# TokenCard Audit

May, 2017

Authored by Peter Vessenes

## Contents

# 1 Introduction

In 2016, New Alchemy was retained to help TokenCard launch their product – a token backed debit card that could be used anywhere in the world. New Alchemy is well known as a top-tier provider of smart contract audits, but it was new to be contracted to write the smart contract code being deployed – after complaining about other people's code we had worries about karmic debt, and took steps accordingly.

## 1.1 Safe Audits Of Internally Written Code

Our number one concern for the codebase was that it securely implement the specifications; we did not want our audit function to be impaired by the specification and engineering process. Therefore we split the work; a functional team led by Dennis Peterson wrote the code, and I personally did the audit only when the team delivered the code as fit for purpose. This is known as a 'Chinese Wall' in some circles, and we followed these tight security procedures internally; I did not have any access, even read-only access to the repository until the audit began.

These constraints might seem extreme, but we are very focused on maintaining safety for all clients, and the caution is warranted for any software that will be asked to secure customer funds.

## 1.2 Long term goals for the codebase

ERC20 contracts are deceptively simple to implement – the API is short, and the general functionality encapsulated is minimal. However, in the last year a number of small-scale vulnerabilities and 'gotchas' have been published or discovered. It was the goal of the New Alchemy team writing the code to deliver an up to date 'safer' ERC20 compliant token. While I have some small quibbles and thoughts for future improvement in architecture, I can say that overall the codebase is excellent, well engineered, and succeeds in these goals.

## 1.3 Methodology

I double audited the code, first working through our internal list of known attacks on smart contracts and ERC20 contracts and verified that the contracts were not vulnerable to these attacks. Attacks considered include recursive calls, over and underflow errors, replay attacks, reordering attacks, the so-called "double cross" and "single cross" attacks and Solar storm style process injection attacks.

I then re-audited the files line by line, usually bottom up to maintain as little 'flow' that might cause complacency while reading as possible and worked to read and audit each small block of code carefully. Nevertheless I may have missed one or more small or large problems; this is why we publish the smart contract code for community review.

# 2 Files Audited

The files audited have been published at https://github.com/MonolithDAO/token. The commit hash of the code I evaluated is 5d2c2f56f695f60f1873f57a3a35fa9e8a9b4754.

The repository uses Dapple for testing and implements over 120 test functions. I did not audit the tests.

The Auction.sol contract is intended for use later, will not be deployed at launch and may be modified in the future, therefore I did not audit it.

The complete list of files audited is:

- Common.sol
- FirstSale.sol
- ICO.sol
- ICOControllerMonolith.sol
- Token.sol

## 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

Overall this is a super ambitious ERC20 contract and Token Sale contract that is well conceived and well executed. This contract is a good model for a modern ERC20 contract with various protections built in; as of May 2017, it is *best in class* for features delivered.

The contract provides for the following interesting features and use cases:

1. 18 month lock for token issuers Monolith Studio
2. A 'reserve' functionality allowing future sales as needed
3. Resistance to recently published attacks on allowances and short message addresses
4. Stubs for the burn mechanism as described in the white paper (although see notes below)
5. Ability to reclaim tokens transferred to the wrong contract (See "Peterson's Law" below)
6. Live test harness for blockchain integration tests.
7. Ability for the admin to recover unclaimed refunds or un-issued tokens (for future sales that have auction or refund mechanics)
8. Approval mechanics for ownership transfers, minimizing 'fat finger' risks that could lock the contract

Over 120 dapple tests are implemented, and the repository passes all of them.

In summary, I consider this contract safe for use and built with the long term in mind. At launch I will have some of my personal funds secured by the contract.

## 5 Discussion

The initial internal audit turned up three critical vulnerabilities and two moderate ones. All of these were from last-minute functionality changes; a good reminder that smart contract creation is best done carefully and thoughtfully. This audit reflects the amended code which does not have these vulnerabilities. Nevertheless, the discussion here does not whitewash complaints or niggles I turned up during the audit process.

## 5.1 Critical

I found no critical bugs or vulnerabilities.

