Tornado Farm Smart Contracts and Circuits. Audit

Mikhail Vladimirov and Dmitry Khovratovich

1st September 2020

This document describes the audit process of the FARM smart contracts performed by ABDK Consulting.

1. Introduction

We've been asked to review the Tornado Farm smart contract and circuits given in private files.

2. Farm.sol

In this section we describe issues found in the Farm.sol.

2.1 Moderate Flaws

This section lists moderate flaws, which were found in the smart contract.

1. Line 142: it is not ensured that the rate for the given instances exists. If such a rate doesn't exist, `args.rate` = 0 will pass.
2. Line 171, 218, 221: returned value is ignored. By the EIP-20 standard, the token operations may return `false`, and a caller must be able to handle this in certain circumstances..

2.2 Suboptimal Code

This section lists suboptimal code patterns, which were found in the smart contract.

1. Line 32-34: these parameters should be indexed: `index, instance`.
2. Line 68, 85, 86: there should be `uint248` type.
3. Line 71, 87: there should be `bytes31` instead of `bytes32`.

4. Line 100-101: passing a single array of the structures with two fields would be cheaper and would make the length check unnecessary.
5. Line 115: there is no check for the case when values of the `_instances` array are unique. The rate could be overridden in this line.
6. Line 123: passing an array of structs wrapping bytes and `RewardArgs` would make the code more readable and probably more efficient:
   ```
   struct RewardProofAndArgs {
   bytes [] proof;
   RewardArgs args;
   }

   function batchReward(
   RewardProofAndArgs[] calldata rewardProofsAndArgs)
   ```
7. Line 125: decoding all elements at once as an array of structs would be more efficient:
   ```
   struct RewardProofAndArgs {
   bytes [] proof;
   RewardArgs args;
   }

   function batchReward(
   bytes calldata rewardArgs) external {
   RewardProofAndArgs [] memory rewardProofsAndArgs = abi.decode
   (rewardArgs, (RewardProofAndArgs[]));
   ```
8. Line 138, 192: the `extDataHash` argument is redundant and can be computed right in the line. Also, the `cutFirstByte(keccak256` actually calculates a custom 252-bit hash function. Consider implementing this custom hash function as a separate Solidity function. Something like this:

   ```
   function keccak252 (bytes memory data) public pure returns
   (bytes31) {
   return bytes31 (keccak256 (data) << 8);
   }
   ```
9. Line 141, 195, 196: changing type of the `fee` and the `amount` to `uint248` would make the `args.fee < 2**248` check unnecessary. Also, the `2**248` should be a named constant.
10. Line 142: probably, the `rate` parameter is redundant and should be taken from the `rates[]` directly.
11. Line 169: the `treeUpdateArgs.newRoot` value may be zero in some cases. Probably not when `args.account.inputRoot != getLastAccountRoot()`, but though. Consider adding an explicit check to ensure that zero value will never be used as account root.
12. Line 174-177, 224-226: logging four events (`AccountCommitmen`, `AccountNullifier`, `RewardNullifier`, `AccountData`) for a single

operation looks cumbersome. Binding these events together later could be hard, as there is no single key that could be used for this. Consider logging a single Reward event with all necessary information. Also true for the next three events: `AccountCommitment`, `AccountNullifier`, `AccountData`.

13. Line 124: the `treeUpdateArgs.newRoot` value may be zero in some cases. Probably not when `args.account.inputRoot != getLastAccountRoot()`, but though. Consider adding an explicit check to ensure that the zero value will never be used as account root.
14. Line 230: the values of the `_previousDepositRoot` and the `_previousWithdrawalRoot` parameters are ignored in case the `_deposits` and the `_withdrawals` are empty. Consider checking that in such cases both, the `_previousDepositRoot`, the `_depositRoot` and the `_previousWithdrawalRoot`, the `_withdrawalRoot` are the same as the current deposit root.
15. Line 240, 255: the `depositRoot` and the `withdrawalRoot` value was already read from the storage in the previous line. Consider reading once and caching in a local variable.
16. Line 274: the `cutFirstByte` name is confusing as the function returns the same number of bytes as passed as an argument. Consider renaming to something like `zero leading byte` or changing return type to `bytes31`. In the latter case the function could be implemented as `bytes31`(source << 8).
17. Line 281: the `isKnownAccountRoot` function wastes a lot of gas. Since the root index in the history array is fixed and known, it can be simply passed to this function as a separate argument.
18. Line 286-294: the loop could be simplified if `currentAccountRootIndex` would not wraped. See comment for the `insertAccountRoot` function (the next comment).
19. Line 344-346: the code would be simpler and probably more efficient if index would not be wrapped:

