TCR Audit Report by ConsenSys Diligence

1 - Introduction

1.1 - Review Goals

The focus of this review was to ensure the following properties:

Security:
identifying security related issues within each
contract and within the system of contracts.

Sound Architecture:
evaluation of the architecture of this system through the lens of established smart contract best practices and general software best practices.

Code Correctness and Quality:
a full review of the contract source code. The primary areas of focus include:

- Correctness (does it do was it is supposed to do)
- Readability (How easily it can be read and understood)
- Sections of code with high complexity
- Improving scalability
- Quantity and quality of test coverage

1.2 - Summary

General Overview of Contract System

A token-curated registry is composed of a registry of entries, editable settings for the registry, and a voting system to approve or reject new registry entries and new settings.

Applicants submit their data and a required deposit to the `Registry` contract. The application remains in the apply stage to allow for challenges before it is included in the registry. A challenger must also transfer the required deposit to contract to initiate a challenge. `Registry` uses a `PLCRVoting` contract to hold a token-weighted vote on the registry entry. The winner between the applicant and the challenger is rewarded with part of the loser's deposit. The rest of the loser's deposit is split among the tokens that voted on the winning side. After an entry is accepted into the registry, it can be challenged at any time.

'Parameterizer' holds the settings for itself and the registry. Changes to these settings use the same `PLCRVoting` contract and the same challenge mechanism as `Registry`.

`PLCRVoting`'s complexity comes from the need to share tokens between polls using partial-lock commit-reveal voting. A `DLL` (doubly-linked list) contract stores the IDs of each user's active polls sorted by the number of tokens that were used to vote. When a vote is revealed, its entry is removed from the list. `PLCRVoting` prevents users from withdrawing tokens that would lower their balance below the number used in the active poll that used the most tokens, which is at the end of the list.

Overview of Findings

The contract system behaves as intended in rational scenarios, but the reliance on sane user input makes it possible for malicious proposals to lock up tokens or distribute them to unintended recipients. This requires those proposals to survive a challenge, which is difficult, but possible. In addition, users who interact with the contracts in unexpected ways can lock up their tokens.

1.3 - Materials Included in Audit

Documentation

Token-curated registries are documented in a series of blog posts, which were used as the reference documentation for this audit.

- Token-Curated Registries 1.0
- Dev Diary I: A Walkthrough of PLCR Voting in Solidity

- Dev Diary II: A Walkthrough of the adChain Registry TCR in Solidity

Dynamic Tests

The `tcr` codebase contains a comprehensive test suite, but our code coverage tool, `solidity-coverage`, throws an error while running the contract migrations before any tests can be executed. The tests execute successfully using the normal test runner.

Source Code

The audit covered the codebase as of commit `b2065612`. The second phase review after our recommendations covered the codebase as of commit `db40cd3c`.

The codebase pulls in ethpm packages `dll` and `attrstore` via Truffle. During the initial review, `PLCRVoting` was copied into the codebase from its original source, but was changed to be consumed as an ethpm package before the second review.

2 - General Findings

2.1 - State changes occur after token contract calls

Severity: Medium

Throughout the contract system, calls are made to the token contract followed by state changes in the contracts themselves. Token contracts can perform reentrancy attacks to take advantage of this.

Recommendation

Move token contract calls after all state changes in each function. Alternatively, document that TCRs must use trusted token contracts that cannot call other contracts.

https://github.com/skmgoldin/tcr/issues/17

Resolution

This recommendation was implemented in `PLCRVoting` commit `abba14c1` and `tcr` commit `8f4823f3`.

3 - Specific Findings

3.1 - Re-inserting the last node in the list creates a cycle

Severity: Critical

The changes in `52d97ca2` address most of the inconsistent list states from re-inserting an existing node, but one remains: if the last node is reinserted with itself as the previous node, the node sets its next and previous pointers to itself, and the actual previous item in the list isn't updated to point to the right place.

