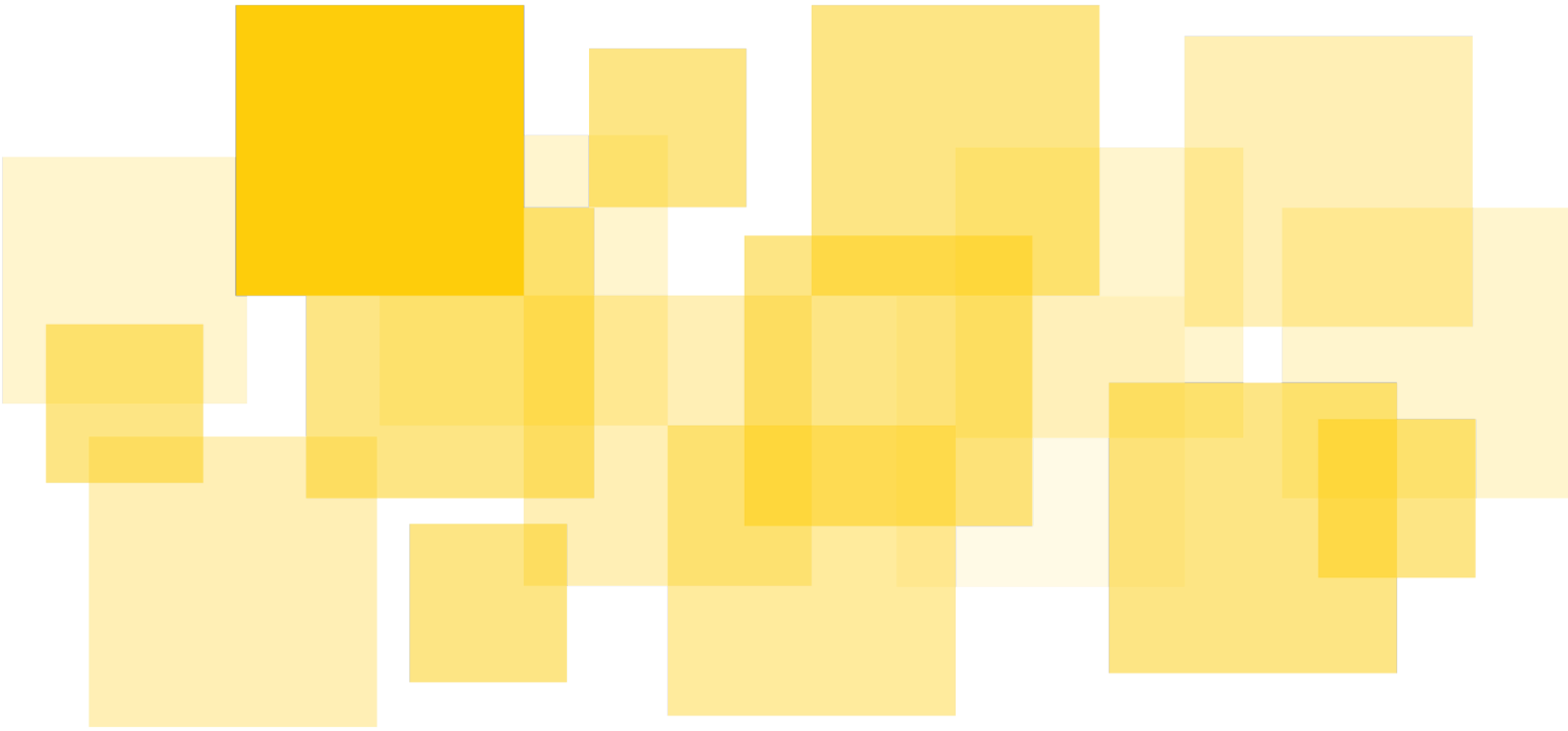
Security Review Report

Tinyman Smart Contracts

Delivered: January 7th, 2022



Prepared for Tinyman by

runtime verification

Table of Contents

Executive Summary

Tinyman engaged Runtime Verification Inc to conduct a security review of their smart contracts in light of a recent exploit of a previously unknown vulnerability. The objective was to examine the vulnerability that enabled the exploit and review the changes made to the contracts.

