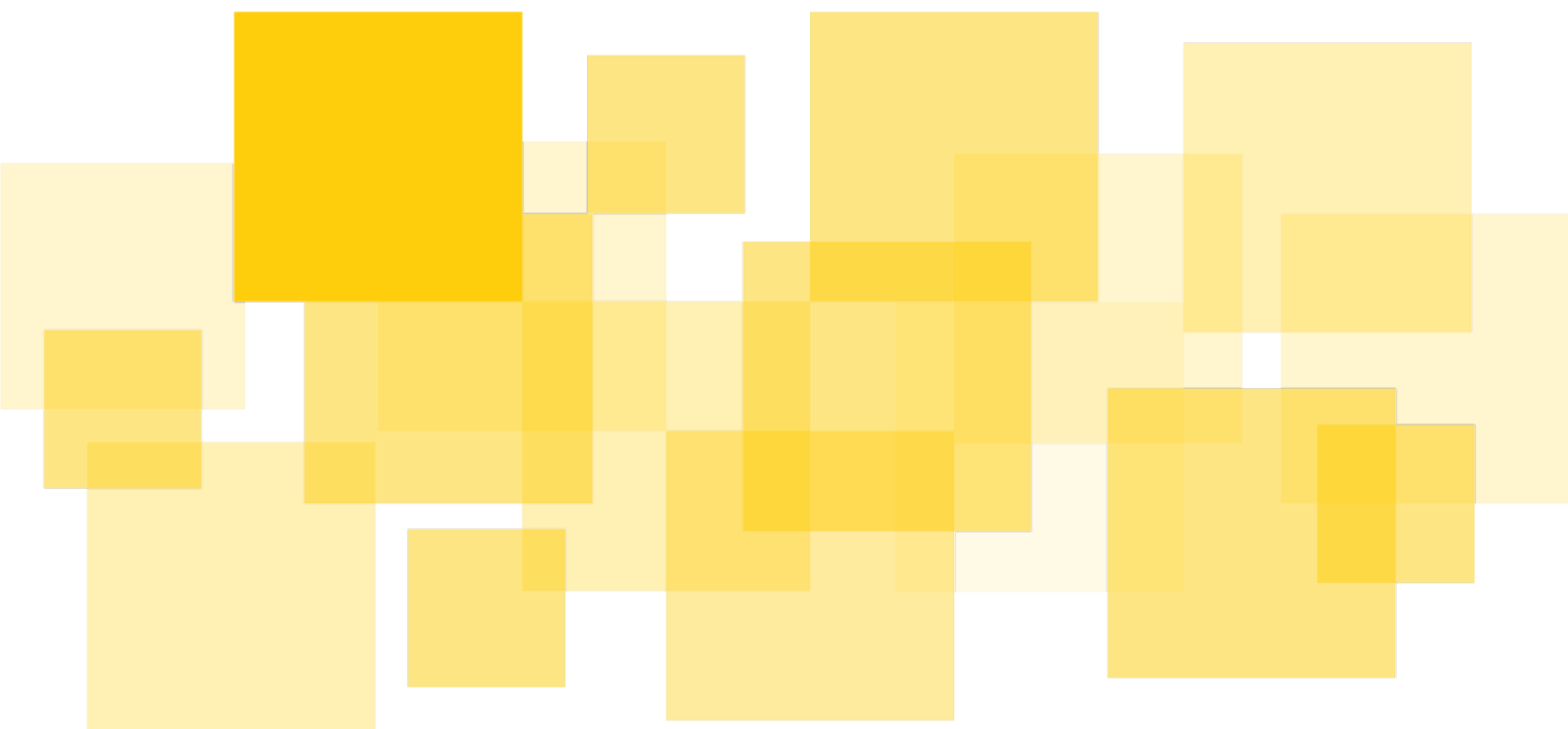


Security Audit Report

Quipuswap TTDex

Delivered: 2021-11-05



Prepared for the Madfish Solutions Inc. by



Contents

Summary	2
Scope	2
Assumptions	2
Methodology	2
Disclaimer	3
Findings	3
Initialization (Race Condition)	3
Re-entrancy Guard Failure (Potential Safety Violation)	4
Unspecified Operation Delay (Safety Violation)	5
Contract Tez Acceptance (Potential Safety Violation)	6
Informative Findings	6
Liquidity Share Token Implementation FA2 Standard Conformance	6
Swap Entrypoint Enables Flash Loans	6
Subtractions Implicitly Guarded Against Underflow	7
Redundant Divestment Deadlock Guard	8
Optimization	9
Optimizing <code>update_operators</code> code size	9
Potential readability and gas optimization	9
Appendix	10

Summary

[Runtime Verification Inc.](#) conducted a security audit on the Quipuswap token-to-token distributed exchange (TTDex) contract developed by Madfish Solutions, Inc. The audit was conducted from October 4, 2021 to November 5, 2021. During the course of the audit, there were four findings, four informative findings, and two optimization suggestions.

