Security Audit Report

Pact.fi

Delivered: 31 January 2022

Prepared for Pact Finance by

Contents

Summary

Pact Finance engaged Runtime Verification Inc to conduct a security audit of their smart contract implementing an automated market maker (AMM).

The objective was to review the contract's business logic and implementation in PyTEAL and identify any issues that could potentially cause the system to malfunction or be exploited.

Timeline

The audit has been conducted over a period of 6 weeks, from November 17, 2021 to December 17, 2021 and from January 10, 2022 to January 21, 2022.

Findings

The audit has identified one medium-severity issue, two low-severity issues and four informative findings. The medium-severity issue is the possibility of bootstrapping a monopolised pool (A1). The low-severity issues are indefinite locking of funds on first mint (A2) and indefinite locking of funds via swaps in absence of liquidity tokens (A3). Informative findings (B1–B4) highlight various aspects of the contract that are not mission-critical but do need to be taken into consideration.

All the issues have been appropriately addressed. The contract's source code is of exceptionally high quality, and we have enjoyed working with the team at Pact Finance, who have provided us with any assistance possible to make the audit process smooth and productive.

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 the 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 recommends always seeking multiple opinions and audits.

This report is also not comprehensive in scope, excluding a number of components critical to the correct 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.

Goals

The goals of the audit were:

- Review the architecture of the liquidity pool smart contract based on the provided documentation;
- Review the PyTEAL implementation of the contract to identify any programming errors;
- Cross check the PyTEAL code of the contract with the documented high-level design and the general expectation of the contract's behaviour.

The audit focuses on trying to identify issues in the system's logic and its implementation that could potentially render the system vulnerable to attacks or cause it to malfunction.

Scope

The audit has been conducted on the commit `b8ea648c5e7b2b7de104b098e484a66f13197473` of Pact Finance's private GitHub Repository. Subsequently, the codebase was prepared for public release, and the audited revision now bears the commit hash 4148978545fa226335a2b85d9a89734989d5ab51. Most of the links in this report refer to this revision which is intended to be released publicly.

The audit scope includes the following files:

- The AMM smart contract source code in PyTeal: ./assets/exchange.py
- Transaction validation code: ./assets/helpers/validation.py
- Natural language specification of the valid transaction groups: `.docs/app_calls.md` (private repository)

Additionally, we have informally reviewed the `pytealext` library code referenced in `exchange.py`:

- Wide multiplication and integer division helper in ./pytealext/muldiv64.py
- Inner transaction preparation helpers in ./pytealext/inner_transactions.py
- Minimul operation in ./pytealext/cas.py

We have not audited any off-chain components of the system. Further patches and developments following the aforementioned commit were not formally audited. We, however, have informally reviewed the patch for the finding A1 (commit hash 2908537948f4fb2f342c05a3c15f4c97f5d46e63).

Background: Constant Product Market Maker model

To facilitate better understanding of this report, We give a brief informal outline of the Constant Product Market Maker model.

Terminology

Liquidity pool is a smart contract, or a number of contracts, that facilitates swapping between two digital assets. In this report, the assets involved are either Algo or ASAs (Algorand Standard Assets).

Liquidity token (LT) is a digital asset that represent a share in the liquidity pool.

LT price is the pair of amounts (x, y) of the two assets of the pool that correspond to one liquidity token.

Mint is an operation of obtaining LTs from the pool by providing amounts of the two assets the pool trades. If the LT price is (x, y), then minting a positive integer number n of LTs requires providing $(n * x, n * y)$.

Burn is an operation of redeeming LTs for the underlying assets of the pool.If the LT price is (x, y), then burning a positive integer number n of LTs redeems $(n * x, n * y)$.