## 5.2 Moderate

I found no "Moderate" bugs, but see discussion on the Owner's Rights during and after the initial token sale period. In particular, reserve tokens and TokenHolder mechanics are worth paying attention to.

Overall the safety of these contracts will increase when `owner` is set to a multisignature wallet; I recommend this be done after completion of the token sale.

## 5.3 Minor

A number of common minor vulnerabilities **do not** appear in this repository; I'm pleased with the overall attention to detail shown here.

### 5.3.1 Approved Ownership Changes

It is in my opinion great to have an approval mechanic for ownership change, but it is worth talking about the security trade-offs. In the very unlikely case of racing an attacker who has compromised the `owner` key but not changed the `owner` yet, requiring approval from a new `owner` address may slow down seizure of the contract from the attacker. It is hard to imagine an attacker that is sophisticated enough to engage as the `owner` after key compromise but not sophisticated enough to change the `owner` at the first chance, though.

In exchange for this very slight reduction in security, you get what seems to me to be a big benefit - nobody, including an attacker with the `owner` key – can lock the contract by changing the `owner` to an invalid address. You are always guaranteed that the new owner can sign transactions that are function calls. I think this is a good trade-off, and recommend this idea in general.

### 5.3.2 Test Coverage Reports

For *even better* safety and security I would like to see a test coverage tool report applied to the 100+ dapple tests in the repository. I'm not aware of one that claims to be in 'solid' shape. The closest is probably https://github.com/JoinColony/solcover, which uses truffle. In general, test coverage, while not a silver bullet would add another layer of comfort, and I believe will be a standard part of high quality Solidity development by the end of 2017.

## 5.4 Owner's Rights

The owner has two sets of rights that are unusual in comparison to other ERC20 contracts: these revolve around the Reserve Tokens and the TokenHolder contract.

### 5.4.1 Reserve Tokens

The smart contract issues reserve tokens for future sales. The white paper specifies a calculation for how many reserve tokens are issued; the smart contracts correctly implement this math. The reserve tokens are not actually issued at launch, they are kept as a variable that can be debited later and turned into newly issued tokens.

These additional issuances later are termed "sales" in the codebase. There are few restrictions on how these future sales may work. This is probably sensible – use cases and discussion with the community will inevitably adjust the details of how they work. However, this means that the owner could create a "sale" that merely granted them the tokens, or used them for some other objectionable purpose.

This flexibility for the owner is called out in the white paper and disclosed appropriately.

If the `owner` key is compromised, one angle of attack that would be to create a sale, then grant all tokens in the sale to the attacker. This would have the effect of lowering the claimable amount of tokens held by the Asset Contract, in effect diluting TKN users pro-rata. This would be bad although not world-ending, and for this reason I recommend the `owner` contract be made a multi signature key and that solid physical and digital security processes are implemented.

In general the `owner` of the contract need almost never sign transactions, and so these keys could be kept in a "cold storage" style situation – the preferred situation without a doubt for keys that have this power.

### 5.4.2 TokenHolder

The TokenHolder contract is intended to be the final destination of the tokens taken as a fee during the operation of the TokenCard project. This contract is not yet implemented. This is fine right now – operations are slated to start in September, 2017 – however, the TokenCard contract will be trusted to fairly implement the `burn` mechanism. Further, the owner is expected to set the `TokenHolder` address, effectively choosing the functionality, at a later date.

This `TokenHolder` contract will need to be carefully audited – worst case, it may `transfer` too many of the held tokens to other parties, or refuse to `transfer` to those burning.

Further, there is an option to `lock` the contract; this is probably desirable for safety – once the TokenHolder contract is finalized, attack surface can be minimized.

In the interim, however, if the `owner` key is compromised, calling `lock` would be truly terrible, necessitating at the least a republishing of the ledger. I recommend the TokenHolder contract get published and audited quickly, and that additionally the `owner` key be set to a multisignature wallet expeditiously.

### 5.4.3 Reclaiming Lost Tokens

One of the sale mechanisms considered for the future would be an auction – for some sorts of auctions, refunds may be due, or partial (pro-rata) token issuance may be appropriate.

