SOLIDIFIED

Audit Report for Melonport. June 8, 2018.

Summary

Audit Report prepared by Solidified for Melonport covering the latest iteration of their protocol intended for use in the context of their Melon Olympiad asset management competition. Solidified was originally engaged for an audit restricted to the Melon competition contract, then subsequently for a full protocol audit. As a result, some issues which where not remediated in the interim or still applicable were re-reported below.

Process and Delivery

Two (2) independent Solidified experts performed an unbiased and isolated audit of the below contracts. The debrief took place on June 8, 2018 and the final results are presented here.

Audited Files

The following files were covered during the audit:

```
src
 ├──── assets
 │     ├──── AssetInterface.sol
 │     ├──── Asset.sol
 │     ├──── ERC20Interface.sol
 │     ├──── SharesInterface.sol
 │     ├──── Shares.sol
 ├──── competitions
 │     ├──── CompetitionInterface.sol
 │     ├──── Competition.sol
 │     └──── ERC20Interface.sol
 ├──── compliance
 │     ├──── CompetitionCompliance.sol
 │     ├──── ComplianceInterface.sol
 ├──── dependencies
 │     ├──── DBC.sol
 │     └──── Owned.sol
 ├──── exchange
 │     ├──── adapter
 │     │     ├──── ExchangeAdapterInterface.sol
 │     │     ├──── MatchingMarketAdapter.sol
 │     │     └──── ZeroExV1Adapter.sol
 ├──── FundInterface.sol
 ├──── FundRanking.sol
```

```
├──── Fund.sol
├──── pricefeeds
│     ├──── CanonicalPriceFeed.sol
│     ├──── CanonicalRegistrar.sol
│     ├──── SimplePriceFeedInterface.sol
│     ├──── SimplePriceFeed.sol
│     └──── StakingPriceFeed.sol
├──── riskmgmt
│     ├──── RiskMgmtInterface.sol
│     ├──── RMMakeOrders.sol
├──── system
│     ├──── Governance.sol
│     ├──── OperatorStaking.sol
│     ├──── StakeBank.sol
│     └──── StakingInterface.sol
└──── version
     ├──── VersionInterface.sol
     └──── Version.sol
```

Intended Behavior

Melon protocol is a blockchain protocol for digital asset management. The purpose of these contracts is to implement the protocol as described here, and furthermore 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 `541b1052a97532f3557cda2891d2c2f805e1aa06`.

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.

Client's response:
We added a delay in the staking contract.

Our Response:
Unfortunately the attempted fix does not properly address the issue. Melonport added a delay that prevents users from unstaking X amount of time after last staking. An attacker can wait an X + 1 amount of time, after initially staking and then perform the attack described above. What is needed is a withdraw delay after the last time they *influenced the price*: so Melonport has sufficient time to realize a price manipulation has been attempted and burn the attacker's stake, before the attacker can remove their funds from danger. In other words the unstake action (the act of removing your influence from the price feed), should be distinct from withdrawal (the act of revoking custody from the stake bank and transferring the assets back to the staker). The delay needs to exist between these two actions without regard to when the user began staking.

AMENDED [2018-6-27]:
This issue is no longer present in commit `8cc187707652f87f9b2837687730a5cafc544f04`. Additionally Melonport has set a fixed MLN-ETH exchange rate for competition investment purposes, rather than rely on the price feed.

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.

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.

AMENDED [2018-6-27]:
This issue is no longer present in commit `8cc187707652f87f9b2837687730a5cafc544f04`.

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.

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.

Client’s response:
When governance is in place, we will have established a mechanism to reward honest operators behaviour. Current ideas are: (i) inflating MLN supply to reward staked operators, (ii) a small maintenance fee collected from funds themselves on an annual basis, or (iii) community pricefeed. This is still being thought through.

6. Malicious investors in a fund can force it to close

In function `emergencyRedeem(uint shareQuantity, address[] requestedAssets)` the code block that checks for the critical error of not having enough funds in custody to fulfill an emergency redemption can be abused.

