SOLIDIFIED

Audit Report for Melonport. May 11, 2018.

Summary

Audit Report prepared by Solidified for Melonport covering the competition and price feed contracts. Fund contracts were not in the scope of this audit.

Process and Delivery

Two (2) independent Solidified experts performed an unbiased and isolated audit of the below contracts. The debrief took place on May 11, 2018 and the final results are presented here.

Audited Files

The following files were covered during the audit:

- assets/Asset.sol
- assets/AssetInterface.sol
- assets/ERC20Interface.sol
- assets/ERC223Interface.sol
- assets/Shares.sol
- competitions/Competition.sol
- competitions/CompetitionInterface.sol
- compliance/CompetitionCompliance.sol
- compliance/ComplianceInterface.sol
- dependencies/DBC.sol
- dependencies/Owned.sol
- pricefeeds/CanonicalPriceFeed.sol
- pricefeeds/SimplePriceFeed.sol
- pricefeeds/StakingPriceFeed.sol
- system/OperatorStaking.sol
- system/StakeBank.sol
- system/StakingInterface.sol
- version/Version.sol
- version/VersionInterface.sol

Intended Behavior

The purpose of these contracts is to facilitate a “competition” for Melon fund managers. The `Competition` contract invests Melon (MLN) in registrants’ funds at a favorable rate. At the end of the competition period, registrants can claim ownership of that share of the managed assets.

The audit was based on commit `ce31411e34a6afdff539965bd262f270db1d7cfb`.

Issues Found

Critical

1. CanonicalPriceFeed does not prevent price manipulation

Prices from the price feed are used to enforce maximum buy-ins and, more importantly, to compute how much Melon (MLN) to invest in a contract. If a malicious actor is able to compromise the price feed, they can contribute a small amount of ether but receive a very large amount of MLN.

The price feed is based on a staking scheme. The point-in-time price is the median of the prices submitted by the top *n* stakers. The `burnStake` function ostensibly allows the feed owner to punish bad actors, but nothing prevents a malicious participant from removing their stake before it can be burned. An attack can be performed in a single transaction, in which an attacker:

1. becomes all top *n* stakers (from different contracts),
2. submits a malicious price from those contracts,
3. registers for the competition (making use of that price),
4. and finally withdraws the stakes.

Additionally, an attacker could front-run transactions attempting to burn their stake.

Recommendation
Any funds staked to participate in the price feed should be held for long enough that they can be later slashed as punishment. Stakers should not be able to withdraw their stake immediately, instead a significant delay should be added between unstaking and stakers being able to withdraw their stake.

Major

2. CanonicalPriceFeed is not trustless

The owner of a `CanonicalPriceFeed` can fully control the price reported by that feed. As described in the previous issue, the mitigation for price manipulation is staking. The owner has the ability to slash a participant's stake in the case of malicious activity. Ignoring weaknesses in

that ability, the owner can slash everyone else's stake and leave only their own. This lets them control the price feed entirely.

Recommendation
We recommend using a decentralized mechanism for slashing stakes. Decentralized oracles are still an area of active research, so it's difficult to recommend a specific approach.

AMENDED [2018-6-7]:
Client's response:
For the purposes of the competition, it is known that this mechanism is centralized/controlled by Melonport AG (was made explicit in the terms and conditions of the competition). This will be addressed when implementing governance over the protocol, which is scheduled to take place in the next few months.

3. Potential denial of service by malicious staker / Dynamic array inefficient for storing sorted stakers

---

`OperatingStaker` imposes no bound on the amount of stakers (beyond requiring a minimum stake): since the `stakeRanking` array is iterated over when updated, a malicious staker can add enough entries to `stakeRanking` that operations on it (i.e. updating stakers' ranking) can exceed the block gas limit, effectively locking up the ranking. More generally, needing to update the entire tail end of `stakeRanking` is very gas inefficient.

Recommendation
Either limit the amount of entries in `stakeRanking` so that the gas block limit cannot be exceeded or utilize a data structure that can't be DOS'd in this way: a doubly linked list that allows providing witnesses for more efficient search as described here.

Client's response:
We added a limit to the number of items in the array, and a test for this.