    ```
    accountRoots [nextAccountRootIndex++ %
    ACCOUNT_ROOT_HISTORY_SIZE] = root;
    ```

    Then account root check would looks like this:

    ```
    uint i = currentAccountRootIndex;
    uint j = i > ACCOUNT_ROOT_HISTORY_SIZE ? i -
    ACCOUNT_ROOT_HISTORY_SIZE : 0;
    while (i --> j) {
    if (accountRoots [i % ACCOUNT_ROOT_HISTORY_SIZE] == _root)
    returntrue;
    }
    return false;
    ```
20. Line 313-315:

- the `treeUpdateArgs.oldRoot == getLastAccountRoot()` implies that the `oldRoot` parameter is redundant.
- the `treeUpdateArgs.leaf == commitment` implies that the `commitment` argument is redundant
- the `treeUpdateArgs.pathIndices == currentAccountIndex` implies that the `pathIndices` is redundant

2.3 Unclear Behaviour

This section lists issues of the smart contract, where the contract behavior is unclear: the business logic might be violated here, but the documentation and functional requirements are not sufficiently documented to make a clear decision.

1. Line 229: the `updateRoots` function does not make any consistency check. Is it OK?
2. Line 268: probably the `setRate` function should emit some event. Also, there is no range check for the `_rate` parameter. Is zero value valid for it? If so, then it is impossible to tell whether the `_rate` for a given instance exists or not.
3. Line 274: does the `cutFirstByte` function need to be public?

2.4 Other Issues

This section lists other minor issues which were found in the token smart contract.

1. Line 1: Pragma Solidity version should be `^0.5.0` according to the common best practice, unless there is something special about this particular version. Also, the mainstream version is not 0.6.x, and 0.5.0 is legacy. Consider upgrading to 0.6.x.
2. Line 14: the next variables are unused:
   - `deposits`
   - `withdrawals`
3. Line 231: the `_depositRoot` parameter should be renamed to the `_newDepositRoot` for clarity.

3. Reward.circom

3.1 Critical Issue

It is not guaranteed that the withdrawal block is later than the deposit block. As a result the following attack is possible:

1. Alice deposits to Tornado with commitment $C=H_P(n_n,s_n)$ in block $B_1$.
2. Alice withdraws with nullifier hash $N_n=H_P(n_n)$ in block $B_2$.
3. Alice deposits to Tornado with commitment $C'=H_P(n_n,s_n')$ using the same nullifier $n_n$ in block $B_3$.
4. Farm owners adds these deposits and withdrawal to the Farm contract.

5. Alice provides a proof of reward using $C'$ as alleged deposit and $N_n$ as alleged nullifier hash of this deposit. This is possible since $C'$ uses the same nullifier as in $N_n$.
6. However, the reward equation now contains negative value $r(B_2-B_3)$.
7. If $v_I$ is zero, the equation underflows and output value becomes very big, but likely under $2^{248}$ for reasonably high $rates$ or hundreds of blocks between $B_2$ and $B_3$.
8. Alice drains the farm.

Note that there is no range check on the `rate` parameter either.

3.2 Minor issues

1. inputRoot better named oldRoot.
2. outputRoot better named newRoot.
3. depositCommitment parameter is redundant as it can be calculated directly from noteSecret and noteNullifier.
4. withdrawalNullifier parameter is redundant as it can be calculated directly from noteSecret and noteNullifier.
5. On dummy constraints: on our understanding, optimizer cannot remove public input even if it is not used in any constraints, as the value of this input will anyway be supplied to verifier and will not be ignored by it. So, probably this is indeed redundant.
6. How is the big input parameter for `main` computed? CIRCOM doesn't have named constants, but it may have constant functions, so consider extracting this value to a constant function

4. Summary

Based on our findings, we also recommend the following:

1. Fix the critical issue.
2. Pay attention to moderate issues.
3. Check issues marked “unclear behavior” against functional requirements.
4. Refactor the code to remove suboptimal parts.
5. Fix other (minor) issues.