Numerai Token Contract Audit

May, 2017

Peter Vessenes and Dennis Peterson

Contents

1 Introduction

Numerai asked us to review their upcoming token contracts supporting the Numeraire system. New Alchemy reviewed the system from a technical perspective looking for bugs in their codebase. Overall we recommend minor feature enhancements and a few improvements which will reduce risks. Read more Below.

2 Files Audited

We evaluated the seven files at:

https://github.com/numerai/contract/tree/master/contracts

The github hash we evaluated was:

d9d0030716efc1bfc1929cbfc86aa37e067799b5

3 Disclaimer

The audit makes no statements or warrantees about utility of the code, safety of the code, suitability of the business model, regulatory regime for the business model, or any other statements about fitness of the contracts to purpose, or their bugfree status. The audit documentation is for discussion purposes only.

4 Executive Summary

This code is very clean, thoughtfully written and in general well architected. The code conforms closely to the documentation and specification – we loved reading it.

The code is based on OpenZeppelin in many cases. In general, OpenZeppelin's codebase is good, and this is a relatively safe start. We recently completed an audit on the OpenZeppelin codebase, and some of our findings apply to this codebase as well.

5 Vulnerabilities

5.1 Critical Vulnerabilities

We found no critical vulnerabilities.

5.2 Moderate Vulnerabilities

As written, the contracts are vulnerable to two common issues: a doublespend attack on approvals, and the short address attack.

5.2.1 Short Address Attack

Recently the Golem team discovered that an exchange wasn't validating user-entered addresses on transfers. Due to the way `msg.data` is interpreted, it was possible to enter a shortened address, which would cause the server to construct a transfer transaction that would appear correct to server-side code, but would actually transfer a much larger amount than expected.

This attack can be entirely prevented by doing a length check on `msg.data`. In the case of `transfer()`, the length should be 68:

```
assert(msg.data.length == 68);
```

Vulnerable functions include all those whose last two parameters are an address, followed by a value. In ERC20 these functions include `transferFrom` and `approve`.

A general way to implement this is with a modifier (slightly modified from one suggested by redditor izqui9):

```
modifier onlyPayloadSize(uint numwords) {
    assert(msg.data.length == numwords * 32 + 4);
    _;
}

function transfer(address _to, uint256 _value) onlyPayloadSize(2) { }
```

If an exploit of this nature were to succeed, it would arguably be the fault of the exchange, or whoever else improperly constructed the offending transactions. However, we believe in defense in depth. It's easy and desirable to make tokens which cannot be stolen this way, even from poorly-coded exchanges.

Further explanation of this attack is here: http://vessenes.com/the-erc20-short-address-attack-explained/

5.2.2 Approval Doublespend

Imagine that Alice approves Mallory to spend 100 tokens. Later, Alice decides to approve Mallory to spend 150 tokens instead. If Mallory is monitoring pending transactions, then when he sees Alice's new approval he can attempt to quickly spend 100 tokens, racing to get his transaction mined in before Alice's new approval arrives. If his transaction beats Alice's, then he can spend another 150 tokens after Alice's transaction goes through.

This issue is a consequence of the ERC20 standard, which specifies that `approve()` takes a replacement value but no prior value. Preventing the attack while complying with ERC20 involves some compromise: users should set the approval to zero, make sure Mallory hasn't snuck in a spend, then set the new value. In general, this sort of attack is possible with functions which do not encode enough prior state; in this case Alice's baseline belief of Mallory's outstanding spent token balance from the Mallory allowance.

It's possible for `approve()` to enforce this behavior without API changes in the ERC20 specification:

```
if ((_value != 0) && (approved[msg.sender][_spender] != 0)) return false;
```

However, this is just an attempt to modify user behavior. If the user does attempt to change from one non-zero value to another, then the doublespend can still happen, since the attacker will set the value to zero.

If desired, a nonstandard function can be added to minimize hassle for users. The issue can be fixed with minimal inconvenience by taking a change value rather than a replacement value:

```
function increaseApproval (address _spender, uint256 _addedValue)
onlyPayloadSize(2)
returns (bool success) {
    uint oldValue = approved[msg.sender][_spender];
    approved[msg.sender][_spender] = safeAdd(oldValue, _addedValue);
    return true;
}
```

Even if this function is added, it's important to keep the original for compatibility with the ERC20 specification.

Likely impact of this bug is low for most situations.

For more, see this discussion on github: https://github.com/ethereum/EIPs/issues/20#issuecomment-263524729

6 General Comments

6.1 Solidity Version Should Be Updated

The code specifies Solidity 0.4.8 or better. We recommend compiling with the latest version of Solidity, currenly 0.4.11.

Solidity 0.4.10 adds features which could be useful in these contracts:

- `assert(condition)`, which throws if the condition is `false`
- `revert()`, which rolls back without consuming all remaining gas.

The github mentions difficulty with Truffle; the latest version of Truffle updates the compiler to 0.4.11, which may resolve the issue.

6.2 onlyManyOwners and Multisignature Discussion

Many functions in this repository utilize the `onlyManyOwners` mechanics which OpenZeppelin pulled in from Gavin Wood's multisignature wallet.

