FlameDeFi DEX Smart Contract Review

October 28, 2021

Contents

1 Introduction

1.1 Context

This report synthesizes the conclusions and results of reviewing the DEX/Pool smart contracts provided by FlameDefi. This report was sent to the team on 26/10/2021. It consists of two main sections. The first one details our understanding of the smart contracts architecture and functionalities. The second one provides the list of defects we discovered, with their respective severity levels, and/or improvements we reported/suggested to the Client.

This version of the reviewed smart contracts consists of a zipped archive whose SHA256 sum hash is:

`19835340ce0e93bf6c5bb80de6d59a08927200f9261e8e2f054f7713dc0c6b24`

Some issues were fixed in subsequent newer versions with the following hashes (the last one being the latest):

- `950ddc81161510f44065f8fd584317b7705cc5dde845830bb966bcd4b4dffcbc`
- `ad81df6fa8b212067458375d0402633db7ebb1bc15b90da2fd3507f30cc610ef`

1.2 Allocated Resources for the Review

Functori is a team of IT experts with a background in programming languages, formal verification, and blockchain technology, especially in Tezos. The code review consists in reading the provided implementation, extracting a specification from the different functions and entrypoints, searching for faults, bugs, and attacks in the code, and challenging the specification to spot possible design issues. Our smart contracts reviewers experts ensure that the code is as safe as possible, but don't guarantee that all defects and bugs are actually found and reported.

The review team was composed of PhDs in programming languages design and formal methods. One DeFi expert in the team was also involved to help in the code review process.

2 The Smart Contracts

2.1 Overall Description

The FlameDeFi DEX under review is a set of smart contracts written in the PascaLigo language for the Tezos blockchain. It is built on top of Tezos tokens smart contracts standards, namely, FA1.2 and FA2.

DEXs (Decentralized Exchanges) are a kind of asset exchange that allows for direct cryptocurrency transactions to take place online securely and without the need for an intermediary.

The provided code consists of the following files:

- `UberPool.ligo.m4`: the main file of the DEX (with some preprocessing directives for testing)
- `CommonTypes.ligo`: contains the type `token_info` and a value `no_operations`
- `FA12Api-michelson.ligo`: code to make calls to the transfer entry point in FA1.2 contracts (with inlined Michelson)
- `FA2Api.ligo`: : code to make calls to the transfer entry point in FA1.2 contracts
- `FA2Errors.ligo`: constant string definitions for FA2 errors
- `NumUtils.ligo`: utility function for operations on natural numbers
- `StringUtils.ligo`: conversion functions for ASCII strings
- `TokenTransfers.ligo`: wrappers for token transfers abstracting over token type (FA2, FA1.2 and Tezos)

2.2 Key Concepts

We introduce important terms and concepts used in the rest of the document:

- **address**/account/wallet: any tz or KT address on the Tezos blockchain
- **user**: an entity (identified by its address) that interacts with the DEX
- **admin**: a particular address (privileged entity) set at the deploy time of the contract
- **tokens**: assets deployed on the Tezos blockchain (to be distinguished from the XTZ native token)
- **tokens transfer**: internal operations performed by the farm: transfer of tokens between users and the farm, depending on the called entrypoint of the contract
- **FA12**: it is a token standard on Tezos. It refers to an ERC20-like fungible token standard (TZIP-7) for Tezos, or to any contract following this standard
- **FA2**: FA2 refers to a token standard (TZIP-12) on Tezos. FA2 proposes a unified token contract interface, supporting a wide range of token types. The FA2 provides a standard API to transfer tokens, check token balances, manage operators (addresses that are permitted to transfer tokens on behalf of the token owner) and manage token metadata.
- **pool**: a pair of tokens
- **swap**: a functionality that allows a user to exchange an amount of a token A for equivalent amount of a token B without an intermediary
- **swap ratio**: the value of an amount of a token A in term of another token B

- **adding liquidity**: an operation from a user to add an amount of a token A and its equivalent value in a token B, in a pool
- **removing liquidity**: an operation from a user to withdraw (some of) his tokens A and their equivalent value in a token B, from a pool
- **liquidity provider**: a user who provides liquidity to a pool
- **swapper**: a user who swaps an amount of his tokens A for an equivalent amount (in value) of tokens B
- **bucket**: in the context of this review, this term is sometimes used for a pool (a tokens pair), to follow the name of types and fields used in the code.