Swap is an operation of depositing an amount x of one assets and receiving the amount y of the other asset. The amount y is determined by the pool based on the amount x and the current pool reserves X and Y as follows:

$$y = \frac{x * Y}{X + x} \qquad X' = X + x \qquad Y' = Y - y \tag{1}$$

Swap fee is an amount that, upon executing every swap, is subtracted from the swapped amount and donated to the pool. The fee is controlled by the amount t, and the *taxed* swap amount is calculated as follows:

$$y_{taxed} = \frac{x * Y}{X + x} * (1 - t), t \in [0, 1). \tag{2}$$

The typical value of t is 0.003, i.e. 0.3%.

Pooler is an account that provides amounts of the two assets of the pool and holds the LTs representing their share in the pool. h

Swapper is an account that swaps an x amount of one assets of the pool for the amount y of the other asset, determined by the pool.

Characteristics of a healthy pool

1. Reserves product does not decrease with swaps

Suppose the pool collects a fee of 0.3% on every swap. Then, the product of the reserves, rather then staying constant, will be *not-decreasing*.

Theorem 1. *Let $X > 0$ and $Y > 0$ be the pool reserves, $x \geq 0$ be the swap amount and $0 \leq t \leq 1$ be the fee. Then, after the swap, the product of the new reserves will always be at lest as large as before the swap:*

$$X \times Y \leq X' \times Y'$$

Proof. We proceed with a proof by contradiction. Suppose otherwise, i.e. $X \times Y > X' \times Y'$. Then we have:

$$\begin{aligned} X \times Y &> (X + x) \times (Y - \frac{x \times Y}{X + x} \times t), \quad \text{by equations (1) and (2)},\\ X \times Y &> X \times Y + x \times Y \times (1 - t),\\ 0 &> x \times Y \times (1 - t) \end{aligned}$$

which is impossible, since $x \geq 0$, $Y > 0$ and $1 - t \geq 0$.

QED

2. Liquidity tokens (LTs) should be affordable.

In a decentralised Automated Marked Maker, the pool's liquidity is attracted from anonymous liquidity providers (poolers) who provide their assets in exchange for the contract's liquidity tokens (LTs). The prerequisite of a healthy pool is the low price of a single liquidity token — a small share in the pool needs to be affordable to almost any investor.

Initial mint sets the LT price. A valid empty pool has empty reserves and no pooler has any LTs of this pool. The initial mint involves a pooler sending two amounts (x, y) of assets. The amount of LTs they get is the geometric mean of the asset amounts:

$$LT_{init} = \sqrt{x \times y} \tag{3}$$

For a stablecoin pool, the pooler may send $(1000, 1000)$ base units of the assets (for example 0.01 USDC and 0.01 USDT), minting $\sqrt{1000 \times 1000} = 1000$ LTs. The price of one LT is then $(X_{init}/LT_{init}, Y_{init}/Y_{init}) = (1000/1000, 1000/1000) = (1, 1)$, i.e. two base units of a dollar-pegged stabelcoin.

The low price of the pool's liquidity token guarantees diversity of poolers, since it is equally affordable for investors with very large and very small capital to provide liquidity.

Methodology

Audit process

The audit process has been an interleaving of manual review of documentation and code and tool-assisted modelling of the protocol.

To build a precise understanding of the Pact's implementation of the CPMM model, we have used the K framework to build a high-level model of the smart contact. The model abstracts away the Algorand-specific peculiarities and allows to focus on the properties described in the background section.

The developed model has allowed us to assess how faithful the Pact.fi protocol is to the Constant Product Market Maker with fees model. In this phase, by identifying the discrepancies between the model and the implementation, we have discovered the findings A1, A2 and B1.

Since the Algorand ecosystem is fairly young, many development best-practises remain to be established. To account for possible Algorand-specific vulnerabilities in the contracts, such as missing transaction Fields checks and malformed transaction groups, we have performed a thorough manual code-review of the system implementation. This phase has lead to identifying the rest of the findings presented in this report.

Over the whole timeline of the audit, we maintained a channel of communication with the team at Pact. The discussions of the on-going progress and reporting of findings in realtime enabled us to make sure we were never digging into a dead end.