In these situations, the contracts may have ETH or other tokens that they should not continue to hold on to, for instance all of a bid in an auction was not used; however to make the bid binding, a user sent the full amount of ETH.

This is a fairly difficult problem to deal with – for instance, the Dfinity contributions did not even attempt to address the possibility of refunds, instead just accepting all contributions over a short one day period. Consider attacks on ERC20 contracts like TheDAO in which an attack wallet was constructed (and triggered logic when it received ETH), and this endeavor becomes even more fraught.

These contracts lay groundwork for a solution, although there is no current precise plan to utilize them. The contracts allow the `owner` to claim any unclaimed ETH after a one year waiting period. This is a mechanism to deal with wallets that have unusually high gas fees (for good or bad reasons) or have abandoned funds.

This is probably a reasonable trade-off – in the future, if multisignature wallets become more popular it may be worth amending the `claim` function to allow additional gas to be offered. For now, most wallets do not require additional gas in the fallback function, and this is a reasonable security trade-off.

Note the initial sale does *not* have any situation in which tokens or ETH may be `claimable`. A full discussion of these mechanisms is beyond the scope of the audit, but readers are encouraged to engage directly with New Alchemy or Monolith with questions.

## 5.5 On token value calculation

A fundamental requirement for the Token Sale was to take tokens as part of the contribution phase. This is described as being both for social reasons and practical – the TokenCard project will want tokens to market make for sales from card swipes, and they wanted to help the token ecosystem in general.

Because ERC20 token transfers do not trigger any outside functions, there are a number of open questions to be answered by the token sale contract, including:

1. What to do if outside tokens are unwanted / unsupported?
2. What value to assign outside tokens when contributed?
3. How to trigger acceptance of a token?

The Token Card contracts choose to trust an outside service / oracle for answers to these questions – in brief, a trusted listener waits for a `transfer` event on a supported token, then calls `depositTokens` with a computed ETH value for the deposit, and a reference ID.

I would term this methodology “trust and verify (without teeth)”. There are no mechanisms in the smart contract to revoke minted tokens if a calculation goes wrong on the server side. There is likely no ‘correct’ value for a token transfer – spot prices vary by market for tokens, and large transfers might be worth less if sold directly than the spot price, especially if thinly traded.

In this scheme, any observer of the smart contract events can validate that the tokens are credited and assess if the credit is reasonable. There is no recourse in the smart contract in case of a ‘mis-credit’.

This is not perfect, but seems to me to be a reasonable set of trade-offs. The developers working on this portion of the smart contract code suggested a more complex set of operations, including a commit and validation step for token transfers, but it was ultimately rejected as too much friction for token holders. The final scheme implemented here has the great virtue of allowing a simple `transfer` from any token wallet without additional work.

I believe an amended ERC20 standard that allowed receiving addresses to ‘accept’ transfers would have utility in many circumstances, and would encourage industry leaders to put this feature as a possible one in any conversations for a second generation token standard.

### 5.6 Peterson's Law

Dennis Peterson instrumented the contracts to be able to reclaim ERC20 tokens sent inadvertently to the wrong place. I love this idea, and have dubbed it Peterson's Law:

> Tokens will be sent to the most inconvenient address possible.

For instance, at time of writing the REP contract itself holds over 200 REP. These are burned REP; as far as I know the REP ERC20 contract has no mechanism of transferring REP held by itself. This leakage is going to become worse over time; old contracts without the ability to call `transfer` on arbitrary token contracts will never be able to claim their tokens. At any rate, kudos to Dennis Peterson for considering this now, before it is too late.

## 6 Line by Line Comments

Included below are the line by line comments and notes I delivered to Dennis Peterson as part of the audit.

### 6.1 Common.sol

#### 6.1.1 Line 22: Refactor?

If the Constants contract really only has DECIMALS in it, I would recommend a re-factor – not clear this is worth the blockchain space savings.

#### 6.1.2 Line 36: Testable comments

I really like this idea of instrumenting some on-chain testing; while software like testrpc and test networks like Ropsten can provide some comfort, it makes good sense to me to be able to control a contract in this way.