3 Detailed Analysis of the Smart Contract Capabilities

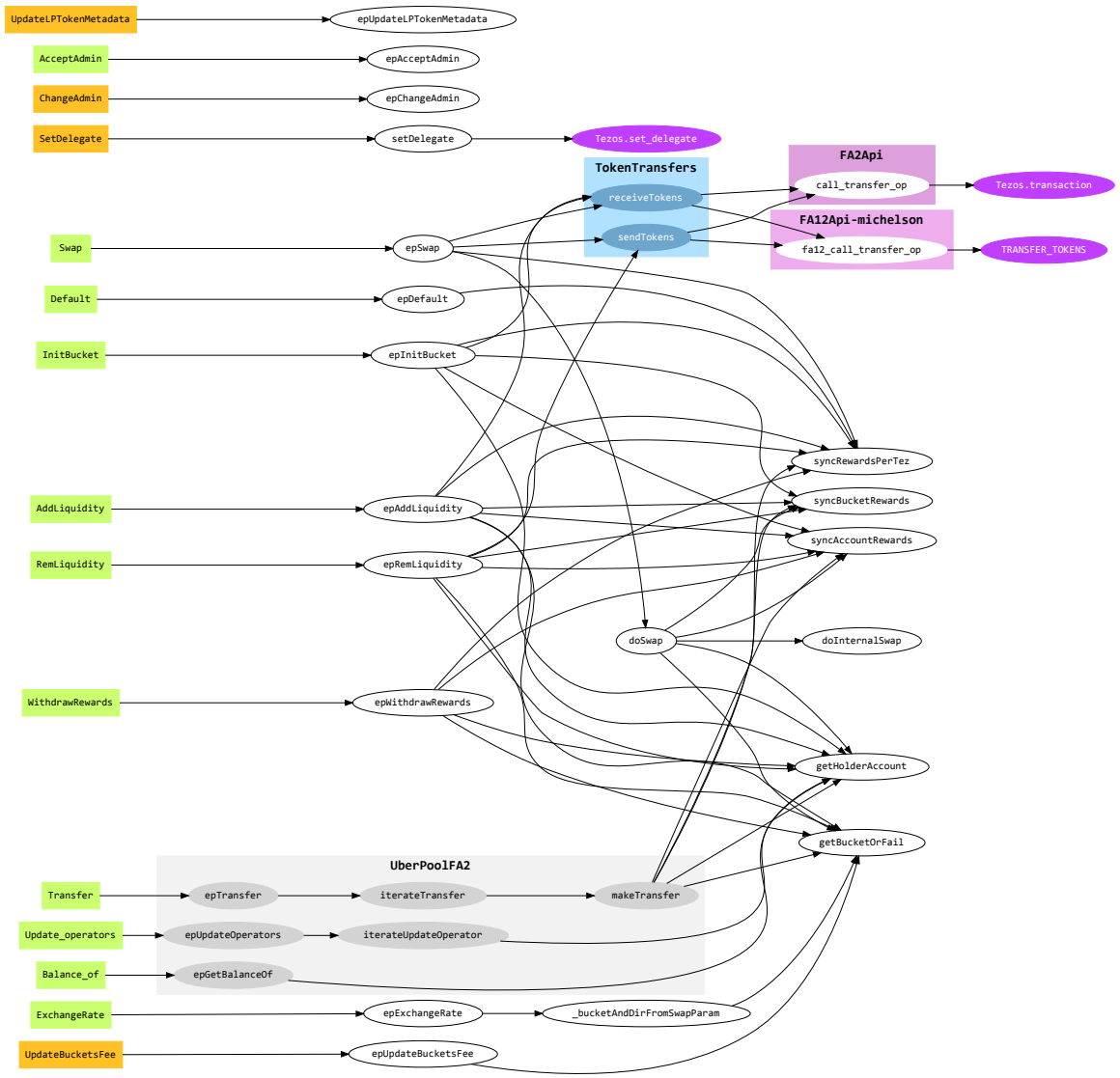

Figure 1: FlameDefi DEX dependencies

The figure above shows the entry points of the UberPool smart contract with the call graph dependencies (without utility functions). In green are depicted *public* entry points and in orange the ones that can only be called by an administrator of the contact. The purple nodes indicate the builtin Tezos functions which emit operations (transfer or set delegate, here).

We can see that the most complex entry point is `swap` (followed by `removeLiquidity` and `addLiq-`

uidity).