Recommendation

```
diff --git a/contracts/DLL.sol b/contracts/DLL.sol
index 18d4965..5226132 100644
--- a/contracts/DLL.sol
+++ b/contracts/DLL.sol
@@ -52,11 +52,13 @@ library DLL {
   */
   function insert(Data storage self, uint _prev, uint _curr, uint _next) public {
     require(_curr != NULL_NODE_ID);
-    require(_prev == NULL_NODE_ID || contains(self, _prev));
-
     remove(self, _curr);

+    require(_prev == NULL_NODE_ID || contains(self, _prev));
+    require(_next == NULL_NODE_ID || contains(self, _next));
+
     require(getNext(self, _prev) == _next);
+    require(getPrev(self, _next) == _prev);

     self.dll[_curr].prev = _prev;
     self.dll[_curr].next = _next;
```

Remove the current item from the list as early as possible in the function, right after the null node check. Since `_prev` will no longer be in the list, the `contains()` check will fail and the cycle won't be created. Make it harder to exploit lists that somehow get into a broken state by checking the invariants for `_next`, even though we expect them to always be consistent with the checks for `_prev`. Add a README that describes `DLL` as a list of unique elements.

https://github.com/skmgoldin/sol-dll/issues/2

Resolution

A failing test case for this issue was added in `3b5d3f56`, and the recommended fix was made in `91cb2448`.

The code from that commit was published as version 1.0.4 of the `dll` package in ethpm. The `tcr` repository was updated to point to that version in `924925ff`.

3.2 - getCommitHash is an unreliable proof that _prevPollID exists

Severity: Major

In `commitVote`, we check to ensure that the insert position (`_prevPollID`) actually exists by checking `getCommitHash(msg.sender, _prevPollID) != 0`. If a user sets their commit hash for that poll to zero, that check will fail even though it is a valid insertion position.

In changes that have been made since the code was frozen for the audit, `hasBeenRevealed` uses `getCommitHash` to throw if the hash is zero, which would prevent both `revealVote` and `rescueTokens` from succeeding if the user submitted a commit hash of zero, which would lock their tokens forever.

Recommendation

To distinguish polls that were never voted on from polls that were voted on and revealed, maintain attributes that are not controlled by user input. For instance, you could have `committed` and `revealed` attributes that are set to true when those events happen. Rather than use an attribute for `unlocked`, using `DLL.contains(pollID)` as the locked state ensures that tokens can always be rescued if they're in the list. This gives a few possible states:

- **all false**: never voted
- **committed, but not revealed or unlocked**: waiting for a reveal or a rescue
- **committed, revealed and unlocked**: user revealed on time
- **committed, not revealed, but unlocked**: user rescued tokens

`commitVote` needs to ensure that `_prevPollID` is the ID of a locked poll, so it would `require(DLL.contains(__prevPollID)`, then set `committed` to true before returning. Here's what would happen to each `hasBeenRevealed` call:

- `rescueTokens`: if `DLL.contains(_pollID)` and `isExpired(pollMap[_pollID].revealEndDate)`, remove the poll ID from the list
- `revealVote`: if `committed` is true and `revealed` is false, reveal the vote and set `revealed` to true
- `getNumPassingTokens`: if `revealed` is true, get the number of tokens they used to vote

Attempting to generalize `hasBeenRevealed` risks locking up tokens. Replacing it with explicit conditions for each case is safer.

https://github.com/ConsenSys/PLCRVoting/issues/22

Resolution

`PLCRVoting` commit `1fec53b8` implements this recommendation with mappings for `didCommit` and `didReveal`. These changes were added to `tcr` in commit `08ae65af`.

3.3 - Integer overflow in startPoll

Severity: Major

`commitEndDate: block.timestamp + _commitDuration` and `revealEndDate: block.timestamp + _commitDuration + _revealDuration` can overflow. This can result in polls that will accept votes, but never unlock tokens in the current code.

Recommendation

Use SafeMath for arithmetic.

https://github.com/ConsenSys/PLCRVoting/issues/25

Resolution

`PLCRVoting` commit `26579ec7` uses `SafeMath` for arithmetic in `startPoll`. The `tcr` repository was updated with this change in commit `845b0498`.

3.4 - pollExists is misleading and incomplete

Severity: Major

`pollExists` doesn't check if a poll exists. It attempts to check if a poll's stage times are reasonable, but misses polls that allow users to commit to a vote later than they can reveal their vote, which would lock their tokens permanently.