That said, I think it is worth being extraordinarily careful to make sure standard users don't end up using a contract in `testing = true` mode. The functions here are careful to only be usable during testing; this is accomplished with the `onlyTesting` decorator.

I recommend that the `testing` bool be made public so that later functionality can be instrumented which checks the state of the contract. (This was done in a subsequent commit)

I do not believe a contract can be removed from `testing` mode after review of the code; this is an additional safety measure, which when combined with the `testing` test could provide safety in case of accident or malicious use of the `testing` designator.

### 6.2 Token.sol

#### 6.2.1 Line 23: Resistance to short address attack

This is nice; it protects from a short address being offered. It's a little esoteric, but I commend the author for including protection from attacks only publicized in the last few weeks.

#### 6.2.2 Line 84: resistance to approve racing

This is nice! It keeps an approver safe from getting raced to a double spend on approval. This could prove to be a very valuable protection for TokenCard's intended heavy use of the ERC20 approval mechanism. I'm not aware of any other ERC20 contracts that implement this protection.

#### 6.2.3 Line 123: Do not use this function

The TokenHolder contract is not complete yet; if `lockTokenHolder()` is called too early, this will be a very bad day. I recommend that `owner` be a multisig wallet to help provide protection from an accidental or malicious call of this function.

#### 6.2.4 Line 136: Burning

Burning is a great mechanic. It makes me nauseous, so I reviewed this many times. I believe that Line 140 does not need the `safeSub` function. But an excess of caution seems quintuply warranted here. A malicious TokenHolder contract could inaccurately report it has burned, and this function will continue on merrily. This underscores the need for a thorough audit of the final TokenHolder contract.

#### 6.2.5 Line 145: TokenHolder

This contract does not comply with the cash and burn specification; it is a placeholder for a future contract implementing the white paper specification. I would recommend this `throw` for safety; in any event the burn function in `Token` will throw on a false return from the function.

### 6.3 ICOControllerMonolith.sol

#### 6.3.1 Line 28: Owner control

The owner of the controller can be changed any time, and repeatedly. Probably the best bet, but not without risks.

#### 6.3.2 Line 63: currTime

Note that `currTime()` can lie if in testing mode; this is the reason for avoidance of the direct request for the block time.

### 6.4 FirstSale.sol

#### 6.4.1 Line 53 Geometric vs linear price steps

Each succeeding step provides a percentage-wise greater drop in value than the prior step. This may be intended, it is specified in the whitepaper. If a constant percentage drop were desired going from 150 to 100 in six steps, then a decay rate of

$$e^{\frac{\log \frac{2}{3}}{5}} = 0.9221079...$$

would be used, yielding

$$\{150.0, 138.316, 127.542, 117.608, 108.447, 100.0\}$$

for the payouts.

## 6.5 ICO.sol

### 6.5.1 Lines 11 -20

Why use mappings for `saleMinimumPurchases`? Since we have a Sale object in a list already, makes sense to combine?

### 6.5.2 Line 32: Who is In Charge?

The contract is owned and payable to the `msg.sender`; makes sense.

### 6.5.3 Line 38: changeOwner

Would be nice to stop a foot fault / attack by making sure at least that `newOwner` isn't `0x0`. That doesn't stop someone from making up a malicious `newOwner` but it would stop a fat finger function call.

### 6.5.4 Line 47: Out of order execution of changePayee

The comments note that `changePayee` should be called before the owner is changed. It would be nice to see this enforced in code.

### 6.5.5 Line 59: Test mode

This is a nice idea. I think it would be nice to make `testing` public (Line 38: Common.sol) so that users can easily see what mode the contract is in.

### 6.5.6 Line 118: deposit

I'd like to see some documentation here or somewhere about the `deposit` plan and operational steps.

I get that the main flow is:

1. Fallback function is called (due to ETH send)
2. `deposit()` is called.
3. `doDeposit()` is called.

How do we get tokens into `doDeposit`? Looks like we enter in from a separate flow, callable by `owner`.