3.1 Swap Formula

The most important feature of the contract is the computation of the amount of tokens that will be swapped. All the swapping logic is implemented in the swap entrypoint: assume the pool has a token amount A and a token amount B, the entrypoint always keeps the following ratio during swaps:

$$A \times B = k$$

To compute the amount of "out" tokens one would get during the swap, the smart contract implements the following logic: if a user wants to swap a certain amount b_in of B token for an equivalent amount of a_out of A token, initially the ratio is:

$$A \times B = k$$

After the swap, the a_{out} amount is computed in such way to keep the same ratio:

$$\begin{aligned} (A - a_{out})\,(B + b_{in}) &= A \times B = k \\ A \times B + A \times b_{in} - a_{out}\,(B + b_{in}) &= A \times B \\ A \times b_{in} &= a_{out}\,(B + b_{in}) \\ a_{out} &= \frac{A \times b_{in}}{B + b_{in}} \end{aligned}$$

The smart contract charges a parametric fee fee (by default 0.3%) on the input tokens. In the default case, the amount of input tokens considered during the swap becomes:

$$\begin{aligned} b_{in} &= b_{in-with-fees} - 0.3\%\, b_{in-with-fees} \\ &= \frac{997\, b_{in-with-fees}}{1000} \end{aligned}$$

After mathematical simplification, the final formula (the one implemented in the smart contract) is:

$$a_{out} = \frac{997\, A \times b_{in-with-fees}}{1000\, B + 997\, b_{in-with-fees}}$$

3.2 Tezos rewards and bookkeeping

A new capability was added since our first review of the FlameDefi Dex contract: Tezos/Token pairs, when previously only Token/Token pairs were allowed. All the `tez` stored inside a given bucket is delegated to a baker, and the baking rewards are distributed among the liquidity providers, taking into account their share of the liquidity and the time elapsed since their share last changed, so as to adjust their rewards. This, if handled improperly, might lead to constant updating of the whole database of each Tezos/Token bucket. The chosen technical solution minimizes bookkeeping operations, while keeping them accurate. We clarify in this section how the solution works.

There are three levels of bookkeeping, where a lower level inherits from the upper ones. Whenever bookkeeping happens at a level, it means it was preceded, in the same transaction (and thus at the same time) with an update to the upper levels.

In the following, we will follow a few conventions to facilitate exposition. Whenever we reason about a smart contract's value at times t_i and t_{i+1}, we assume:

- that t_i and t_{i+1} refer to times when this value was updated in the smart contract;
- that there was no update strictly between t_i and t_{i+1}, meaning that t_{i+1} is the 'time of first update' of the value we are considering since t_i.

Whenever there is a mention of updating a field, it is important to recall that updates are done *before* any balance (in `tez` or in liquidity shares) has been updated. This invariant holds in the code.

3.2.1 At the storage level

At the highest level, the smart contract storage level, three variables are maintained for our purpose:

- `tez_total` is the total amount of `tez` currently deposited into the smart contract. It can be increased by adding liquidity or performing a swap, or by receiving rewards from the baker through the `epDefault` entrypoint. It can be decreased by removing liquidity or doing a swap, as well as by claiming one's rewards.
- `rewards_per_sec` represents the rewards per second of the contract until the end of the next period (or until it is updated within the period). Another formulation would be the rate at which rewards (more precisely, those currently accounted for) would be accumulated if distributed evenly in the time remaining until the next period. It is reinitialized after the end of a period (a new period may only start when a delegation reward is deposited).

  Here is how it is updated:

$$\texttt{rewards\_per\_sec}(t_2) = \texttt{rewards\_per\_sec}(t_1) + \frac{\texttt{amt}}{\texttt{period\_end} - t_2} \tag{1}$$

Note that if we are starting a new period, then $t_1 = t_2$ and before the update, $\texttt{rewards\_per\_sec}(t_1) = 0$.