The review was conducted in cooperation with the Tinyman team during the period January 3-6, 2022. Initial analysis of the vulnerability was based on commit ID a22a1278eaa1b73fdf264db27dcfee145ca31dc3, and a review of the changes made was based on a later commit with ID 8be3e7f8005bb131c51e10e82885a8e764d7a336.

Analysis of the exploit identified the missing checks in the contracts that lead to the vulnerability: In a burn operation, the type of the fourth transaction in the transaction group and the asset IDs supplied in the third and fourth transactions of the group were not checked to ensure that they had the expected values. This along with the fact that there were liquidity pools with pairs where one asset is worth much more than the other, enabled the attacker to craft malicious requests to effectively retrieve much more than what they were entitled for in a legal burn operation.

This vulnerability is addressed in the latest commit 8be3e7f8005bb131c51e10e82885a8e764d7a336 by adding the missing checks and confirming that the attack no longer applies. This commit also addresses an issue that was highlighted by the Tinyman team back in November 2021 that affected pools with pairs of extreme price disparities.

The issues described in this report were not discovered in the original audit of the codebase and as such not included in the audit report published in September 2021.

Goal and Scope

The goal of this review is twofold:

1. Investigate the vulnerability that enabled the January 2022 exploit on Tinyman contracts and analyze the attack
2. Review the changes made to the contracts that address this vulnerability, along with other minor fixes.

The engagement was a targeted manual review of relevant parts of Tinyman's smart contracts rather than a full security audit.

The scope of this review is limited to the affected parts of the smart contracts:

1. Pool_logicsig.teal.tmpl
2. validator_approval.teal

Other parts of the contracts, off-chain and client-side portions of the codebase as well as deployment and upgrade scripts are not in the scope of this work.

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, 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 a nascent software arena, and their deployment and public offering carries risk. This report makes no claims that its analysis is fully comprehensive. The possibility of human error in the manual review process is real. We recommend always seeking multiple independent opinions and audits.

Missing Asset ID Checks in Burn Operations

Overview

The burn operation enables a pooler to exchange their previously minted liquidity pool tokens for what these tokens currently represent in terms of tokens of the asset pair of the pool. For example, for the goBTC-Algo pool, a LP token may represent 1 goBTC token and 30,000 Algo, so burning 2 LP tokens involves sending them back to the pool while receiving 2 goBTC and 60,000 Algo.

The burn operation in Tinyman is specified as an atomic transaction group that consists of exactly five transactions (indexed from 0 through 4):

1. **Txn0**: A fee-paying transaction from the Pooler to the Pool
2. **Txn1**: A "burn" application call to the validator signed by the Pool
3. **Txn2**: An asset transfer from the Pool to the Pooler with the amount of tokens for the first asset (the 2 goBTC in the example above)
4. **Txn3**: An asset transfer (or a pay transaction) from the Pool to the Pooler with the amount of tokens for the second asset (or Algos) (the 60,000 Algo in the example above)
5. **Txn4**: An asset transfer from the Pooler to the Pool with the amount of LP tokens to be burned (the 2 LP tokens in the example above)

In normal operation, these transactions are constructed and submitted by the Tinyman frontend interface (after the pooler signs Txn0 and Txn4 above), but in general, any non-Tinyman interface could be used. Therefore, transactions that are signed by the Pool (in this case, Txn1, Txn2 and Txn3 in the group) are untrusted and they need to be checked by the contracts to ensure that they use proper values in their fields to protect the pool logic signature accounts.

Vulnerability

Despite the fact that the pool_logicsig and validator_approval contracts already perform some extensive checking of transaction fields, in a burn operation, they fail to check that:

1. The fourth transaction (Txn3 in the group) is of the correct type (a pay transaction for Algos vs. an asset transfer transaction for ASAs), and that

2. The third and fourth transactions (Txn2 and Txn3) are asset transfers for the expected asset IDs

These missing checks mean, in particular, that a burn transaction group could potentially request payment/transfer of the same asset twice (by having both Txn2 and Txn3 above be asset transfers of goBTC in our example).