Why is `deposit()` payable? shouldn't all ETH come in from the fallback?

#### 6.5.7 Line 125: Great logging!

I applaud the use of the log prefix for this event.

#### 6.5.8 Line 129: doDeposit

This all looks good to me.

#### 6.5.9 Line 177: Depositing

I can't tell if I like the idea of IDing tokens by address. I guess it is very precise. But it puts burden on the front end. I guess that's the right place for it.

#### 6.5.10 Line 200: safebalance and topUp

These nicely considered functions make sure the last withdrawer in a refund doesn't have problems. I would suggest that anyone be allowed to trigger `topUp` in case of admin griefing.

#### 6.5.11 Line 219: Top up calculation

The top up methodology seems a little janky to me. I believe it works, and in any event it is solely controlled by the `owner` who has many greater powers. But I would recommend this get redesigned for a future version.

#### 6.5.12 Line 233: claim() and timing

So this relies solely on `claimable` knowing that a token sale is finished. I think I'd be more comfortable if this was decorated `noSaleInProgress` or some such decorator.

#### 6.5.13 Line 253: Admin powers with claimFor

This function makes an admin wait a year and then allows her to claim any unclaimed tokens. This is a mechanism to deal with expensive wallets and abandoned funds.

Any user with an overly expensive wallet can appeal to an admin for help in the form of the admin sending to an intermediate (or new) address. The admin will be unable to do this for one year from the end of a sale.

This is probably a reasonable trade-off – in the future, if multisignature wallets become more popular it may be worth amending the `claim` function to allow additional gas to be offered. For now, most wallets do not require additional gas in the fallback function, and this is a reasonable security trade-off.

This function relies on `getRefund` and `getTokens` heavily.

#### 6.5.14 Line 315-323 Recursive call protection in claimOwnerEth

This function implements a custom mutex by checking if `ownerClaimed[salenum]` is true, then setting it true before the `call` in 321. This is fine. I might slightly prefer a standard `mutexed` or `singleCall` decorator in the function signature. In any event, this checks properly and is not vulnerable to a recursive call on the `claimOwnerEth` function.

#### 6.5.15 Line 330 Comment and Token Transfer

I hereby dub this "Peterson's Law": Tokens will be sent to the most inconvenient address possible.

This is a very fine bit of consideration about the future. I strongly commend this thinking.

Note that line 337 assumes the token's `transfer` function works largely like ERC20 tokens, and has the same function signature. Admins should be cautious using this function for non ERC20 compliant tokens, or token contracts they do not trust.

#### 6.5.16 Line 388: Best comment in the repo

I like this disclaimer: "If you start actually calling this refund, the disaster is real."

Note that this refund function is **only** for emergencies; most of the contracts are decorated `notAllStopped`. It does still allow token transfers. Whether this is a good idea is probably a matter for philosophers; invoking Peterson's law, the contract should be able to reclaim tokens forever, but opinions may vary.

I commend the thinking behind a 'method of last resort' for ending the contract and granting emergency refunds. This is good. On the other hand, I would recommend this function be harder to call.

Two things come to mind – a 'safer' address than the owner address could be designated and stored on paper, perhaps in pieces. This would slow response in the event the owner key is compromised, but also limit the owner's rights in a safe way.

Alternately, this function could have a timelocked "are you sure?" function that would require a cooling off period. This raises other attack vectors to consider. Either would *probably* be preferable to this as it stands.

## 7 Notes on Some Vulnerabilities Not Present

### 7.1 Short Address Attack

The codebase fixes a common vulnerability in ERC20 tokens; some explanation on the vulnerability is included here.

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.

Because dividend-paying tokens execute additional code on `transfer` we think this issue is worth looking at more carefully than most token contracts, and recommend that at very least the `assert` be added to the code.

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

## 7.2 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, but exchanges without preferred mining contracts may wish to consider carefully their `approve` workflow.

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

## 8 Contact

New Alchemy provides full stack services for companies and individuals engaging in the Token ecosystem. Please contact us at hello@newalchemy.io!