Inside a period, the *total* rewards accumulated between t_1 and t_2 can be computed as:

$$rewards[t_1, t_2] = (t_2 - t_1)\,\texttt{rewards\_per\_sec}(t_1) \tag{2}$$

- `rewards_per_tez` is the *cumulated* amount (in `tez`) of the rewards to which each `tez` entitles its owner. This means that this value only makes sense *differentially*. It is an increasing function of time. Anticipating a bit on what follows, and in order to leave no mystery here, suppose a bucket `B` (Tezos/Token) has had a fixed balance of `balance(t_1)` $=1000$ `tez` since time t_1. We want to compute what amount of `tez` baking rewards it has accrued for its liquidity providers until time t_1. The bucket `B` is entitled to :

$$reward_B[t_1, t_2] = (\texttt{rewards\_per\_tez}(t_2) - \texttt{rewards\_per\_tez}(t_1)) \times balance(t_1) \tag{3}$$

Note that this `rewards_per_tez` value is updated whenever the contract receives `tez`, and that it does so by anticipating the rewards which will be distributed at the end of a given (two-week) period. The value of `rewards_per_tez` is updated[1] at time $t_2 > t_1$ in the following way:

$$\texttt{rewards\_per\_tez}(t_2) = \texttt{rewards\_per\_tez}(t_1) + \frac{\texttt{rewards\_per\_sec}(t_1)\,(t_2 - t_1)}{\texttt{tez\_total}(t_1)} \tag{4}$$

We can make sense of this formula. Assume that at time t_1, $\texttt{rewards\_per\_tez}(t_1)$ was correctly computed. Per (2), $\texttt{rewards\_per\_sec}(t_1)\,(t_2 - t_1)$ is the amount of rewards accrued for the whole balance $\texttt{tez\_total}(t_1)$ during the period $[t_1, t_2]$. Thus dividing by $\texttt{tez\_total}(t_1)$ we get a value "per `tez`".

- `last_update_time` represents the last time the above storage variables were updated.

3.2.2 The bucket level

Let's go down one level, that of a bucket `B` holding a pair between Tezos and a token. There may be various buckets of this type. Each bucket maintains it own set of variables for bookkeeping:

[1] Updated for the first time since t_1.

- B.$total\_supply(t)$ is the total amount of shares at time `t`.
- B.$last\_rewards\_per\_tez(t)$: The value of the `rewards_per_tez` variable at the last time when the bucket's `tez` holdings changed. Thus we always have for t_1, t_2 following our convention:

$$\text{B}.last\_rewards\_per\_tez(t_2) = \texttt{rewards\_per\_tez}(t_1) \tag{5}$$

- B.$rewards\_per\_share(t)$: the *cumulated* rewards accrued by shares at time `t`. Much like `rewards_per_tez` above, this value only makes sense differentially. The rewards accrued by a share between t_1 and t_2, for a given `balance(t_1)`, can be computed as:

$$\begin{aligned}&\text{B}.rewards\_per\_share[t_1, t_2] = \\ &(\text{B}.rewards\_per\_share(t_2) - \text{B}.rewards\_per\_share(t_1)) \times balance(t_1)\end{aligned} \tag{6}$$

This value is updated in the following way:

$$\begin{aligned}&\text{B}.rewards\_per\_share(t_2) = \\ &\text{B}.rewards\_per\_share(t_1) + \frac{(\texttt{rewards\_per\_tez}(t_2) - \text{B}.last\_rewards\_per\_tez)\ \texttt{B.token\_a\_res}}{\text{B}.total\_supply}\end{aligned} \tag{7}$$

Remember that `B.token_a_res` is actually a sum in `tez`, because the first token in a pair with `tez` is always XTZ. And thus in (7), substituting (5), we recognize the formula (3) and we get

$$\text{B}.rewards\_per\_share(t_2) = \text{B}.rewards\_per\_share(t_1) + \frac{reward_B[t_1, t_2]}{\text{B}.total\_supply}$$

which immediately makes sense: by dividing by the total supply, we get the rewards "per share".

3.2.3 The account level

The final level is the actual account level, the one which a user is most interested in. The above serves the only purpose of computing, when necessary an only then, the reward increase since the last update for an account `Acc`. Note that an account is specific to a bucket. Two variables are maintained:

- `Acc`.$rewards$ are the rewards computed at the last update of the account;
- `Acc`.$balance(t)$ is the balance of the account `Acc` in liquidity shares.
- `Acc`.$last\_rewards\_per\_share$, much like what we have seen previously, contains at time t_2 the value of B.$rewards\_per\_share(t_1)$

Then the update is as follows:

$$\begin{aligned}&\texttt{Acc}.rewards(t_2) :=\\&\quad \texttt{Acc}.rewards(t_1) + \texttt{Acc}.balance(t_1) * (\texttt{B}.rewards\_per\_share(t_2) - \texttt{Acc}.last\_rewards\_per\_share(t_2))\\&\quad = \texttt{Acc}.rewards(t_1) + \texttt{Acc}.balance(t_1) * (\texttt{B}.rewards\_per\_share(t_2) - \texttt{B}.rewards\_per\_share(t_1))\end{aligned}$$

And applying formula (6) we see that this is in fact the accumulated reward for balance $\texttt{Acc}.balance(t_1)$.