Attack

Therefore, if one of the assets in the pair of a pool has a unit price that is much greater than that of the other asset in the pool, an attacker could request both transfers in a burn to be in the higher-valued asset, which would result in retrieving much more than what the attacker is entitled to using their LP tokens. For example, in the goBTC-Algo pool example above, the attacker constructs a transaction group that includes for Txn2 and Txn3:

- Txn2: An asset transfer for 2 goBTC from the Pool to the attacker (legal asset transfer transaction)
- Txn3: An asset transfer for 600 goBTC from the Pool to the attacker (Illegal asset transfer transaction)

The transaction Txn3 is supposed to be a payment transaction for 60,000 Algos, but the contracts do not check the transaction type or that it uses the correct asset ID. So the contracts are fooled into calculating the amount of goBTC while thinking that it's Algos. Since goBTC has 8 decimals while Algo has 6 decimals, the 60,000 amount interpreted as goBTC would be 600. However, given the significant price difference in the pair, the value transferred to the attacker (600 goBTC) is much higher than what the attacker was entitled to (60,000 Algos).

So, in general, for this attack to succeed, the attacker needs to ensure that there are enough basic units of the correct asset to cover the requested amount of the targeted asset in the pool, otherwise the transaction group will fail (the pool has to have at least 600 goBTC in our example). Moreover, the burn amount has to be selected so that the amount calculated by the contracts is less than the supply but more than the amount requested by the attacker (interpreted using the decimals of the target asset). So in our example, requesting any amount less than 600 goBTC would also work.

Note that the example used in this section is for illustration purposes only. The actual attack targeted multiple pools and used different amounts (See the reports posted by Tinyman).

Recommendation

The solution is to add the missing checks by:

1. Checking that the fourth transaction Txn3 in a burn operation is of the correct type (if the second asset is Algo, then the transaction is a pay transaction, otherwise it's an asset transfer transaction)
2. Checking that the third and fourth transactions (Txn2 and Txn3) are transfers of the correct asset ID (if Txn3 is an asset transfer). The correct asset IDs are the ones for which the pool was initially bootstrapped.

Status

This vulnerability has been addressed in the latest commit referenced by this document (commit ID: 8be3e7f8005bb131c51e10e82885a8e764d7a336). The pool_logicsig contract has been augmented to include both checks for the burn operation:

```
// ensure asset id is asset 1
gtxn 2 XferAsset
int TMPL_ASSET_ID_1
==
assert


// ensure asset id is asset 2
gtxn 3 XferAsset
int 0
gtxn 3 TypeEnum
int pay
== // check if Algo
select
int TMPL_ASSET_ID_2
==
assert
```

Similar checks have also been added to the mint logic, and the logic of other operations was also reviewed to ensure that these checks for asset IDs are in place.

The Overflow Issue In the Price Oracle

Overview

In order to support the use of Tinyman as a decentralized and permissionless price oracle, the validator_approval contract includes logic that is executed at most once per block to update the cumulative prices in a pool (which can be used by other contracts to calculate a spot price for the pair). The relevant part of the update is shown here:

```
cumulative_price += (price * time_delta)
```

Where `price` is the pool's price of an asset (with 6 decimals), and `time_delta` is the time amount (in seconds) since the cumulative price was last updated. This update is performed before a mint, burn or swap is executed.

Issue

This calculation has a known issue that is well documented here. Essentially, the product (`price * time_delta`) may overflow in extreme cases for pools where the price ratio of the asset pair is extremely high (e.g. 10,000,000:1) and the pool is dormant (so `time_delta` is large). The problem is that since this calculation is performed before every mint, burn and swap operation on the pool, an overflow means that the pool becomes unusable, and will stay forever locked as the price won't change (no swaps can go through) and `time_delta` will only increase over time. Tinyman has a detailed analysis of the impact of this issue along with a full list of affected pools here.

Resolution

The Tinyman team addressed this issue in the latest commit (commit ID: 8be3e7f8005bb131c51e10e82885a8e764d7a336) so that if this calculation overflows, it will not cause execution to panic. The logic now uses wide multiplication and checks whether the high 64 bits of the product are zero (so no overflow). In the case of an overflow, the update is simply skipped.

Therefore, an overflowing calculation of the cumulative price will not block operations on the pool, allowing it time to adjust prices through swaps and mints. It may theoretically still happen that a pool never gets to a reasonable price, and thus it's price oracle will be unusable, but at least the pool won't get stuck and will continue to function. This

limitation of using the pool as a price oracle in this extreme and rare case should be documented as a known minor limitation.