Scope

The target of the audit is the smart contract source files located in the [Quipuswap Github repository](#) at commit id [aa3d6b7](#). In particular, the source files included:

- `contracts/main/Dex.ligo`
- `contracts/partials/Dex.ligo`
- `contracts/partials/Errors.ligo`
- `contracts/partials/IDex.ligo`
- `contracts/partials/MethodsDex.ligo`
- `contracts/partials/MethodsFA2.ligo`
- `contracts/partials/TypesFA2.ligo`
- `contracts/partials/Utils.ligo`

The audit is limited in scope to the LIGO smart contract source code only. Off-chain and client side portions of the codebase as well as deployment and upgrade scripts are *not* in the scope of this engagement, but are assumed to be correct.

Assumptions

Our accompanying appendix available at our [publications repo](#) contains all important assumptions and proofs. We review the most important assumptions below:

1. a contract's storage can only be updated invoking that contract
2. the evaluation order of operations is depth-first, i.e., the operation call graph uses depth-first search (DFS)
3. all FA1.2/FA2 contracts that TTDex interacts with properly conform to the standard
4. each of TTDex's FA1.2/FA2 token balances cannot be *decreased* except via calling the token contract's `transfer` function from the TTDex contract

Methodology

Although the manual code review cannot guarantee that all potential security vulnerabilities are found (as mentioned in the Disclaimer), we have adhered to the following methodology to make our audit as rigorous as possible. First, we rigorously reasoned about the business logic of the contract, validating security-critical properties to ensure the absence of loopholes in the business logic and/or inconsistency between the logic and the implementation. Then we carefully checked if the code is vulnerable to [known security issues and attack vectors](#).

In order to simplify our reasoning, our security property validation proceeded in two phases. By observing that each entrypoint is (essentially) independent of the others (and any shared code between can be duplicated so that they are totally independent), we performed a two phase analysis as follows:

1. discover the semantic effect of each independent entrypoint (i.e. perform a pre/post-condition analysis for each entrypoint)
2. consider the combined effects of any possible chains of calls using our entrypoint abstractions from phase (1)

In phase (1), we may already be able to find bugs if the code of an individual entrypoint has flaws. In phase (2), we will check that the relevant security properties hold after multiple contract calls.

Disclaimer

This report does not constitute legal or investment advice. The preparers of this report present it as an informational exercise documenting the due diligence involved in the secure development of the target contract only, and make no material claims or guarantees concerning the contract's operation post-deployment. The preparers of this report assume no liability for any and all potential consequences of the deployment or use of this contract.

Smart contracts are still a nascent software arena, and their deployment and public offering carries substantial risk. This report makes no claims that its analysis is fully comprehensive, and the preparers of this report note that it excludes a number of components critical to the operation of this system.

The possibility of human error in the manual review process is very real, and we recommend seeking multiple independent opinions on any claims which impact a large quantity of funds.

Findings

Initialization (Race Condition)

There is a potential race issue when initializing the contract. This is because, after the contract is originated, anyone may call the `set_token/dex_function` entrypoints to initialize the token/dex functions. To avoid this issue, one would need to atomically originate the contract and call its `set_token/dex_function` entrypoints. Note that this is an easily detectable attack, but, if it is not properly detected could have serious consequences including arbitrary code execution.

Tezos nodes provide an RPC to create a single operation which batches multiple operations together, which provides a means to atomically execute all operations in the batch. However, as far as we are aware, it is not possible to batch a contract origination and then multiple calls to the originated contract since, before the contract is originated, the contract address will be unknown.

Recommendation: Assuming our analysis of the Tezos batch functionality is correct, there are several mitigation strategies:

1. Ideally, if the gas cost is not too high, the contract can be originated with its storage fully initialized, avoiding all race issues.
2. Originate the contract in one transaction and then gather all of the `set_token/dex_function` calls into one transaction batch. In this scheme, if the transaction batch fails to apply, then someone has tampered with the contract and the contract address should be abandoned.
3. Add an `admin` field to the storage which authenticates `set_token/dex_function` calls against `Tezos.sender`.