4 Discovered Defects and Suggested Improvements

In addition to trying to infer the specification and compare it to the state of the art DeFi projects, we attempted to find flaws in the following items during the review:

- arithmetic overflows
- storage leak
- unauthorized calls/spends
- taking ownership of the contract(s)
- blocking funds on the contract(s)
- dead/duplicate code, code quality
- reentrancy/external interactions

We classify the remarks and observations below in decreasing severity level defined as:

Severity	Description
Critical	Loss (or unexpected creation) of tokens, deadlock (or livelock) of the smart contract. The contract is unusable and/or its logic is broken.
High	Unexpected failures or behaviors that can be acted upon. Some design choices or some flaws in the implementation may allow malicious users to break some invariants or get extra benefits, and cause assets loss for regular users if they accidentally miss-interact with the contract.

Severity	Description
Moderate	Confusing code (e.g. dead code), moderate inefficiencies, or code that can be significantly improved. The logic of the smart contract is not impacted, but the users who face these issues may be quite annoyed.
Low	Code improvements for legibility and maintainability. Improvement of users experience and the product's interface/usability
Suggestion	Software engineering suggestions and best practices.

All recommendations in this review should not be considered a guaranteed and complete fix, but merely a partial direction to explore with all due precautions.
Please note that in this ranking, we do not rely on the probabilities of defects, only on their severity.

For the review, we have expanded the M4 macros in `UberPool.ligo` as if `TEST` was `false`, and only considered the contracts with this configuration, as we presume that this is how they will be deployed.

4.1 High

4.1.1 Avoid using sets for allowances (fixed)

High

Fixed in `950ddc81161510f44065f8fd584317b7705cc5dde845830bb966bcd4b4dffcbc`

Unlike big_map, set and map data structures' content is fully deserialized when the structure is accessed. This results in a higher paid deserialization gas fee. The deserialization gas can become larger than the maximum authorized in a Tezos operation if a user if keeps allowing addresses without removing them afterwards. When this happens, the user will not be able to perform any action on this contract

(excepted for swaps as almost all functions read the big map `shares` with `getHolderAccount`). If this problem happens with the admin account, then no one will be able to swap. Note however that a too large allowance set for an account has no impact (unless it is the admin account) on the ability of other accounts to perform operations normally.

Relevant code

In file `UberPool.ligo.m4`, line 46.

Comments

Yet, putting a `big_map` here is not possible because data of type `holder` will themselves be put in a big_map (nested big_maps is not allowed in Michelson). An alternative is to have a global allowances big map in the storage indexed by the holder, the "allowee", and the bucket id, *i.e.* from `(src:address * dest:address * bucket_id:nat)` to `unit`.

4.2 Low

4.2.1 Use option types instead of giving special meaning to a value

Low

Here the value `0n` is used to mean that a bucket was not found, which was already signified by having `s.info_to_id[ti]` equal to `None`. Immediately afterwards in the code, the test `bucket_id = 0n` is (repeatedly) done; it would be simpler and safer to just keep the option value and match against it.

Relevant code

In file `UberPool.ligo.m4`, line 148 to 151.

4.2.2 Metadata of already existing bucket is overwritten (fixed)

Low

Fixed in `ad81df6fa8b212067458375d0402633db7ebb1bc15b90da2fd3507f30cc610ef`.

If a bucket has already been initialized (but the total supply went to 0), the call to `initBucket` will overwrite the name of the bucket in the `token_metadata` table with a generic one.

Relevant code

In file `UberPool.ligo.m4`, line 213 to 222.

Comments

This is problematic if an admin had changed the name of the bucket previously. Consider not changing the metadata if the bucket already existed.

4.2.3 Functions that return multiple values should use records instead of tuples *(1)*

Low

Some functions encapsulate multiple return values in tuple. This leads to identifying the values at the call site with index numbers (such as `res.0`, `res.1`, *etc.*) and make the code confusing and prone to mistakes.