Recommendation

In `startPoll`, require reasonable stage times. Both must be greater than `block.timestamp`, and `commitEndDate` must be less than `revealEndDate`. Remove sanity checks from `pollExists` and just require that `_pollID <= pollNonce` so users can't vote on polls that haven't been started yet.

https://github.com/ConsenSys/PLCRVoting/issues/26

Resolution

As described above, `PLCRVoting` commit `26579ec7` replaces overflow-prone arithmetic with calls to `SafeMath`, which ensures that the stage times never overflow and produce unexpected values. `PLCRVoting` commit `a069730f` implements the recommended changes to `pollExists` with an additional check that the poll ID in question is not zero, which is an invalid poll ID. The `tcr` repository was updated with these changes in commits `b0c1374e` and `845b0498`.

3.5 - Parameterizer proposal deposits and challenge deposits can differ

Severity: Major

If `pMinDeposit` is changed after a proposal has been created, tokens will not be distributed properly.

If `pMinDeposit` is increased, `challenge.stake` and `challenge.rewardPool` will be based on a larger value than what the proposer deposited. Winning challengers will attempt to withdraw `(2 * challenges[_challengeID].stake) - challenges[_challengeID].rewardPool`, which will fail if the deposit has been increased enough to make the reward pool larger than the proposer's deposit. Even when it succeeds, the reward pool will be larger than the balance that remains the contract, so some voters won't get rewarded.

If `pMinDeposit` is decreased, `challenge.stake` and `challenge.rewardPool` will be based on a smaller value than what the proposer deposited. The winner will withdraw `(2 * challenges[_challengeID].stake) - challenges[_challengeID].rewardPool`, which will leave some tokens allocated neither to the winner nor the winning voters. The tokens will remain stuck in the contract.

Recommendation

In `challengeReparameterization`, don't reference `pMinDeposit`. Reference `proposal.deposit` so challenges are always consistent with their proposals.

https://github.com/skmgoldin/tcr/issues/15

Resolution

This recommendation was implemented in `tcr` commit `1e71a652`.

3.6 - Integer underflow in challengeReparameterization

Severity: Major

`100 - get("pDispensationPct")` can underflow, which would allow tokens to be drained from the contract.

Recommendation

Use SafeMath for arithmetic, and assert that these calculations succeed in processProposal so proposals can't get the contract in a state where the arithmetic for new challenges will revert every transaction.

https://github.com/skmgoldin/tcr/issues/19

Resolution

This recommendation was implemented in `tcr` commit `a9f0d97d`, along with a requirement for new proposals that the value for `pDispensationPct` and `dispensationPct` must be less than or equal to 100.

3.7 - Integer underflow in withdrawVotingRights

Severity: Medium

`voteTokenBalance[msg.sender] - getLockedTokens(msg.sender)` can underflow. This shouldn't underflow under normal circumstances, but detecting the underflow makes it harder to exploit abnormal circumstances.

Recommendation

Use SafeMath for arithmetic.

https://github.com/ConsenSys/PLCRVoting/issues/21

Resolution

This recommendation was implemented in `PLCRVoting` commit `c8df984d`, and in `tcr` commit `c8fa4889`.

3.8 - Integer overflows in proposeReparameterization

Severity: Medium

`now + get("pApplyStageLen")` and `now + get("pApplyStageLen") + get("pCommitStageLen") + get("pRevealStageLen") + PROCESSBY` can overflow.

Recommendation

Use SafeMath for arithmetic, and assert that these calculations succeed in `processProposal` so proposals can't get the contract in a state where the arithmetic for new proposals will revert every transaction.

https://github.com/skmgoldin/tcr/issues/18

Resolution

This recommendation was implemented in `tcr` commit `81c3274b`.

3.9 - Integer overflow in Registry.apply

Severity: Medium

`listingHash.applicationExpiry = block.timestamp + parameterizer.get("applyStageLen")` can overflow.

Recommendation

Use SafeMath for arithmetic. If the token holders set `applyStageLen` too high, all new applications will revert until they pass a proposal to change the setting.

https://github.com/skmgoldin/tcr/issues/27

Resolution

This recommendation was implemented in `tcr` commit `9d8b7af5`.

3.10 - Use EIP20Interface instead of EIP20

Severity: Minor

`PLCRVoting` and `Registry` should work with any ERC20 token, but they refer to a specific implementation of an ERC20 token. This doesn't have any effect on compatibility, but could be misleading.

Recommendation

Use EIP20Interface instead of EIP20.