All schemes provide application security, but scheme (2) cannot fully defend the application deployment from griefers whose sole goal is to waste time/money.

Response: Madfish Solutions applied strategy (1) to resolve this issue.

Re-entrancy Guard Failure (Potential Safety Violation)

NOTE: this issue was originally discovered and fixed by Madfish Solutions, independently of the Runtime Verification audit, and is reported here for completeness.

The `use` entrypoint asserts that the `entered` value in the storage is `false` and then sets it to `true`. The `entered` variable can only be reset to `false` by the `close` entrypoint. Together, these two entrypoints enforce a re-entrancy guard. This mechanism was designed so that the `use` entrypoint's *last* emitted operation calls `close` (thus preventing intermediate operations from re-entering the `use` entrypoint). However, the `close` entrypoint is currently called by the *first* operation emitted by `use`, thus negating the utility of the re-entrancy guard.

Response: In this case, Madfish Solutions realized that the `close` entrypoint was the first emitted operation instead of the last emitted operation. Madfish Solutions fixed this issue by properly emitting the `close` entrypoint last.

Recommendation: More generally, we have three recommendations:

1. All implemented features (including re-entrancy guards) should be thoroughly tested (as was done here).
2. In particular, transaction ordering issues should be thoroughly examined, as they can easily lead to security issues.
3. The construction of cons lists should be done carefully, as developers may infer an item ordering that is the reverse of what they expect (see description below).

In Tezos, each entrypoint returns a cons list of operations to emit, but the inferred ordering of emitted operations may vary depend on how a list is built. In fact, the re-entrancy problem in the original TTDex code stemmed from exactly this representational issue. We illustrate the potential issue using LIGO Pascal syntax:

Pattern A:

Suppose we return the following operation list:

```
const operations : list operation := operation1 # operation2 # operation3 # list
[];
```

then `operation1` will be executed first while `operation3` will be executed last, agreeing with the visual order.

Pattern B:

Alternatively, we may construct and return an operation list as follows:

```
var operations : list operation := list[];
operations := operation1 # operations;
operations := operation2 # operations;
operations := operation3 # operations;
```

In the above case, the `operation3` will be executed first while `operation1` will be executed last, which disagrees with the visual ordering.

Unspecified Operation Delay (Safety Violation)

Operations that involve a moving exchange rate require time-shift security so that an operation that was intended to occur at time t cannot be indefinitely delayed and then applied at time $t+k$ for large values of k . Note that, there currently is a built-in transaction expiration mechanism, via the `branch` field of Tezos operations. For a transaction to be valid for inclusion in an upcoming, two requirements on the `branch` field must be met:

1. it must contain a valid block hash of an ancestor of the current blockchain head;
2. the difference between the level of the referenced ancestor block and the current blockchain head must be less than some protocol limit.

According to our research, the protocol limit is currently 60, but it may have changed. This means that a transaction will be valid for approximately 1 hour before automatically expiring. Since a trader could be expected to perform several trades in a 1 hour window, this hour expiration time may not be tight enough for all applications. Furthermore, since protocol-defined constants are subject to change via protocol upgrades, there is no guarantee that the current transaction-expiration behavior will continue.

Recommendation: Note, even if the blockchain now has a very short transaction delay with high probability, as the blockchain usage increases over time, the transaction delay will likely increase. As a defensive measure, against increasing blockchain popularity and protocol updates, we recommend that:

- deadline `timestamps` be included in each `use` input (since these are the only ones which have time-dependent results)
- logic be added to the `use` entrypoint to reject operations whose `deadline` has passed

Response: Madfish Solutions added deadlines for the liquidity investment and divestment operations.

Contract Tez Acceptance (Potential Safety Violation)

The contract does not reject transactions which give it Tez. A naive or malicious user or improperly implemented interface could send their own or others Tez to the contract, locking it up forever.

Recommendation: Add a check to reject all transactions which have Tez.

Informative Findings

Liquidity Share Token Implementation FA2 Standard Conformance

From our reading, the TTDex implementations of `transfer` and `balance_of` implementations do not fully conform to the FA2 standard. By the standard, both functions are required to implement the following check:

If one of the specified `token_ids` is not defined within the FA2 contract, the entrypoint MUST fail with the error mnemonic “FA2_TOKEN_UNDEFINED”.