6.2.1 Multisignature and Its Gaps

Key safety is critical for these functions, although not more critical than it would be if there were only one owner. Still, we recommend that the same security procedures and processes be put in place around each of the multisignature keys that would be expected for an `owner` type key.

Additionally, in general we recommend divvying up functional abilities in the code between mid- and high- security keys as a safer approach than just a raw multisignature. If the "high security" keys are kept physically secure except for rare uses, then compromise of the 'hot/mid-security' keys will have a bounded and knowable set of consequences, and will not be catastrophic.

We believe the best current balance between practical security and safety is to have a paper key for the owner that literally lives in a bank-quality safe, and to instrument common actions for safe usage by trusted, but not overly trusted daily use keys secured by something like a Ledger Nano S.

6.2.2 Some Gotchas With _required and _owners

May contracts listed use the OpenZeppelin constructor for `onlyManyOwners` - we recommend that additional logic be used. In particular, addresses in the `_owners` list should be at least checked to not be `0x0` and `_required` should be at least less than or equal to the length of `_owners`.

6.3 Upgradeability

The delegate structure does a good job of providing upgradeability.

This structure does give the owner substantial power; e.g. a new delegate could be empowered to adjust token balances in arbitrary ways. This makes key control especially critical for these contracts. We consider this a reasonable design since the Numerai system necessarily relies on substantial trust in the owner. In fact, it maybe even be worth considering an additional layer that allows the ledger itself to be swapped out.

6.4 Addresses as uints

The owner addresses are cast to uints. We don't love this, since it obscures intent and gives up some static typing. There's no particular reason not to use addresses directly.

There is one function noted below that allows only "addresses" consisting of numbers less than one million. We presume the purpose here is to allow only fake addresses that have no matching private key, since they are intended only as deposit addresses that allow withdrawal by the contract owner.

This is a reasonable design, and can still be accomplished by casting uints to addresses, instead of the other way around, thus isolating the casts to this one spot.

6.5 Reclaiming Stranded Tokens

We recommend instrumenting all contracts with the ability to call `transfer` on arbitrary ERC20 token contracts in case tokens are stranded there.

As an example, the REP token contract has been sent REP (presumably accidentally). We generalize this to Peterson's Law: "Tokens will be sent to the most inconvenient address possible."

6.6 Reclaiming Stranded Ether

It is very rare but possible to see Ether sent to a contract that is not `payable` through use of the `selfdestruct` function. For this reason we recommend allowing all contracts that are not holding Ether for other reasons to allow the `owner` to withdraw the funds. Both reclaims can be combined into a single function – we used this one for the TokenCard project:

```
function claimTokens(address _token) onlyOwner {
        if (_token == 0x0) {
            owner.transfer(this.balance);
            return;
        }

        Token token = Token(_token);
        uint balance = token.balanceOf(this);
        token.transfer(owner, balance);
        logTokenTransfer(_token, owner, balance);
    }
```

6.7 Pause / Unpause and Permanence

Our experience with TokenCard leads us to believe that it is likely a group managing a Token project may wish to pause the token transfers at some point. This could be in event of a critical bug, publication of an upgraded token contract, or other circumstance.

As we've discussed, this sort of control comes with regulatory and social burden. We have started to propose a middle ground for customers with finalizable pausing. In essence, we recommend these contracts be published supporting `pause` and `unpause`. We also recommend implementing a function that locks the pause status; we sometimes call it `neverPauseAgain()`. This function could be used to lock the contract open: as it is now, or closed: in the event it is defunct for some reason.

7 File By File Discussion

7.1 Shareable.sol

7.1.1 Line 55 onlyThis

`if (msg.sender == thisContract)` will never return `true`. The sender is always either an external caller or another contract. With a delegate call, the msg.sender in the called contract will be the external caller.

What is the intent here? This modifier doesn't appear to be in use anywhere, so we recommend removing it.

7.1.2 Line 86

There's no check to make sure there are at least as many owners as required, so it's possible to initialize in an unworkable state. This is mitigated by the presence of `changeSharable()`, but whenever possible it's best not to allow invalid states in the first place. The Zeppelin code includes this check:

```
if (required > owners.length) {
  throw;
}
```

The same validity check should be added to `changeSharable()`.

(Also note the inconsistent spelling "sharable.")

7.2 DestructibleSharable.sol

This file implements the `destroy()` function for any contract decorated `DestructibleShareable`. We don't recommend ever using destroy for contracts that may interact with tokens – after destruction they may not be able to `transfer` or engage with token contracts.

This is not desirable; consider a user that sends tokens each week to a `DestructibleShareable` contract, and is not notified the contract has been destroyed – all tokens sent to the contract will be lost after the `selfdestruct`.

This issue is compounded by the ERC20 standard which means recipients of `transfer`s do not need to accept a transfer. None of the numeraire contract functions are marked payable; we therefore recommend removing this from the repository completely.

See our notes about `onlyManyOwners` above.

7.3 StoppableShareable.sol

This file implements a simple "Stop/Release" functionality. We see no major issues with this file.

