FunFair Token Contract Audit

June, 2017

Authored by Peter Vessenes

Contents

1 Introduction

In 2017, FunFair retained New Alchemy to manage their token creation event, advising on mechanics, issuance and strategy, and implementing the underlying software for the token sale itself as well as the ledger.

Token contracts rightly deserve audits. New Alchemy has a long list of excellent customers that have taken advantage of our audit services, and we wanted to extend these services to FunFair with the rest.

However, we did not want the same engineers to write the code as would audit it; this defeats the purpose. Typically external audits are indicated, however because FunFair's technology can be used in the gaming industry, our preferred list of outside vendors was unavailable.

Further complicating an internal audit is the desire by most to be nice to their co-workers. We therefore proceeded with a 'Chinese Wall' style system. I took no part in the coding of the smart contracts, and only reviewed code when it was deemed 'ready' by the team. As the Managing Director, I am willing to anger internal engineers with caustic audits (although I had only one significant complaint).

1.1 Design Goals for the Codebase

ERC20 contracts look 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. Further, experiences with TokenCard and other clients show that there is real need in the near-term for facilities to pause, update and enhance functionality around the token contract without disturbing the ledger of balances.

The Token creation process can be fraught as well; recent ICOs have brought the Ethereum blockchain to its knees often in the last months. I expect to see more of this for some time, so a contract with minimal gas requirements was desired.

The design document and white paper published at https://funfair.io details the New Alchemy team's goals for the FunFair contracts. My assessment is that they have generally succeeded in their goals.

1.2 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, short data attacks and logic errors.

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 source code used to create these contracts has been published to etherscan and verified, and I used the source code published at etherscan for the review.

The contracts have been published to these addresses:

- Token 0xbbb1bd2d741f05e144e6c4517676a15554fd4b8d
- Controller 0x6f444af44be5c398f57a2016d191e01ffd8f6931
- Ledger 0xe6a51bd48f93abcd6c1d532112094044971d8d4e
- Creation/ICO: 0x55c44fbad82686afb0ca41cefb8d086cb937b2e6

The actual code repository uses Dapple for testing and implements many test functions. I did not audit the tests.

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

I found no critical or moderate vulnerabilities in the code.

Overall this is an ERC20 and Token Sale contract set that values upgradability, stability and correctness. It is well executed. This set of contracts is a good model for a modern ERC20 contract and token sale with various protections built in. It would be suitable for any group that wishes to enhance functionality at a later date. It would not be suitable for groups that must stay completely 'hands off' controlling their token contract – the owner of the contracts will have significant rights over the contract until they choose to release them.

For more on owner rights, read the "Owner Rights" section below.

4.1 Contribution Contract

The contribution contract is advertised as having 'constant gas under all circumstances except the final contribution'. This appears to be accurate. The ICO Contract is very simple.

In order to meet the design goals of constant, low gas for the token creation event, the team chose to offload processing and crediting to a server tool written in go. The tool generates logs and other files to aid audit; I believe that FunFair will publish these logs as part of the completion of the crowdsale process.

This is a reasonable design decision, and the thinking behind it is documented at length in the FunFair whitepaper. In brief, the desire to credit Bitcoin, Zcash and ERC20 token transfers for contributors meant server crediting was required as a feature; doing all crediting in this way allowed for simpler code.

I would prefer to see these happening on chain, but I'm unaware of any realistic technologies that allow this to happen right now.

4.2 Nice Features

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

1. Resistance to recently published attacks on allowances and short message addresses
2. Ability to reclaim tokens transferred to the wrong contract (See "Peterson's Law" below)
3. Approval mechanics for ownership transfers, minimizing 'fat finger' risks that could lock the contract
4. Highly considerate of the blockchain, minimizing gas in safe ways wherever possible
5. Upgradability of the Controller logic, and the Ledger contract if necessary

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 Vulnerabilities Discussion

The initial internal audit turned up two moderate vulnerabilities. Community member "bokkypoobah" identified at least one minor vulnerability, and will receive the first FunFair bug bounty.

