Token Group Ltd. Patch Review

Feb, 2019

Contents

Chapter 1

Introduction

1.1 Scope of Work

Token Group Ltd. requested technical feedback on a series of patches applied applied to their Ethereum smart contract "wallet" implementation.

The purpose of this document is to describe implementation-specific defects present in the source files for the aforementioned smart contracts, along with appropriate mitigations and/or alternative architectural solutions, where appropriate. This source code review, like any source code review, is not an endorsement of the code or its suitability for any legal/regulatory regime, and it does not claim to be a definitive or exhaustive list of defects. Rather, this audit is an attempt to document discrepancies between the code's stated purpose (as described in written documentation and through verbal conversations with Token Group Ltd.'s engineering staff) and its behavior when correctly compiled and executed on the Ethereum blockchain.

1.2 Source Files

This review covers a subset of the code found in the following git commit:

`345761153883117b9b60e455f9f5f65d6df36047`

Specifically, this review only covers lines 531-545 of `contracts/wallet.sol` and all lines of `contracts/licence.sol`.

Chapter 2

Summary

2.1 Purpose

2.2 Architecture

2.2.1 Current Architecture

The `licence.sol` contract is designed to enforce that a fee is paid to a contract (`_tokenHolder`) as funds are loaded into a separate `_cryptoFloat` contract. The fee is determined by `_licenceAmountScaled`, which is an integer value representing the fee amount in tenths of a percent. The `_licenceAmountScaled` variable is initially set by the contract constructor, and can later be set by a contract matching the address in `_licenceDAO`. The `owner` of the contract can set `_licenceDAO` freely until `lockLicenceDAO()` is called.

Note that the implementation of `_cryptoFloat`, `_tokenHolder`, and `_licenceDAO` is not in scope for this review, and some of these contracts appear to be unimplemented at the time of review.

Chapter 3

Critical Defects

3.1 Deposited Ether is Lost

The `Licence` contract has a payable fallback function, but it does not provide any functionality that allows Ether to be withdrawn from the contract. The only Ether that is sent by the contract is through the `load()` function, which requires that the input and output Ether amounts are identical. Consequently, Ether deposited into the `Licence` contract is lost.

The `Licence` contract should not have a payable fallback function if there is no reason for the contract to accept arbitrary Ether.

Chapter 4

Moderate Defects

4.1 Sequential Calls to Attacker-controlled Code

On lines 169 and 170 of `licence.sol` and line 538 of `wallet.sol`, the code makes external calls to attacker-controlled addresses with unbounded gas input. Consequently, the code that receives those calls may call back into the calling contract and exploit any partially-comitted state from the parent call.

Although the code doesn't appear to have an exploitable reentrancy bug in its current state, it is possible that future changes could introduce a vulnerability. (Keep in mind that making calls to attacker-controlled code has reentrancy implications for every function in the calling contract, as well as every function in every contract that trusts the calling contract.) It is also notable that the implementation of the `_tokenHolder` and `_cryptoFloat` contracts may influence the exploitability of this defect, even though that code is outside the scope of this review.

Since the Token Group Ltd. wallet already has a token whitelist, that whitelist should be used to limit which tokens can be used with the `Licence::load()` and `Wallet::loadTokenCard()` functions. (We presume that Token Group Ltd. will only whitelist tokens that have been audited to a sufficient standard such that callers can be sure that transferring tokens does not lead to arbitrary code execution.)

4.2 Loading Card Consumes Arbitrary Ether

When `Wallet::loadTokenCard()` is called to send Ether to `licenceAddress`, it sends however much Ether is specified by the `_amount` argument, even if the transaction is sent with more or less Ether. Moreover, the call may send more Ether than the daily top-up limit to the card, which may not be desirable.

Token Group Ltd. should consider requiring that the `_amount` specified in `loadTokenCard()` match `msg.value`, and/or consider whether or not the Ether transfer should be governed according to the top-up limit rules. (If not, the documentation should include a rationale. Presently, a key disclosure plus card theft would allow an attacker to load the card with the entire `Wallet` balance.)

Chapter 5

Minor Defects

5.1 Counterintuitive Fee Math

The documentation for `_licenceAmountScaled` indicates that it represents the fee amount as tenths of a percent (i.e. 1 = 0.1%, 1000 = 100%). However, if `_licenceAmountScaled` is set to 1000, for example, then the arithmetic in `load()` will send 50% of `_amount` to the `_tokenHolder` address and the other half to `_cryptoFloat`. Concretely:

```
loadAmount = _amount.mul(MAX_AMOUNT_SCALE).div(_licenceAmountScaled
+ MAX_AMOUNT_SCALE);
           = _amount.mul(1000).div(1000 + 1000);
           = _amount.div(2);
```

In other words, `_licenceAmountScaled` is the percentage of the amount sent to the float address that is sent to the token holder address, rather than the percent of the total input sent to the token holder address. Consequently, it's not clear why the fee is arbitrarily capped at 100%, since it would be entirely possible under this scheme to require that 200% of the float address value be sent to the token holder address.

Token Group Ltd. should evaluate whether or not it would be simpler and clearer to store the fee ratio as a percentage of total input rather than *ex-ante* contribution amounts. The arithmetic for such a scheme would look like:

```
loadAmount = _amount.mul(_licenceAmountScaled).div(MAX_AMOUNT_SCALE);
```

Such a scheme would have the obvious outcome that a 100% fee would take 100% of `_amount`, and a 2% fee on an input `_amount` of 100 would be exactly 2.

5.2 Redundant Accessor Methods

The following accessor functions are redundant. They can be eliminated by declaring the variables that they access as `public`.

- `licenceAmountScaled()`
- `cryptoFloat()`
- `tokenHolder()`
- `licenceDAO()`
- `floatLocked()`
- `holderLocked()`
- `licenceDAOLocked()`

Removing the redundant functions above eliminates nearly 10% of the non-empty lines in `contracts/licence.sol` and more than 50% of the functions in the `Licence` contract.