Fund.sol / Line 605-610

```
// CRITICAL ERR: Not enough fund asset balance for owed ownershipQuantitiy,
eg in case of unreturned asset quantity at address(exchanges[i].exchange)
address
if (uint(AssetInterface(ofAsset).balanceOf(this)) < ownershipQuantities[i])
{
    isShutDown = true;
    ErrorMessage("CRITICAL ERR: Not enough assetHoldings for owed
ownershipQuantitiy");
    return false;
}
```

The attacker can pass the address of a malicious smart contract that spoofs an ERC20 token: by implementing a `balanceOf` function that initially returns a value to make `ownershipQuantities[i]` positive, but then returns 0, triggering the above conditional.

Recommendation
Check that all assets passed to `emergencyRedeem` are present in mapping `isInAssetList`.

AMENDED [2018-6-27]:
This issue is no longer present in commit `8cc187707652f87f9b2837687730a5cafc544f04`.

7. If statement incorrect in quantityHeldInCustodyOfExchange

In the `Fund` contract the function `quantityHeldInCustodyOfExchange` has an if statement:
Fund.sol / Line 647

```
if (exchanges[i].takesCustody) {
    totalSellQuantityInApprove += sellQuantity;
}
```

From our reading of the code, the value `totalSellQuantityInApprove` should only be increased for exchanges which don't take custody (and hence use `approve` instead).

Recommendation
Replace with
Fund.sol / Line 657

```
if (!exchanges[i].takesCustody) {
    totalSellQuantityInApprove += sellQuantity;
}
```

AMENDED [2018-6-27]:
This issue is no longer present in commit `8cc187707652f87f9b2837687730a5cafc544f04`.

Minor

8. Two stake transactions in a single block will fail

The `updateCheckpointAtNow` function assumes it will receive a single staking operation per block. In the case of two staking operations happening in the same block (in two separate transactions or otherwise) the function will fail on the second transaction.

StakeBank.sol / 132

```
Checkpoint storage checkpoint = history[length];
```

assumes that the length of `history` has been incremented in the previous if statement to ensure that `length` is a valid index into `history`. If this function is called twice in the same block this won't be true.

AMENDED [2018-6-27]:
This issue is no longer present in commit `8cc187707652f87f9b2837687730a5cafc544f04`.

9. Shares contract missing an ERC223 Transfer event

The `Shares` contract is intended to be ERC20 & ERC223 compatible, however function `transfer(address _to, uint _value)` emits only the ERC20-compatible `Transfer` event.

Recommendation

Replace

Shares.sol / Line 67

```
Transfer(msg.sender, _to, _value);
```

With

```
Transfer(msg.sender, _to, _value);
Transfer(msg.sender, _to, _value, empty);
```

AMENDED [2018-6-27]:

This issue is no longer present in commit `8cc187707652f87f9b2837687730a5cafc544f04`. Melonport has decided against using the ERC223 “standard”.

10. Missing check in enableInvestment

In the `enableInvestment` function, there should be a check that each asset in `ofAssets` is registered in the pricefeed, as a sanity check.

Recommendation

Add to `enableInvestment`

```
require(modules.pricefeed.assetIsRegistered(ofAssets[i]));
```

AMENDED [2018-6-27]:

This issue is no longer present in commit `8cc187707652f87f9b2837687730a5cafc544f04`.

Note

11. Inconsistencies in variable naming

There is some inconsistency in approach to variable naming. It would make the code more readable to use a consistent approach in naming, for example all variables passed as parameters into a function should start with an underscore.

Client’s response:

We will be conducting a greater refactoring shortly after our upcoming competition, and will keep this suggestion in mind when we go through the code, to adjust its style in a more consistent way. This will give us more time to discuss code style, and agree upon a standard way to name variables, among other things.

12. Inconsistent Solidity version

Most files specify: `pragma solidity ^0.4.21;`

However there are a few files that specify a different version:
competitions/Competition.sol: `pragma solidity ^0.4.19;`
competitions/CompetitionInterface.sol: `pragma solidity ^0.4.19;`
compliance/OnlyManagerCompetition.sol: `pragma solidity ^0.4.19;`

You may wish to consider pinning an explicit Solidity version.

AMENDED [2018-6-27]:
This issue is no longer present in commit `8cc187707652f87f9b2837687730a5cafc544f04`.