Whether this violates the standard depends on what we mean by a “defined” `token_id` in TTDex. From our understanding, defined `token_ids` should correspond to either:

- `token_ids` which have been generated by some call to `initialize_exchange` in the past; equivalently, when `token_id` is less than the value of the `pairs_count` storage member
- `token_ids` which currently have non-zero liquidity (i.e. once all liquidity has been drained from a token, it cannot be used again until more liquidity is added)

The second check is more precise in that it prevents useless transactions, but it is not stable over time.

We note that, in this case, standard conformance is *not* a security issue; however, it may impede integration of TTDex into other dapps.

Recommendation: We recommend implementing one of the two definedness checks for the `transfer` and `balance_of` operations. Even though the standard does not seem to require it, we recommend that a similar check be performed by the `update_operators` entrypoint, for consistency.

Response: Madfish Solutions added these missing checks.

Swap Entrypoint Enables Flash Loans

Because the `swap` entrypoint emits the transaction to pay the receiver assets before the transaction in which the sender pays TTDex, this contract can enable flash loans for sophisticated FA1.2/FA2 tokens. Using a pool between assets A and B, the user can do the following:

1. deploy or use an existing FA1.2/FA2 asset that enables setting “hooks” on `transfer`
2. set an on-receive-funds hook for their address that performs a flash loan

3. perform a swap of the form A-to-B and then B-to-A where the user sets `receiver == sender`
4. TTDex pays the receiver first, which invokes the potentially profitable flash loan logic (e.g. using arbitrage)
5. TTDex emits the transaction that makes the user pay back the necessary funds; the user keeps the profit

We do note that, in order for this flash-loan method to be effective, there must be sufficient liquidity in the pool. Naturally, such pools will only exist if the token in question is popular. Thus, this method will only be effective if some hookable token implementation reaches a sufficient level of popularity.

Recommendation: We do not have an explicit code-level recommendation, as we cannot currently find a method to use flash loans to exploit TTDex. In general, we recommend that all features that are available as part of an application be thoughtfully and carefully considered before deployment.

Response: Madfish Solutions swapped the transaction ordering to prevent flash loans.

Subtractions Implicitly Guarded Against Underflow

The code uses the natural number subtraction six times, where the subtraction is defined by:

`A -Nat B = Abs(A -Int B)`

where `Abs` is the absolute value function. In this case, subtraction underflow behavior occurs when $B > A$. We say the subtraction is guarded whenever we require that $A \geq B$.

Of the six subtractions, two of them (4,6) are guarded by an explicit assertion in the code that requires $A \geq B$. The other four subtractions (1-3,5) must be proved guarded via an inductive proof over contract executions to ensure underflow safety. The subtractions are:

1. `(divest_liquidity) TotalShares -Nat Shares`
2. `(divest_liquidity) TokenA -Nat TokenAReq` where `TokenAReq = Shares *Nat TokenA /Nat TotalShares`
3. `(divest_liquidity) TokenB -Nat TokenBReq` where `TokenBReq = Shares *Nat TokenB /Nat TotalShares`
4. `(divest_liquidity) SenderShares -Nat Shares`
5. (swap) for each pair in the route, one of the following:
 - `TokenA - TokenOut` where `TokenOut = (TokenIn *Nat 997 *Nat TokenA) /Nat [(TokenB *Nat 1000) +Nat (TokenIn *Nat 997)]`
 - `TokenB - TokenOut` where `TokenOut = (TokenIn *Nat 997 *Nat TokenB) /Nat [(TokenA *Nat 1000) +Nat (TokenIn *Nat 997)]`
6. (transfer) for each transfer destination `FromBalance -Nat Amount`

By algebra, we can examine weaker conditions under which the other subtractions are safe:

- (1) the *defined liquidity shares consistency* condition from our accompanying appendix is sufficient since it implies `TotalShares >= Shares`

- (2-3) similarly, the *defined liquidity shares consistency* condition from our accompanying appendix is sufficient since it implies $\text{TotalShares} \geq \text{Shares}$, which by algebra, implies $\text{TokenA} \geq \text{TokenAReq}$ and $\text{TokenB} \geq \text{TokenBReq}$
- (5) the condition $\text{TokenB} > 0$ ($\text{TokenA} > 0$) implies $\text{TokenA} \geq \text{TokenOut}$ ($\text{TokenB} \geq \text{TokenOut}$) respectively

Recommendation: Though we believe all the above sufficient conditions are satisfied by the code, we recognize there could be an error in our calculations or that the language semantics may subtly change in the future. Thus, we recommend adding an explicit guard against natural number subtraction underflow to prevent subtle errors or changes from causing unexpected problems.

