SOLIDIFIED

Audit Report for Agrotoken - February 3, 2021

Summary

Audit Report prepared by Solidified covering the Agrotoken smart contract.

Process and Delivery

Three (3) independent Solidified experts performed an unbiased and isolated audit of the code below. The final debrief took place on February 3, 2021, and the results are presented here.

Audited Files

The source code has been supplied in the form of a private GitHub repository:

https://github.com/agrotoken-io/token

Commit number: `c2c1e771eb4f3b56a461320c8fea298c0252ae97`

Fixes have been supplied in commit number: `b71f2fe478f107ddd3ecdbf09e60220cae29c3e3`

Intended Behavior

The smart contract implements a token compliant with the ERC-20 (EIP-20) standard with the following parameters:

- Decimals: 4
- Symbol: AGTSoyBean
- Name: AgroToken SoyBean

The token adds some initial functionality, including:

- A fee is charged on operations
- A privileged admin role can mint tokens, and halt the token transfer until a certain block number
- A grain data structure keeps track of different grain contracts associated with token holder balances. These can only be modified by the contract admin.

The smart contract relies on a privileged admin account as a design choice, allowing this account to modify user balances, burn users' tokens, and add, remove and update grain contracts on the users' behalf.

Findings

Smart contract audits are an important step to improve the security of smart contracts and can find many issues. However, auditing complex codebases has its limits and a remaining risk is present (see disclaimer).

Users of a smart contract system should exercise caution. In order to help with the evaluation of the remaining risk, we provide a measure of the following key indicators: **code complexity**, **code readability**, **level of documentation**, and **test coverage**.

Note, that high complexity or lower test coverage does not necessarily equate to a higher risk, although certain bugs are more easily detected in unit testing than a security audit and vice versa.

Criteria	Status	Comment
Code complexity	Medium	Whilst code complexity is low compared to larger projects, it is unusually complex for an ERC20 token
Code readability and clarity	High	-
Level of Documentation	Medium	-
Test Coverage	High	-

Issues Found

Solidified found that the AgroProtocl contracts contain no critical issue, no major issue, 3 minor issues, in addition to 5 informational notes.

We recommend all issues are amended, while the notes are up to the team's discretion, as they refer to best practices.

Issue #	Description	Severity	Status
1	Insecure approve() Implementation	Minor	Resolved
2	Tokens transferred to Admin will be lost	Minor	Resolved
3	removeGrainContract() and updateGrainContract() will fail if _contractOwner account has less than grain.amount of tokens to be burned.	Minor	Resolved
4	The contract is admin controlled	Note	-
5	Higher fee is collected when a grain contract is updated	Note	Resolved
6	Fee charging tokens break integration with third-party protocols	Note	-
7	Tautology in changeFee()	Note	Resolved
8	Redundant modifier	Note	Resolved

Critical Issues

No critical issues have been found.

Major Issues

No major issues have been found.

Minor Issues

1. Insecure approve() Implementation

The ERC-20 standard has a security vulnerability related to the `approve()` method being vulnerable to race-condition. See https://github.com/sec-bit/awesome-buggy-erc20-tokens/blob/master/ERC20_token_issue_list.md#a20-re-approve for details.
The current implementation does not mitigate this.

Recommendation
Consider adding functionality to increase or decrease allowances, such as implemented by the Openzeppelin ERC-20 implementation: https://github.com/OpenZeppelin/openzeppelin-contracts/tree/master/contracts/token/ERC20
Alternatively an extra check can be added after line 470 to enforce allowance being reset to 0 first:

```
require((_value == 0) || (allowed[from][_spender] == 0);
```

Update
Resolved

2. Tokens transferred to Admin will be lost

The `transfer()` function when called from admin account will replace the from to AgroToken's contract address. Similarly, the `approve()` function will set allowance from to AgroToken's contract address when called from the admin account.

```
address from = (admin == msg.sender) ? address(this) : msg.sender;
```

The above code makes it impossible for the admin to transfer tokens belonging to his account.

Recommendation
There are several solution to this problem, such as changing the code to allow transfers from the admin account or adding the following check `isTransferValid()`:

```
 if (_to == address(0) || to == admin) {
            return false;
 }
```

In addition, it may be worth considering limiting the mint function to limit the admin role from minting tokens to its own address.

Update
Resolved

3. removeGrainContract() and updateGrainContract() will fail if _contractOwner account has less than grain.amount of tokens to be burned

---

The `removeGrainContract()` calls `burn(_contractOwner, grain.amount)`. If `_contractOwner` owner has less tokens than grain.amount, the `burn()` function will revert. There exists a workaround for `admin` to first mint the lacking tokens to the `_contractOwner` account before removing the contract. But it is unclear if business logic would allow that.

`updateGrainContract()` in some cases also calls the `burn()` function, thus it would suffer from the same problem.

Recommendation
There is no single solution to this issue with the current design. Since the issue seems a result of no keeping track of contract owners in the `Grain` data structure, a refactoring linking grain contracts with token holders could simplify the contract and solve this issue.

Update
Resolved

Notes

4. The contract is admin controlled

The admin role has significant privilege over smart contract operations, including the ability to add and remove grain contract data at will and halt the token. This also means that the admin can transfer and burn users' tokens at will.
The linkage between the `Grain` data structure and the users' balance without keeping track of the grain contract owner, also means that the contract depends on the admin to supply this information correctly.
Whilst this seems to be a deliberate design choice, end-users should be aware of this.

Recommendation
Document the privileged admin role clearly for your users or consider a less centralized design.

5. Higher fee is collected when a grain contract is updated

When a grain contract is updated, and if there is a difference in grain amounts, the function burns and mints the whole amount. Since both actions cost a fee, the user is charged for the whole amount twice instead of charging for the difference.

Recommendation
Consider changing the free mechanism to account for the difference.

Update
Resolved

6. Fee charging tokens break integration with third-party protocols

Tokens that charge a fee for operations are incompatible with a number of third-party protocols, such as the liquidity pools of many automatic market makers and other DeFi protocols.

Recommendation

Document this limitation clearly or consider redesigning the fee-system to not charge the fee on standard ERC-20 operations.

7. Tautology in `changeFee()`

Function `send()` checks for the following pre-condition:

```
require(_newAmount>=0 && _newAmount<=2, "Invalid or exceed white paper definition");
```

However, the parameter `_newAmount` is of type `uint256`, which by default is always `>= 0`.

Recommendation
Remove the unnecessary check `_newAmount>=0` for code clarity.

Update
Resolved

8. Redundant modifier

The `mint()` and `burn()` functions use modifiers `onlyAdmin` and `blockLock(msg.sender)`. However, the latter is redundant since `blockLock` does not apply to admin.

Update
Resolved

Recommendation
Remove the redundant modifier or change the locking mechanism.

Audit Report for Agrotoken - February 3, 2021

Disclaimer

Solidified audit is not a security warranty, investment advice, or an endorsement of Agrotoken or its products. This audit does not provide a security or correctness guarantee of the audited smart contract. 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.