Our Response:
Unfortunately the attempted fix is problematic because:
(i) The function `stakeFor` is missing a modifier. When calling `stake` the code checks:

```
pre_cond(
    stakeRanking.length < MAX_STAKERS ||    // still room in array
    amount > stakeRanking[0].amount         // or larger than smallest
element
)
```

This precondition is missing from `stakeFor` where presumably the same conditions should apply.

(ii) Due to the aforementioned precondition, if users continue to stake increasing amounts, `stakeRanking` can grow to an unbounded length. This is because if users continue to stake increasing amounts `amount > stakeRanking[0].amount` will always be true. The function `addStakerToArray` could prune `stakeRanking` if its length grows larger than `MAX_STAKERS`.

(iii) It's still gas inefficient, at minimum using binary search for operations on `stakeRanking` would be prefered.

4. CanonicalPriceFeed allows for small difficult-to-detect price manipulation

---

`CanonicalPriceFeed` puts late participants in a position of privilege. They can examine the prices submitted so far and choose their price such that it pushes the median in one direction or another. This manipulation is difficult to detect because the submitted price can fall well within the range already present in prices submitted.

This issue was originally called out in the contract's documentation:

"A member of the pool of Price Feed Operators (PFO) can manipulate the protocol-selected price data point as follows: Assume a pool of 5 PFOs. PFOs 1-4 submit price points:

PFO1 22.14 PFO2 22.76 PFO3 23.18 PFO4 23.21

PFO 5 can basically determine price the price 22.76 or 23.18 by waiting for all others, analyzing the result and submitting a price point <= 22.14 or >= 23.21, respectively. They could also select their own price by arbitrarily submitting a price point between 22.76 and 23.18. This could also result in PFOs all waiting until the last possible moment to submit price data in order to essentially have the privileged role of determining price."

Recommendation
As the documentation suggests, a commit/reveal scheme would prevent late participants from seeing the other submitted prices and thus remove the ability for subtle manipulation.

AMENDED [2018-6-7]:
Client's response:

This is another issue that will be addressed in the coming months. For the purposes of the competition this bug should not present a problem, since any potential price manipulation should be small in magnitude, mitigating its effects, and punishment via stake burning may provide a deterrent to this behaviour.

5. CanonicalPriceFeed provides no incentive for honest participants

---

Participating in the price feed carries significant staking risk and gas cost, but there is no reward for submitting accurate prices. That means the only existing incentive (at the protocol level) is for dishonest participants who are trying to manipulate the reported price.

Recommendation
Provide some incentive for honest participants, likely in the form of a reward for submitting an accurate price. This could simply be a reward granted to all participants, assuming some working form of slashing stakes as punishment is implemented to dissuade the malicious actors.

AMENDED [2018-6-7]:
Client's response:
When governance is in place, we will have established a mechanism to reward honest operator behaviour. Current ideas are : (i) inflating MLN supply to reward staked operators, or (ii) a small maintenance fee collected from funds themselves on an annual basis. This is still being researched.

Minor

6. Asset follows neither the ERC20 nor ERC223 standard

A comment on `Asset` states "Asset Contract for creating ERC20 compliant assets," but the contract inherits from `ERC223Interface` and implements neither standard fully.

In terms of ERC223 compatibility:

a) The `Asset` contract doesn't call `tokenFallback` when transferring ether to a contract. (That code is commented out.) This could cause problems for developers who assume the ERC223 standard is followed.
b) The ERC223 standard dictates that the fourth parameter to the `Transfer` event is the byte array sent with the transfer transaction. `Asset` instead always logs an empty byte array.

In terms of ERC20 compatibility:

a) The contract doesn't emit the ERC20 Transfer event. (The signature is instead for the ERC223 `Transfer` event.) Tools that work with ERC20 contracts won't be able to detect this event.

Recommendation
Implement either ERC20 or ERC223 fully, and clarify in the code and documentation which standard is being followed.

AMENDED [2018-6-7]:
This issue is no longer present in commit `02a6ba14a0342f3546324c77e07f92b573138ac8`.

7. Shares contract implements only a portion of ERC20 standard

The `Shares` contract seems to be an ERC223-compatible token but also includes the ERC20 functions approve and transferFrom. It does *not*, however, emit the ERC20-compatible `Transfer` event.