Instead of returning tuples and using numbers to identify them in the callers' function, use a record instead and name each field. That facilitates the global understanding of the code and limits the issue of confusing two values of the same type.

Relevant code

In file `UberPool.ligo.m4`, line 378.

Comments

Consider defining a record type for the result of `doInternalSwap` such as the following.

```
type doInternalSwap_result is record [
  param : directional_swap_params;
  out_amt : nat;
]
```

4.2.4 Admin fees are in the order of 0.1% instead of the intended 0.05% (fixed)

Low

Fixed in `950ddc81161510f44065f8fd584317b7705cc5dde845830bb966bcd4b4dffcbc`

Admin fees in LPTokens represent both tokens A and tokens B shares in the pool (in proportions). This means that removing liquidity from the pool will return tokens A and tokens B in equal value (wrt. the

pool). However the swap fees are taken at swap time in tokens A. The computations done here thus gives rights to twice the expected fee because the admin will also get tokens B.

Relevant code

- In file `UberPool.ligo.m4`, line 407.
- In file `UberPool.ligo.m4`, line 411.

Comment

To have fees closer to 0.05% we would recommend to at least divide the computed `admin_fee` by 2. But to better account for extreme values, we want the subsequent *increase* in LPTokens to account for 0.05% of `amt_a` in *value*. And because the admin fees in LPTokens give rights (to remove liquidity) in both tokens A and tokens B (in equal values wrt. the reserves in the pool), they should account for half the fees in A (as the other half will be in tokens B).

This means that we need to resolve this equation:

$$\frac{fee_{admin}}{\texttt{total\_supply} + fee_{admin}} = 0.05\% \times \frac{\texttt{amt}_a}{2\, reserve_A}$$

which yields:

$$fee_{admin} = \frac{\texttt{total\_supply} \times \texttt{amt}_a}{4000\, reserve_A - 2\, \texttt{amt}_a}$$

4.2.5 Functions that return multiple values should use records instead of tuples *(2)*

Low

Some functions encapsulate multiple return values in tuple. This leads to identifying the values at the call site with index numbers (such as `res.0`, `res.1`, *etc.*) and make the code confusing and prone to mistakes.

Instead of returning tuples and using numbers to identify them in the callers' function, use a record instead and name each field. That facilitates the global understanding of the code and limits the issue of confusing two values of the same type.

Relevant code

- In file `UberPool.ligo.m4`, line 426.
- In file `UberPool.ligo.m4`, line 445 to 446.

Comments

Consider defining a record type for the result of `doSwap` such as the following.

```
type doSwap_result is record [
  storage : storage;
  out_amt : nat;
  param : directional_swap_params;
]
```

4.2.6 Avoid using inlined Michelson when possible

Low

When possible, it is better to avoid using inlined Michelson code in Ligo function. Michelson code is not typechecked by the Ligo compiler and is inserted "as is" in the generated output. Michelson code is typechecked by the Tezos node at deployment time so this could catch some bugs but it is dangerous and makes the code obscure.

Relevant code

In file `FA12Api-michelson.ligo`, line 5 to 24.

Comments

Ligo allows to choose which annotations will be generated in the Michelson output, if it is possible, use them to keep the guarantees offered by having structured types and *readable* code in the smart contract.

Here we assume that the aim of this inlined Michelson code is to avoid reserved keyword `to`. The purpose of `[@layout:comb]` is to make the Michelson datatype into a right-balanced tree (aka comb) with the order of fields preserved (otherwise it would be alphabetic, not in the `annot` fields but in the actual ligo record fields).

The following ligo type will also be compiled to michelson as `pair (address %from) (pair (address %to) (nat %value))`:

```
type fa12_transfer_param is [@layout:comb] record [
    from : address;
    [@annot:to] to_ : address;
    value : nat;
]
```

(see the ligo documentation)

4.2.7 Dead code (fixed)

Low

Fixed in `950ddc81161510f44065f8fd584317b7705cc5dde845830bb966bcd4b4dffcbc`

`token_addr_id` is not used anywhere so it can be removed safely.

Relevant code

In file `CommonTypes.ligo`, line 1.

4.2.8 Potential zero token transfer

Low

Transfering zero tokens in a FA2 or FA1.2 token contract is not needed in general.

Relevant code

- In file `TokenTransfers.ligo`, line 7.
- In file `TokenTransfers.ligo`, line 8.
- In file `TokenTransfers.ligo`, line 9.
- In file `TokenTransfers.ligo`, line 14.
- In file `TokenTransfers.ligo`, line 15.

