforce (> amount 0.0) "amount must be positive")
     ;; Wrap the provided KDX into a sKDX account belonging to the user.
     (alchemist.wrap amount 'skdx from from (get-kdx-guard from))
     ;; Write the stake-record. If no prior stake-record exists, create one
     ;; with default values.
```

```
      (with-default-read stake-table from
        { 'account: ""
        , 'amount: 0.0
        , 'last-claim: PAST_EPOCH
        , 'effective-start: PAST_EPOCH
        , 'last-stake: PAST_EPOCH
        , 'start-cumulative: 0.0
        , 'pending-add: 0.0
        , 'rollover: 0.0
        , 'locks: [] }
        { 'account := account
        , 'amount := prev-amount
        , 'rollover := prev-rollover
        , 'last-claim := last-claim
        , 'last-stake := last-stake
        , 'effective-start := effective-start
        , 'start-cumulative := start-cumulative
        , 'pending-add := prev-pending-add
        , 'locks := locks }
        (write stake-table from
         { 'account: from
         , 'amount: prev-amount
         , 'last-stake: last-stake
         , 'last-add-request: (at 'block-time (chain-data))
         , 'pending-add: (+ amount prev-pending-add)
         , 'effective-start: effective-start
         , 'start-cumulative: start-cumulative
         , 'rollover: prev-rollover
         , 'last-claim: last-claim
         , 'locks: locks }))))
```

A number of parameters in the table are required due to staggered pool inclusion (`last-stake`, `last-add-request`, `effective-start` and `start-cumulative`).

The interface for unstaking KDX is below:

```
  (defun unstake (account:string unstake-amount:decimal)
     "Unstake some amount from the given staking account. This will call rollup \
     \ in order to realize deserved rewards with the pre-unstake stake amount. \
     \ Partial unstaking is supported, and will reset effective-start to the current \
     \ block time. If the last stake add was less than PENALTY_PERIOD ago, apply \
     \ an unstake penalty of PENALTY_FRACTION."
     (with-capability (UNSTAKE account)
     ;; Since we will be changing the user's staked amount, ensure that their
     ;; deserved rewards are realized.
     (rollup account)
     (with-read stake-table account
       { 'amount := stake-amount
       , 'locks := locks
       , 'rollover := rollover
       , 'last-stake := last-stake
       , 'effective-start := effective-start
       , 'last-claim := last-claim }
       (let*
         ( (g (get-kdx-guard account))
           ;; The maximum KDX amount that can be unstaked given any stake locks.
           (available-amount (calculate-unlocked-stake stake-amount locks))
           ;; The current pool state.
```

```
          (state (read state-table STATE_KEY))
          ;; Current amount of staked KDX in the pool.
          (total-staked-kdx (at 'staked-kdx state))
          ;; Current block time.
          (now (at 'block-time (chain-data)))
          ;; Seconds passed since last stake add. Used to calculate whether the
          ;; user is within PENALTY_PERIOD.
          (seconds (diff-time now last-stake))
          ;; Unstake penalty amount to be incurred, if any.
          (stake-penalty (if (< seconds PENALTY_PERIOD) (floor (* unstake-amount
          ↪ PENALTY_FRACTION) (kdx.precision)) 0.0))
         ;; The net KDX amount to be transferred to the user.
          (net-unstake (- unstake-amount stake-penalty))
         )
         ;; The user must have enough unlocked staked KDX to unstake the requested amount.
         (enforce (>= available-amount unstake-amount)
          (format "Insufficient unlocked stake ({} available {} requested)" [available-amount
          ↪ unstake-amount]))
         ;; The user must request a positive unstake amount.
         (enforce (> unstake-amount 0.0) "Unstake amount must be positive")
         ;; If the user is incurring an unstake penalty, unwrap their penalty from
         ;; their sKDX account into the KDX_BANK KDX account and record it as burnt
         ;; in the pool state.
         (if (> stake-penalty 0.0)
          (with-capability (INTERNAL)
            (alchemist.unwrap stake-penalty 'skdx account KDX_BANK (kdx-guard))
            (record-burn 'stake account stake-penalty)) {})
         ;; If the user has any net KDX out, unwrap that amount from their sKDX
         ;; account into their KDX account.
         (if (> net-unstake 0.0)
          (alchemist.unwrap net-unstake 'skdx account account g) {})
         ;; Inform the aggregator of an unstake event. Commented in the REPL as
         ;; the aggregator isn't integrated into here.
         ;(kaddex.aggregator.aggregate-unstake account unstake-amount)
         ;; Reset the user's effective-start, and decrement their staked amount
         ;; by their requested unstake amount.
         (update stake-table account
          { 'amount: (- stake-amount unstake-amount)
          , 'effective-start: now })
         ;; Update the pool state to mark a decrease in total KDX staked.
         (update state-table STATE_KEY
          { 'staked-kdx: (- total-staked-kdx unstake-amount) })
         true))))
```

Unstaking starts with a rollover, which calculates how much the participant can claim before the unstaking took place. After applying penalties and sending them to the `KDX_BANK` account in the KDX contract, the user's desired amount of funds is sent to their KDX account.

Additional testing notes

This section lists some of the observations made during the second phase of the review, mostly focusing on the Wrapper.

`create-principal` not implemented in Pact version 4.2: While running the `wrapper.repl` unit tests with Pact 4.2.1, the Pact interpreter returned the following error:

```
<interactive>:1:0: Cannot resolve "create-principal"
 at : module
```

Upgrading to Pact 4.3 resolved the issue. The wrapper code enforces versions 3.7 and 4.2 in two different files (`tokens.pact` and `exchange.pact`). This was previously discussed in finding "Outdated Pact Dependency" and in the following comment in code:

```
;;; TODO: we probably want to add an (enforce-pact-version) call (on every file?)
```

In this observation it is noted that enforcing Pact 4.2 is not sufficient, version 4.3 should be used.

`add-liquidity` can add liquidity in ratio different than A:B: It is possible to send `A` tokens to the exchange's account in `tokenA` out of band. This will be picked up by `exchange.pact:add-liquidity` and, as such, liquidity can be added in arbitrary A:B ratios. This does not appear to lead to any direct security issues, but is worth documenting.

In more detail, the `add-liquidity` caller provides the desired and minimal token A and token B as arguments to the function. The contract then computes the actual amounts that will be sent and they respect the ongoing token ratio.

Later in the `add-liquidity` function, the *actual* increase in the exchange's accounts in token A and token B is computed:

```
      (let* ;; first we calculate the actual amounts transferred by calling `get-balance`
        ( (token0:module{fungible-v2} (at 'token (at 'leg0 p)))
          (token1:module{fungible-v2} (at 'token (at 'leg1 p)))
          (balance0 (token0::get-balance pair-account))
          (balance1 (token1::get-balance pair-account))
          (reserve0 (at 'reserve (at 'leg0 p)))
          (reserve1 (at 'reserve (at 'leg1 p)))
          (amount0 (- balance0 reserve0))
          (amount1 (- balance1 reserve1))
```

Right before calling `add-liquidity`, a user may fund the `pair-account` in token A. This will affect the LP amount calculation and the exchange's reserves will be updated in a ratio that is different from A:B.

Inconsistent usage of word "Internal" in comments: Comments for several functions in the Wrapper and the Staking contracts indicate that functions are *internal*. All functions described this way require capabilities and indeed are internal. There exists a minor deviation from this: `get-one-sided-liquidity-swap-amount` is described as internal, however, this function is read-only and does not require any capabilities.

`get-other-side-token-amount-after-swap` slippage parameter not validated: This function is used with `add-liquidity-one-sided` to account for slippage created by the swap. The `slippage` parameter is expected to be in a certain range, but this is not enforced:

```
(defun get-other-side-token-amount-after-swap:decimal
   ( amountA-total:decimal
     tokenA:module{fungible-v2}
     tokenB:module{fungible-v2}
     slippage:decimal ;; this value needs to be greater than or equal to 1
     ;; a value of 1.01 gives a 1% slippage
   )
   "Returns the tokenB amount to use for signing a TRANSFER capability when doing an `add-
   ↳ liquidity-one-sided` call."
```

In addition, comment clarity can be improved in this function, eg.:

```
     ;; multiply by the slippage in case the price changes between the user
     ;; querying this function and calling the function below
     (exchange.truncate tokenB (* slippage (at 'out (at 0 out-result))))
   )
 )
```

The comment refers to the function that’s currently just under the ongoing function, which could change in the future.

6 Finding Field Definitions

The following sections describe the risk rating and category assigned to issues NCC Group identified.

Risk Scale

NCC Group uses a composite risk score that takes into account the severity of the risk, application's exposure and user population, technical difficulty of exploitation, and other factors. The risk rating is NCC Group's recommended prioritization for addressing findings. Every organization has a different risk sensitivity, so to some extent these recommendations are more relative than absolute guidelines.

Overall Risk

Overall risk reflects NCC Group's estimation of the risk that a finding poses to the target system or systems. It takes into account the impact of the finding, the difficulty of exploitation, and any other relevant factors.

Rating	Description
Critical	Implies an immediate, easily accessible threat of total compromise.
High	Implies an immediate threat of system compromise, or an easily accessible threat of large-scale breach.
Medium	A difficult to exploit threat of large-scale breach, or easy compromise of a small portion of the application.
Low	Implies a relatively minor threat to the application.
Informational	No immediate threat to the application. May provide suggestions for application improvement, functional issues with the application, or conditions that could later lead to an exploitable finding.

Impact

Impact reflects the effects that successful exploitation has upon the target system or systems. It takes into account potential losses of confidentiality, integrity and availability, as well as potential reputational losses.

Rating	Description
High	Attackers can read or modify all data in a system, execute arbitrary code on the system, or escalate their privileges to superuser level.
Medium	Attackers can read or modify some unauthorized data on a system, deny access to that system, or gain significant internal technical information.
Low	Attackers can gain small amounts of unauthorized information or slightly degrade system performance. May have a negative public perception of security.

Exploitability

Exploitability reflects the ease with which attackers may exploit a finding. It takes into account the level of access required, availability of exploitation information, requirements relating to social engineering, race conditions, brute forcing, etc, and other impediments to exploitation.

Rating	Description
High	Attackers can unilaterally exploit the finding without special permissions or significant roadblocks.
Medium	

Rating	Description
	Attackers would need to leverage a third party, gain non-public information, exploit a race condition, already have privileged access, or otherwise overcome moderate hurdles in order to exploit the finding.
Low	Exploitation requires implausible social engineering, a difficult race condition, guessing difficult-to-guess data, or is otherwise unlikely.

Category

NCC Group categorizes findings based on the security area to which those findings belong. This can help organizations identify gaps in secure development, deployment, patching, etc.

Category Name	Description
Access Controls	Related to authorization of users, and assessment of rights.
Auditing and Logging	Related to auditing of actions, or logging of problems.
Authentication	Related to the identification of users.
Configuration	Related to security configurations of servers, devices, or software.
Cryptography	Related to mathematical protections for data.
Data Exposure	Related to unintended exposure of sensitive information.
Data Validation	Related to improper reliance on the structure or values of data.
Denial of Service	Related to causing system failure.
Error Reporting	Related to the reporting of error conditions in a secure fashion.
Patching	Related to keeping software up to date.
Session Management	Related to the identification of authenticated users.
Timing	Related to race conditions, locking, or order of operations.

7 Contact Info

The team from NCC Group has the following primary members:

- Eric Schorn – Technical Lead, Consultant
  eric.schorn@nccgroup.com
- Aleksandar Kircanski – Consultant
  aleksandar.kircanski@nccgroup.com
- Kevin Henry – Consultant
  kevin.henry@nccgroup.com
- Parnian Alimi – Consultant
  parnian.alimi@nccgroup.com
- Javed Samuel – Vice President, Cryptography Services Practice Director
  javed.samuel@nccgroup.com

The team from Kaddex, Inc. has the following primary members:

- Nicolas Ramsrud
  nicolas@kaddex.com
- Adrian Cardoso
  adrian@kaddex.com
- Daniele De Vecchis
  daniele@kaddex.com
- Giuseppe Pace
  giuseppe@kaddex.com
- Gustavo Spelzon
  gustavo@kaddex.com
- Kate Oztunc
  kate@kaddex.com