Merely for balance and beauty we suggest `release()` not be decorated `onlyInEmergency` - `emergencyStop()` is callable whether or not the contract has been stopped, so perhaps it would be nicer if `release()` could be called in either scenario as well.

We also recommend that you instrument events here for the stop and release; a server could listen for these events and trigger a number of urgent notifications if it sees these events triggered, like a slack notification, SMS or other actions.

See our notes about `onlyManyOwners` above.

7.4 NumeraireShared.sol

This file contains common variables for the `NumeraireBackend` and `NumeraireDelegate` files.

7.4.1 Line 11 - 12

Just for avoidance of a terrible error, we recommend instead

```
uint256 public supply_cap = 21000000 * decimal;
```

Re: `the disbursement_cap = 96153`... What is the meaning of this number? Is it just random? We spent some time factoring it and numbers near it, but can't find any obviously cool or notable things about it. A comment would be nice.

7.4.2 Line 17: Rename total_supply

If this variable were called `totalSupply` then the code would satisfy ERC20 rules for checking the supply automatically; Solidity would compile in the public accessor functions directly.

As a side note, it is not standard for ERC20 tokens to allow an internal audit of supply balances; instead `totalSupply` is kept as a `uint256` and the engineers are expected to make sure that the contract "balances". One exception is the MakerDAO contracts – they have decorators for functions which specify expectations for the balance change (if any) in `totalSupply`.

This is a good idea. The Ethereum VM is very careful to make sure that no Ether is created or destroyed during all VM operations, but token contracts do not receive these benefits.

7.4.3 Line 19: Rename balance_of

If this variable were called `balanceOf` then the code would satisfy ERC20 rules for checking the balance automatically; Solidity would compile in the public accessor functions directly.

7.4.4 Line 20: Rename allowance_of

If this variable were called `allowance` then the code would satisfy ERC20 rules for checking the allowance automatically; Solidity would compile in the public accessor functions directly.

7.4.5 Line 47: Event names should be given standard names.

In general we recommend using an event designator like `Log` ahead of events. This is not standard for ERC20 contracts, but it adds clarity for programmers working with the code, and for auditors.

So, `LogStakeDestroyed`, `LogStakeReleased`, etc.

Events specified in ERC20 should keep the standard names for compatibility with exchanges, client software, etc.

7.5 NumeraireBackend.sol

7.5.1 Line 26

There's no need for an overflow check here with the disbursement period hardcoded to one week. If then intent is to protect against the disbursement period being set too long, it should probably check for a span of time shorter than the age of the universe.

7.5.2 Line 31 disableContractUpgradability

Definitely a good idea to let owners shut down upgrading once they're confident the delegate requires no further changes.

7.5.3 Line 109 getStake

This and other places return dynamic arrays. This works fine as long as it's being called from client code (e.g. web3), and we presume that's the intended use. Just noting that it will not work to call these functions from other contracts.

7.5.4 Lines 121,124-125, 144-145 Math checks

This code makes checks whether math operations can be safely performed without overflow. Later the operations are performed separately. We have found no cases where the necessary checks were omitted, but the possibility of error can be removed by combining these operations with functions like `safeAdd()`.

One argument for *not* combining these operations might be if you want to return false instead of throwing when checks fail, but in these functions the code throws.

7.5.5 Lines 115-125 transferFrom()

As discussed above, for extra safety this should add a `msg.data.length` check.

```
function transferFrom (address _from, address _to, uint256 _value)
onlyPayloadSize(3)
returns (bool success) {
    //...
}
```

7.5.6 Lines 157-161 approve()

This should add both the `msg.data.length` check, and force the user to set the approval to zero before setting a new nonzero value.

```
function approve (address _spender, uint256 _value)
onlyPayloadSize(2)
returns (bool success) {
    if ((_value != 0) && (allowance_of[msg.sender][_spender] != 0)) return false;
    //...
}
```

7.6 NumeraiDelegate.sol

7.6.1 Line 27

Just to be sure, is disbursement is intended to replace the max weekly disbursement, rather than add to any unused disbursement from the previous week? Mentioning only because it'd be a trivial typo to forget the "+".

7.6.2 Line 49 unnecessary <

`<=0` isn't an actual problem, but note `<0` is impossible since these are uints.

7.6.3 Line 51 throws if resolving late

If owner misses the deadline, stake is stranded, it can't be destroyed or released. Is this an acceptable risk? How fast-paced will this be? What if there's a flood of transactions for a popular ICO right when you need to resolve some stake?

7.6.4 Line 164 max_deposit_address

Actual addresses are very unlikely to have values less than 1,000,000. We assume this is a way to enforce the use of fake addresses, for these deposits which are extracted by the owner. As mentioned above, we would prefer that these fake values be cast to addresses, rather than requiring all addresses to be case to uints.

7.7 Safe.sol

An interesting comment from Edcon was that the overflow behavior of Solidity is undocumented, so in theory, source code that relies on it could break with a future revision (though compiled code would be fine). Aside from this small caveat, these are fine.

As noted above, it would be more error-proof to use functions like `safeAdd()`, instead of checking `safeToAdd()` and then doing the math directly.