Response: Madfish Solutions added an additional underflow check for the divest liquidity function for subtraction (1).

Redundant Divestment Deadlock Guard

Summary: There is a possibility, where, if the divestment entrypoint is not implemented properly, it can cause an exchange pair managed by the contract can deadlock. Though our proofs indicate that such an error state is unreachable, since asserting a guard condition that obviously prevents this state is cheap, we see no reason not to do so.

Analysis: During liquidity divestment, for a chosen `PairId`, the metadata for a single exchange pair is updated in the following manner:

```
<pairs> Pairs => Pairs [ PairId <- LP(TotalShares -Nat Shares, TokenA -Nat
    TokenAReq, TokenB -Nat TokenBReq) ] </pairs>
```

with:

- $\text{LP}(\text{TotalShares}, \text{TokenA}, \text{TokenB}) := \text{Pairs}[\text{PairId}]$
- $\text{TokenAReq} := \text{TokenA} * \text{Nat Shares} / \text{Nat TotalShares}$
- $\text{TokenBReq} := \text{TokenB} * \text{Nat Shares} / \text{Nat TotalShares}$

The deadlock occurs when:

- $\text{TotalShares} - \text{Nat Shares} > 0$
- $(\text{TokenA} - \text{Nat TokenAReq}) * \text{Nat} (\text{TokenB} - \text{Nat TokenBReq}) == 0$

In this state, all entrypoints relating to the pair become disabled:

- `initialize_exchange` - disabled due to $\text{total_supply}(\text{Pairs}[\text{PairId}]) == 0$ check
- `invest_liquidity` - disabled due to $\text{TokenA} * \text{Nat TokenB} != 0$ check
- `divest_liquidity` - disabled due to $\text{TokenA} * \text{Nat TokenB} != 0$ check
- `swap` - disabled due to $\text{TokenA} * \text{Nat TokenB} != 0$ check

By the *defined liquidity shares consistency* condition from our accompanying analysis, we can prove that the subtractions above will not underflow. Of course, our proof may have a bug. In case the condition fails to hold, there are situations where the above deadlock can occur, e.g., let:

- `TotalShares = 19`
- `Shares = 20`
- `TokenA = 10`
- `TokenB = 10`

Then, we have:

- `Abs(TotalShares -Nat Shares) == Abs(19 -Int 20) == 1`
- `TokenA -Nat (TokenA *Nat Shares /Nat TotalShares) == 10 -Nat [(10 * 20) /Nat 19] == Abs(10 - 10) == 0`
- `TokenB -Nat (TokenB *Nat Shares /Nat TotalShares) == 10 -Nat [(10 * 20) /Nat 19] == Abs(10 - 10) == 0`

Of course, the underflow prevention above can prevent this particular issue.

Recommendation: As an alternative or redundant protection mechanism, another possibility is to simply assert that:

```
[(TokenA -Nat TokenAReq) *Nat (TokenB -Nat TokenBReq) == 0] == [TotalShares -Nat Shares == 0]
```

That is, the liquidity is fully drained iff the token reserves are drained.

Response: Madfish Solutions implemented this check.

Optimization

These notes are about optimizing the contract's code size/gas usage.

Optimizing `update_operators` code size

From analysis of the generated CFG, we see that the code for adding/removing operators is identical with the exception of the performed set operation (i.e., adding an item to a set or removing an item from the set). By unifying redundant code in the two branches, the code size can be reduced. This may enable the compiler to automatically inline the `get_account` function call, which will further reduce code size and gas cost.

Response: Madfish Solutions implemented this optimization, resulting in a net reduction in code size and gas cost.


Potential readability and gas optimization

From our simple tests, the pattern:

```
assert_with_error(predicate, error_message)
```

compiles to a more compact Michelson byte code than the pattern:

```
if predicate then skip else failwith(error_message)
```



We also find the `assert` pattern to be more readable. However, the downside is that adopting this convention would require negating the assertion predicates, i.e., the `assert` function fails if the predicate is false while the current assertions-as-if-statements convention fails if the predicate is true.

Response: Madfish Solutions implemented this change.

Appendix

Our accompanying appendix available at our [publications repo](#) contains all important assumptions and proofs.