13. Inconsistent event handling

From version 0.4.21 onwards, the convention for events is to use `emit` followed by the event name and parameters. This is done in some places, but not in others. It would make the code clearer and more readable to follow this approach throughout the code base.

AMENDED [2018-6-27]:
This issue is no longer present in commit `8cc187707652f87f9b2837687730a5cafc544f04`.

14. Usage of `assert` vs. `require` / `revert`

`assert` should be used to establish invariants or otherwise check conditions which are never expected to be `false`. `assert` / `revert` should be used to check for user errors (e.g. passing in incorrect parameters) so as to not penalise users with high gas costs on failure.

This convention is not followed consistently, for example:
Fund.sol / 265

```
assert(AssetInterface(request.requestAsset).transferFrom(request.participan
t, this, costQuantity));
```

which should use `require`. Using `require` where appropriate (e.g. where failures are possible due to bad inputs) provides a better user experience by using less gas on failed transactions.

AMENDED [2018-6-27]:
This issue is no longer present in commit `8cc187707652f87f9b2837687730a5cafc544f04`.

15. Usage of `address(this)` vs. `this`

There is some inconsistency in approach to variable naming. It would make the code more readable to use a consistent approach in naming, for example all variables passed as parameters into a function should start with an underscore.
Whilst `this` can be implicitly cast to an `address`, it is more readable to use `address` consistently throughout the codebase rather than a mixture of the two styles.

This also applies to inconsistent usage of `address(0)` vs. `0` vs. `0x0`, for example:
Version.sol / 90

```
require(managerToFunds[msg.sender] == 0);
```

Competition.sol / 203

```
require(registeredFundToRegistrants[fund] == address(0) &&
registrantToRegistrantIds[msg.sender].exists == false);
```

AMENDED [2018-6-27]:
This issue is no longer present in commit `8cc187707652f87f9b2837687730a5cafc544f04`.

16. Redundant code

There are many instances of redundant code, usually where a condition is checked using a `pre_cond` modifier and then further checked in the body of the code. Whilst this doesn't cause functional issues, it increases the gas cost of transactions, making the overall protocol more expensive to use. Additionally, the sanity checks in Shares are made redundant by using DSMath which already checks for overflow.

Client's response:
Since the code is not harmed by these redundant safety checks, we decided to wait until after the upcoming competition for this one. Afterward, as we go through the code and notice these, we will remove unnecessarily redundant checks.

17. 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.

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.

18. Missing Transfer event when creating / annihilating shares

The `createShares` and `annihilateShares` modify the balances of users. They should emit corresponding `Transfer` events to and from `0x0` address to comply with the ERC20 specification and make transfers easier to track off-chain.

AMENDED [2018-6-27]:
This issue is no longer present in commit `8cc187707652f87f9b2837687730a5cafc544f04`.

19. The `getTimeTillEnd` function throws

If the competition has ended (i.e. `endTime < now`). It may be clearer to return 0 in this case.

AMENDED [2018-6-27]:
This issue is no longer present in commit `8cc187707652f87f9b2837687730a5cafc544f04`.

20. Ordering of functions is inconsistent

As per the Solidity style guide, constructors should appear before functions, whereas in `Competition.sol` the constructor appears after constant methods. Following a consistent approach to function ordering in Solidity files would improve the code readability.

AMENDED [2018-6-27]:
This issue is no longer present in commit `8cc187707652f87f9b2837687730a5cafc544f04`.

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. Overall the code is laid out in a very structured and comprehensible fashion. The code would benefit from following a consistent style guide with respect to variable naming, function ordering and events. The code implements complex logic, and interacts with third party contracts (e.g. exchanges, ERC20 tokens) in a trustful fashion which greatly increases its potential attack surface. Given the code complexity, we would recommend strongly considering a bug bounty in addition to audits.

AMENDED [2018-6-27]:
All major and critical issues have been addressed by Melonport.

It should be noted that `CanonicalRegistrar` is a single point of failure. If a malicious/compromised asset or module were to be registered, critical security breaches throughout the Melon protocol could easily occur. For the purposes of the competition Melonport is considered a trusted entity, however this is a substantial security assumption and if violated could result in major losses (e.g. if their private key were compromised).

Disclaimer