BUG REPORT : Wrapper/Proxy contracts unable to deposit/withdraw tokens to GNTDeposit contract

Author : Philippe Castonguay

Date : 11.04.2018

Repository : https://github.com/PhABC/golem-contracts

Short Description : No wrapper / proxy contract can safely/realistically deposit and withdraw **GolemNetworkTokenBatching** tokens to the **GNTDeposit** contract.

Description

The **GolemNetworkTokenBatching** token contract allows depositing tokens to **GNTDeposit** contract via the `transferAndCall()` function. `transferAndCall()` calls `transfer(to, value)` and sets the **from** argument in `ReceivingContract(to).onTokenReceived(from, ...)` to **msg.sender**. On the **GNTDeposit** contract end, `onTokenReceived()` functions reset the `locked_until` period for **msg.sender** via `locked_until[_from] = 0` (since **from** is **msg.sender** when calling `transferAndCall()`). This is problematic because if **msg.sender** was to be a contract used by many people, the `locked_until[_from]` mapping could be continually reset to 0. This means that no contract acting as a proxy for many users could effectively deposit tokens to the **GNTDeposit** contract, since their tokens might be locked for a long, undefined period of time.

The `withdraw()` function on the **GNTDeposit** contract is also problematic if **msg.sender** is a wrapper/proxy contract calling `GNTDeposit.withdraw()`, since every time a user would try to withdraw token from the proxy contract, they would lock the tokens for all other users using the proxy/wrapper contract.

Examples of Proxy/Wrapper contracts

Wrapper/Proxy contracts that could want to deposit/withdraw **GNTB** tokens to/from the **GNTDeposit** contract could be numerous.

1. ETH to GNTB with ProxyExchangeContract contract

In order to improve the user experience, one could decide to write a contract that converts **ETH** to **GNTB** tokens via an exchange smart contract (e.g. 0x, KyberNetwork, Bancor, etc.) and *in the same transaction* deposit these newly acquired **GNTB** to the **GNTDeposit** contract. If this contract was used by many users, `locked_until[ProxyExchangeContract]` would always be reset to `0`, even if the users actually depositing the tokens are different. Indeed, **msg.sender** is always **ProxyExchangeContract**.

2. MigrateTokensAndDeposit Contract.

In order to exchange their original **GNT** tokens to **GNTB** tokens, users need to follow 3 steps;

1. Create an individual Gate for migration. The Gate address will be reported with the **GateOpened** event and accessible by `getGateAddress()`.
2. Transfer tokens to be migrated to the Gate address.
3. Execute `Proxy.transferFromGate()` to finalize the migration.

To improve the user experience, a **MigrateTokensAndDeposit** proxy contract could first create a gate (step 1) and then allow anyone to do **step 2** and **step 3** followed by depositing the newly obtained **GNTB** to the **GNTDeposit** contract, all in a single function call. This contract would make it easy for users to deposit their *original* GNT tokens to the **GNTDeposit** contract, effectively hiding part of the migration steps. Unfortunately, the current contracts do not allow this, since **MigrateTokensAndDeposit** would always have `locked_until[MigrateTokensAndDeposit]` set to 0 every time a different users uses this proxy contract. The same could be said if you had a `withdrawAndMigrateTokens()` function on a proxy contract.

These are only two simple examples, but useful contracts that try to deposit to the **GNTDeposit** would almost always have `locked_until[MigrateTokensAndDeposit]` set to `0`. Proxy contracts could use a queuing system to allow users to withdraw and deposit tokens and make sure they will eventually be able to, but this solution scales poorly. Another approach would be to have proxy contracts withdraw tokens from all their users at the same time at a time `t`, but this also doesn't seem optimal. One could also have a single proxy contract for each user (similar to the `gate` in the `TokenProxy` contract), but requiring each users to deploy a contract is *very* inefficient and such approach should only be used if necessary. This latter approach is especially problematic if storage rent fees are implemented on Ethereum (see https://ethresear.ch/t/a-simple-and-principled-way-to-compute-rent-fees/1455 for a recent discussion).

Attack scenario

Impact: High

No wrapper / proxy contract can safely deposit and withdraw GolemNetworkTokenBatching tokens to the GNTDeposit contract.

Components:

- **GNTDeposit.sol**
- **GolemNetworkTokenBatching.sol**

Reproduction:

Github Repository : https://github.com/PhABC/golem-contracts

Instructions :

```
npm install
npm test
```

MigrateTokensAndDeposit contract ; https://github.com/PhABC/golem-contracts/blob/master/contracts/MigrateTokensAndDeposit.sol

Flaw test script ; https://github.com/PhABC/golem-contracts/blob/master/test/ProxyContractCantDeposit.test.js

Details:

The test is an implementation of the **MigrateTokensAndDeposit** contract described above. In the test, one can see that Bob is resetting the lock period for all tokens that were deposited via the **MigrateTokensAndDeposit** proxy contract. Hence, if Alice unlocked the GNTB tokens that were deposited via proxy contract and tried to withdraw after the `withdrawal_delay` is passed, it's very likely the `time_lock[]` period would've been reset to 0 by another user depositing tokens via the **MigrateTokensAndDeposit** contract.

It is important to note that this is a trivial example of a proxy contract that could want to deposit **GNTB** to **GNTDeposit** and it is not meant to be a useful proxy contract. Nevertheless, you can see that when at least one user tries to deposit tokens via **MigrateTokensAndDeposit.sol* before the `time_lock[]` is passed, `time_lock[]` will be set again to `0`. With enough users, this would render the usage of the proxy practically impossible.

Fix

Add `transferFromAndCall()` to GolemNetworkTokenBatching.sol

One solution is to add the following function in **GolemNetworkTokenBatching.sol** contract.

```
function transferFromAndCall(address from, address to, uint256 value, bytes data)
external {
  // Transfer always returns true so no need to check return value
  transferFrom(from, to, value);

  // No need to check whether recipient is a contract, this method is
  // supposed to used only with contract recipients
  ReceivingContract(to).onTokenReceived(from, value, data);
}
```

This function allows proxy contracts to deposit *on the behalf* of users, where `time_lock[from]` is directly reffering to each user. Conforming to the ERC20 standard, the users now only need to set an allowance to the proxy contracts they are using. A new `transferFrom()` function could be implemented that would take a signed message from the users as an additional argument, removing the need to set an on-chain allowance to proxy contracts.

Fix test ; **https://github.com/PhABC/golem-contracts/blob/master/test/ProxyContract_CAN_Deposit.test.js**

This fix is not sufficient if you want to allow proxy contracts to also *withdraw* on the behalf of users from **GNTDeposit**.

This could be fixed by also adding `withdrawFrom()` in **GNTDeposit.sol** :

```
function withdrawFrom(address _from) external {
    require(isUnlocked[_from]);
    uint256 _amount = balances[_from];
    balances[_from] = 0;
    locked_until[_from] = 0;
    require(token.transfer(_from, _amount));
    emit Withdraw(_from, _from, _amount);
}
```

** (I did not write tests for this function, but I believe it works as intended).*

This is a very simple implementation, but this would allow contracts to withdraw on behalf of users. With both these changes, you could allow users to use proxies like an ETH-GNTB proxy contract, where users can send ETH to the deposit contract and retrieve ETH when withdrawing (ETH and GNTB are converted via the proxy contract). There are other examples where fixing the issue here described could be useful.