- https://github.com/ConsenSys/PLCRVoting/issues/24
- https://github.com/skmgoldin/tcr/issues/25

Resolution

This recommendation was implemented in `PLCRVoting` commit `079c75d0` and `tcr` commit `d735858c`.

3.11 - Challenges with zero votes will succeed

Severity: Minor

PLCRVoting determines the winner with `(100 * poll.votesFor) > (poll.voteQuorum * (poll.votesFor + poll.votesAgainst))`. As a result, polls resolve as failing when they expire with no votes. This is probably the right behavior, but it's surprising.

The existing comments about this behavior are inconsistent.

Parameterizer.sol: `// Edge case, nobody voted, give all tokens to the winner.`

Registry.sol: `// Edge case, nobody voted, give all tokens to the challenger.`

Recommendation

Near the calls to `voting.isPassed`, document that either the challenge didn't get enough votes, or nobody voted at all.

https://github.com/skmgoldin/tcr/issues/16

Resolution

The comments in `challengeWinnerReward` and `resolveChallenge` for both `Parameterizer.sol` and `Registry.sol` were updated to be consistent in `tcr` commit `93d27093`.

3.12 - An unchallenged application cannot be cancelled

Severity: Minor

Whitelisted applicants can exit the registry when they discover new information that threatens

their deposit. New applicants cannot do so because `exit` calls `require(isWhitelisted(_listingHash))`. Even without an active challenge, a new applicant with new information cannot protect themselves from a challenge the way whitelisted applicants can.

Recommendation

If this is desired behavior, keep it. Just pointing out the discrepancy.

https://github.com/skmgoldin/tcr/issues/28

Resolution

No changes. This behavior will be documented.

4 - Third Party Findings

After the initial audit, the two issues below were reported by third parties. The fixes for those issues were included with our recommended fixes in our second phase review.

4.1 - Parameterizer proposal owner never gets token back if proposal goes unchallenged

> If a proposal is not challenged, and processProposal is called after the appExpiry date, but before the processBy date has elapsed, the proposal's value will be set, but the prop owner will never get their tokens back.

https://github.com/skmgoldin/tcr/issues/30

Resolution

This issue was resolved in `tcr` commit `106410c6`.

> - Add an integrity check in processProposal.js asserting that the proposer's balance is as-expected following a successful, unchallenged processing of their proposal.
> - Add a token transfer in the first clause of processProposal's if-else logic to the prop owner of their entire deposit amount.
> - Grab values for prop.owner and prop.deposit at the top of processProposal so that they can be used in both the first and third clauses.

4.2 - PLCRVoting edge case

This issue was reported in `tcr#40`, then clarified in `PLCRVoting` commit `1a4fea9a`:

> The getInsertPointForNumTokens function fails to provide correct insert points in a number of cases involving in-place updates.
>
> For in-place updates where the new value is greater than or equal to the old value, the function will return, incorrectly, the insert point as being the same pollID as the poll being updated.
>
> For in-place updates where the node being updated's token value is zero, the function will always compute, incorrectly, that the correct insert point is at the end of the list.

Resolution

This issue was resolved in `PLCRVoting` commit `1a4fea9a` and `tcr` commit `6ec8fcc6`.

Appendix 1 - Audit Participants

Security audit was performed by Niran Babalola and Suhabe Bugara.

Appendix 2 - Terminology

A.2.1 - Severity

Measurement of magnitude of an issue.

A.2.1.1 - minor

Minor issues are generally subjective in nature, or potentially deal with topics like "best practices" or "readability". Minor issues in general will not indicate an actual problem or bug in code.

The maintainers should use their own judgement as to whether addressing these issues improves the codebase.

A.2.1.2 - medium

Medium issues are generally objective in nature but do not represent actual bugs or security problems.

These issues should be addressed unless there is a clear reason not to.

A.2.1.3 - major

Major issues will be things like bugs or security vulnerabilities. These issues may not be directly exploitable, or may require a certain condition to arise in order to be exploited.

Left unaddressed these issues are highly likely to cause problems with the operation of the contract or lead to a situation which allows the system to be exploited in some way.

A.2.1.4 - critical

Critical issues are directly exploitable bugs or security vulnerabilities.

Left unaddressed these issues are highly likely or guaranteed to cause major problems or potentially a full failure in the operations of the contract.