Severity classification

We have adopted a severity classification inspired by the one of Immunefi. The A-labelled findings include their severity estimate in parenthesis following the title. The B-labelled additional findings are purely informative.

Findings

A1. Pool monopolisation attack (Medium)

Description

An attacker possessing substantial but realistic capital can bootstrap a valid pool and create prohibitively expensive liquidity tokens, preventing other people from adding small amounts of liquidity to the pool.

Attack scenario

We present a particular instance of the attack that leads to creation of a `goBTC/USDC` pool with the LT price of (`BTC` 0.69, $30000). The capital requirements of, in this particular scenario, are 0.7 `BTC` and 1 million USDC. The pool bootstrap (not counting creation which mush be separate) comprises 10 transactions and fits into one atomic transaction group.

TXN	Action	Attacker holdings			Pool holdings	
		BTC	USDC	LT	BTC	USDC
0	`create_lt`	0.7	1,000,000.000001	0	0	0
1,2,3	`mint(3, 1)`	0.69999997	1,000,000	1	3 satoshi	1 μUSDC
4,5	`swap(`10^6` USDC)`	0.69999998	0	1	2 satoshi	≈1,000,000
6,7	`swap(0.69 BTC)`	0	≈ 970000	1	≈ 0.69	≈ 30,000

The resulting ratio of the pool is

$$\frac{30000USDC \times 10^6}{0.69BTC \times 10^8} \approx 434\frac{\mu USDC}{satoshi}$$

and is close to the today's price. The parameters of the attack can be altered to achieve other prices if needed.

What makes the attack possible

The attack exploits the fact that the pool's smart contract allows supplying microscopic amounts of initial liquidity and thus minting just one liquidity token. Since, upon initial mint, the attacker will own all assets in the pool, they exploit the perfectly valid swapping logic to effectively deposit more liquidity into the pool without minting more LTs. Additionally, the attacker's funds are perfectly safe, since the attack fits into one atomic group and thus does not present an arbitrage opportunity.

Recommendation

We recommend to follow the approach implemented by the prominent Uniswap V2 protocol of the Ethereum ecosystem. Uniswap V2 requires at least 1000 LTs to be minted on initial liquidity provision, thus forcing a reasonable LT price. The attacker may still try inflating the liquidity token price with the same swap trick, but the capital requirements become unrealistic.

Status

Acknowledged by Pact. The issue has been addressed in a subsequent commit 2908537948f4fb2f342c05a3c15f4c97f5d46e63. The changes were not formally audited.

A2. Indefinite locking of funds on first mint (Low)

Description

On initial LT mint, if a user deposits either $(a, 0)$ or $(0, b)$ such as $a > 0$ and $b > 0$ then zero liquidity tokens are minted and the non-zero amount is donated to the pool.

Note that this donation will be invisible to the pool, since on a consecutive liquidity provision `total_liquidity` is still zero, and the reserves will be reinitialised with the new deposits. Therefore the donated assets will be on the contract's balance, but will not be tracked by the protocol.

This issue does not seem to present an attack scenario. No price manipulation attack seems to be possible, since on initial mint the geometric mean of the initial *deposits*, not the *reserves*, is computed. However, we suggest blocking such deposits to make the protocol more faithful to the traditional CPMM model and avoid accidental locking of funds.

Recommendation

To make such donations impossible, the contract needs to assert that at least some liquidity tokens were minted upon initial liquidity deposit:

```
# terminate the execution if no tokens were minted
Assert(lt_minted.load()) > Int(0))
```

this assertion may be inserted after line 389 of `exchange.py`.

Note also that introducing a fix for A1 will make this fix obsolete, since the initial liquidity will be asserted to be at least 1000. Therefore, the patch for A1 will introduce the necessary changes for A2 too.

Status

Acknowledged by Pact. Changes to be introduced before launch on Algorand mainnet.