Recommendation
If ERC20 compatibility is desired, implement the ERC20 `Transfer` event.

AMENDED [2018-6-7]:
This issue is no longer present in commit `02a6ba14a0342f3546324c77e07f92b573138ac8`.

8. Mitigation for approval race condition in Shares contract is not backward compatible

The `approve` function in the `Shares` contract enforces a mitigation for an approval race condition. The ERC20 standard states that although *clients* should enforce this mitigation, *contracts* should *not*, so they can maintain backward compatibility.

Recommendation
To stay fully compatible with ERC20, `Shares` should allow updates to an allowance without first setting the allowance to 0. Alternative mitigations which preserve backwards compatibility are described here.

AMENDED [2018-6-7]:
This issue is no longer present in commit `02a6ba14a0342f3546324c77e07f92b573138ac8`.

Note

9. claimReward does not follow the Checks-Effects-Interactions pattern

To avoid potential reentrancy issues, this function should follow the Checks-Effects-Interactions pattern.

Recommendation
Set `registrant.isRewarded` before calling out to the `fund` contract.

AMENDED [2018-6-7]:
This issue is no longer present in commit `02a6ba14a0342f3546324c77e07f92b573138ac8`.

10. Remove unused code

Some contracts are unused but remain in the repo: for example, `PriceFeed.sol` and `AssetRegister.sol`. Of the used code, various sections are commented out but still present: for example, the ERC223 features in the `Asset` contract. The presence of unused code is confusing and misleading for readers of the code.

Recommendation
For clarity, unused code should be removed.

AMENDED [2018-6-7]:
This issue is no longer present in commit `02a6ba14a0342f3546324c77e07f92b573138ac8.`

11. Erroneous comments

There are a number of inaccurate comments in the code that could cause trouble for competition participants who wish to audit the code:

- `Competition.sol` line 106 is an incorrect copy/paste from another comment.
- `Competition.sol` line 203 says “and transfer it to registrant,” but this is inaccurate. (Shares are transferred only at the end of the competition.)
- `Version.sol` line 89 has an incomplete comment (“// Check if the”).

- `Version.sol` line 91 says "// Either novel fund name or previous owner of fund name", but no such check is performed. (We believe the code is doing what's expected, and the comment should just be deleted.)
- `Version.sol` line 112 says "and trigger selfdestruct" in reference to shutDownFund. The function doesn't do that (nor should it).

Recommendation
For clarity, make sure the comments match the actual and intended behavior of the code.

12. Getters are unnecessary for public state variables

Getter functions are automatically created for public state variables, so view functions which solely return a public state variable are extraneous (e.g. `MELON_ASSET` being public makes `getMelonAsset()` unnecessary).

Recommendation
Remove redundant view functions.

AMENDED [2018-6-7]:
Client's response:
This is certainly a valid point, but we use some of these functions in the frontend right now. In order to keep compatibility with our frontend for the competition (time-sensitive), we will keep them until we remove them from API in a later version.

13. Update compiler version

We recommend using the latest compiler, currently `0.4.23`. This version includes support for a less ambiguous constructor syntax, "reason strings" to communicate why transactions are reverted, and a clearer "emit" syntax for events.

AMENDED [2018-6-7]:
Client's response:
Changed in `02a6ba14a0342f3546324c77e07f92b573138ac8`. Updated to `0.4.21` instead, since >=`0.4.22` gives a strange runtime error, which needs to get investigated

SOLIDIFIED

Audit Report for Melonport. May 11, 2018.

Closing Summary

Multiple issues were found during the audit that can break the desired behaviour. It's strongly advised that these issues be corrected before proceeding to production.

Disclaimer

Solidified audit is not a security warranty, investment advice, or an endorsement of the Melonport platform or its products. This audit does not provide a security or correctness guarantee of the audited smart contracts. Securing smart contracts is a multistep process, therefore running a bug bounty program as a complement to this audit is strongly recommended.

The individual audit reports are anonymized and combined during a debrief process, in order to provide an unbiased delivery and protect the auditors of Solidified platform from legal and financial liability.

Solidified Technologies Inc.