Comments

Making internal calls can add significantly to the gas used (and so can impact the fee) of a transaction, so, when possible, it is a good practice to not make any unneeded calls. We would suggest to not make the call (*i.e.* not adding it to the list `ops`) when `amt` is zero, unless it is absolutely needed by the FlameDEX backend or frontend.

4.3 Suggestions

4.3.1 Name of function `assertTezosSent` is a bit misleading

Suggestion

The name of the function `assertTezosSent` implies that it checks that Tezos tokens were sent.

Relevant code

In file `UberPool.ligo.m4`, line 99.

Comments

Because this functions checks if sending Tezos tokens is allowed in the call, maybe call it `checkAllowedTezosSent(const from_token: option(token_info))`.

4.3.2 Consider checking that the pretended `amt_a` is `Tezos.amount` for Tz pool (fixed) Suggestion

Fixed in `950ddc81161510f44065f8fd584317b7705cc5dde845830bb966bcd4b4dffcbc`

When `ti.token_a` is `Tz` the `amt_a` parameter is thrown away (and replaced by the correct actual `Tezos.amount`). This means a user can make a mistake and send the wrong amount with his or her transaction, without warning.

Relevant code

- In file `UberPool.ligo.m4`, line 143.
- In file `UberPool.ligo.m4`, line 257.

4.3.3 Normalize pair by sorting Suggestion

Instead of checking if there already exists a bucket for the pair (B, A), maybe consider swapping A and B at the beginning if B < A.

Relevant code

In file `UberPool.ligo.m4`, line 153 to 165.

Comments

This will guarantee the invariant that there is only one representation for a token pair, and so there will be only one bucket for such a pair.

4.3.4 Consider allowing to receive 0 tokens from one of the reserves when removing liquidity

Suggestion

As is, if the reserves of one of the tokens is low, a user is prevented from withdrawing his share in the other reserve. Maybe the implemented behavior is what is intended (as per the comment in the quoted code).

Relevant code

- In file `UberPool.ligo.m4`, line 330.
- In file `UberPool.ligo.m4`, line 325.

Comments

Consider checking if `amt_a = 0 or amt_b = 0` (can be done with `(amt_a + amt_b) * amt_shares = 0n` if you really want to use arithmetic). Please note that in Uniswap(V2), this problem cannot happen because a minimim amount of 1000 shares has to added first in a new bucket.

4.3.5 The way reserves can be affected by a swap is hardcoded

Suggestion

A swap cannot take more than one third of the reserve. This value is hardcoded in the contract. Maybe it would be better to have this (optionnal) value be a parameter of the bucket.

Relevant code

In file `UberPool.ligo.m4`, line 374.

4.3.6 Rename wrong route to empty route

Suggestion

The only way to have a wrong route is for it to be empty, so we would suggest to rename the error to account for it.

Relevant code

In file `UberPool.ligo.m4`, line 476.

4.3.7 The contract allows to change decimals information

Suggestion

This is not an issue with the contract, but allowing an administrator to change the decimals for one or some of the LPTokens can lead to issues in the interface.

Relevant code

In file `UberPool.ligo.m4`, line 515.

Comments

For instance, a user could see his or her LPTokens balance (say for one bucket), drop by a factor of 1000 on the web platform.

4.3.8 Useless subtraction

Suggestion

`s.rewards_period_end` is updated just before, so this subtraction is always equal to `reward_period_const`.

Relevant code

In file `UberPool.ligo.m4`, line 606.

Comment

Using `reward_period_const` directly allows to remove on arithmetic operation and a call to `natOrZero`.

5 Informational Remarks

5.0.1 Rewards are actually in unit: 10^{-15} per mutez

Info

Rewards are not expressed in tez but rather are expressed in mutez amplified by `precision_const` = 10^{15}. They also do not represent an actual reward amount (per tez) but rather a cumulated value (or an integral).

Relevant code

- In file `UberPool.ligo.m4`, line 63.
- In file `UberPool.ligo.m4`, line 90.

6 Conclusion

During the review, we analyzed the smart contract(s) code, extracted a specification, and discovered 0 critical, 1 high, 0 moderate, and 8 low severity bugs. We also made 8 suggestions to improve the code as well as 1 informational remark.

All those issues were sent to the corresponding authors.