A3. Indefinite locking of funds via swaps in absence of liquidity tokens (Low)

Description

Neither the top level function `on_swap`, nor the internal functions `_swap_primary` and `_swap_secondary` check that the number of LTs minted is non-zero. This causes an unintended loophole allowing, via swaps, to bootstrap a pool that would be valid for swaps, but in which no one will own any LTs. If on such a pool an initial mint is performed, the current pool's reserve amounts will be lost forever, similar to A2. Even though funds can be lost, **this issue does not seem to be presenting an attack scenario**.

Recommendation

Forbid bootstrap of such "premature" pools by asserting that non-zero liquidity tokens were minted before performing a swap. To achieve that, one can add an assertion after line 336 of the `on_swap` function:

```
Assert(self.total_liquidity.get() > 0)
```

Status

Acknowledged by Pact. Changes to be introduced before launch on Algorand mainnet.

Additional Findings

B1. Bootstrapping large symmetrical pools can be denied due to integer overflow (Informative)

Description

When the liquidity is first provided, the amount of LTs minted is $\sqrt{a * b}, a > 0, b > 0$. The multiplication operation will overflow the `uint64` type if the product of a and b is greater then $2^{64} - 1$, in base units. For example, for a dollar-pegged stabelcoin pool, depositing 10000 USDC and 10000 USDT will cause overflow, because both assets have 6 decimals and $10^{10} \times 10^{10} > 2^{64} - 1$.

Pact's current mitigation strategy

The Pact team is aware of this problem and has a valid mitigation strategy. If an initial mint with critical amounts is requested, it will be split by Pact's frontend web application into two transactions. For power users interacting with the smart contract directly, a warning will be added to the documentation.

Recommendation

Ultimately, the solution would be to use a wide squere root operation, that would allow a byte-slice argument. Unfortunately, such operation is not yet available, as of TEALv5.

Status

The Pact team has been aware of the potential issue prior to the audit. The team has both a short-term mitigation strategy and a plan to integrate the wide square root operation once it becomes available in a further versions of TEAL.

B2. Pool bootstrap process is not always atomic (Informative)

Bootstrapping a valid pool involves three mandatory and one optional steps:

1. deploying an instance of the `exchange.py` contract,
2. opting the contracts into the primary and secondary ASAs,
3. triggering the creation of the liquidity ASA,
4. (optional) providing initial liquidity.

The first step must be executed in its own transaction group, since only upon its success there will be an `ApplicationID` available. Steps 2 and 3 are independent, but must both be done by the pool's creator in order for the pool to be operational. Now, the forth step, if not bundled with steps 2 and 3 in an atomic transaction group, can be performed by anyone else. Often the creator would want to supply the initial liquidity too, and thus they need to be made aware of this peculiarity.

Status

Acknowledged by Pact. This is not a problem, but may need to be mentioned in the documentation for power-users.

B3. Redundant computation in `on_add_liquidity` (Informative)

Description

The expression `Min(lt_primary.load(), lt_secondary.load()` is first computed on line 408 and then recomputed again several times.

Recommendation

The evaluated expression's result can be stored in the scratch space and reused. However, the code can also be left as is since, currently, the application is far from reaching the code size and computing time limits.

Status

Acknowledged by Pact.

B4. Liquidity token unit name is the same for every pool (Informative)

Description

The unit name for a pool's Liquidity Token is always LQT. This may become confusing if users want to create pools trading LTs of other pools for algos or other asas. The LT unit name for any of these ”second-order” pools will be ”LQT/X”.

Status

The Pact team has been aware of this prior to the audit.

In Algorand, assets are identified by their unique asset id. The unit name is purely cosmetic and is restricted to be at most 8 bytes [1] and has no restriction to be unique. Pool tokens that become valuable will have to be empathised through off-chain mechanisms.

[1]See Algorand developer documentation table AssetParams