For reference, the vulnerabilities found in prior versions were all logic bugs: finalization had some edge cases which resulted in suboptimal choices for the contract owner. Bokkypoobah turned up a logic error in when the hard cap was triggered which would have triggered it slightly earlier than desired.

This audit reflects the amended code; none of the vulnerabilities are present in the current versions. Nevertheless, the discussion here does not whitewash complaints or poor findings in the code.

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 upgradability are worth paying attention to.

If you do not trust the FunFair team, I do not recommend engaging with these contracts. They have broad rights, and will have broad rights for some time.

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 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 upgradability of the three tier system (Token -> Controller -> Ledger).

5.4.1 Reserve Tokens

The smart contract issues reserve tokens for a future sale or auction. The white paper specifies a calculation for how many reserve tokens are issued; the server will credit these tokens as part of the multimint process. These reserve tokens are issued to an address and as the contract is written, can be `transfer`ed.

It is our understanding that FunFair will lock these tokens into a smart contract implementing their future Dutch Auction. Until that happens, customers rely on FunFair to treat them per the white paper, and not use or transfer them. I would recommend this transfer happen quickly.

5.4.2 Upgradability

The FunFair Token contracts allow upgrading of the Controller and the Ledger. The desire to do this is almost certainly an acknowledgement that the FUN mechanics – and real implementation details about them – aren't sorted yet.

Rather than inconvenience users with a new FUN token address, these tokens allow the logic to be switched out at a later date in a single step. This is cool, and I like the idea.

Similarly the ledger is switchable; also a good idea.

However, this very upgradability gives rise to the chance for bad actions by the `owner` of the contracts. The contracts have a "finalization" step which can be used to lock the logic layers down, and the FunFair whitepaper tells readers to expect that the logic will be locked eventually.

Until that lock happens, the contracts are simply controlled by the owner. This seems like a sensible trade, and in fact, I expect some high profile token contract upgrades in the next six months as existing projects realize they cannot support all necessary business plans with their old tokens.

However, be warned. If you do not believe the FunFair team is trustworthy, you cannot (yet) trust the Token contracts.

5.5 ERC20 Contributions

A requirement for the Token Sale was to take tokens as part of the contribution phase.

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 FunFair 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 adds a data record which will be used when calling `multimint` at the time of token creation.

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' once the minting period is completed.

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 and far too much gas. 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. ERC223 is being discussed as I write this; it has its warts, but may be a good next step.

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 and Philip Hofer as part of the audit.

6.1 FunFairSale – 0x55c44fbad82686afb0ca41cefb8d086cb937b2e6

Generally, this contract is very simple and clearly safe on inspection. But. I really dislike a few of the decisions made as a matter of style and management. More complaints below.

6.1.1 Line 27: I like this

Hurrah for confirmable ownership changes. I approve.

6.1.2 Line 55: This is a bad name

Why is this called `setSoftCapDeadline`? It should be called just `setDeadline`. Also, the idea that the owner should be restricted from lengthening the deadline, but not shortening it seems wrongheaded to me. I can imagine reasons why an owner would wish to extend the deadline.

6.1.3 Line 68: Gas

I think if Line 68 read: `deadline = 0;` then the contract would require slightly *less* gas to complete, and the full design goals of constant gas throughout would be bested. As it is, the final contributor will pay slightly more gas than the standard contributor. I don't believe this is a problem in practice.

6.1.4 Line 73: Why Throw?

This seems wrong to me. The owner could have perfectly good reasons to withdraw funds during the crowdsale – they will be trusted with them at the end in any event. And, they can set the deadline to zero by calling `setSoftCapDealine(0)`.

So, this is harmless in most circumstances that don't involve racing an attacker, but I don't love it. In fact, the token reclamation is not locked this way, so this seems doubly questionable to me.

6.1.5 Line 80: Naming

This should be called `setTimes`. I agree with the idea that this shouldn't be called after the sale starts. It might be nice to have one call set all three items: the cap, the start and the deadline.

6.2 Token, Ledger and Controller

All of these are visible at any of the verified addresses, for instance:

- https://etherscan.io/address/0xbbb1bd2d741f05e144e6c4517676a15554fd4b8d

The architecture is Token -> Controller -> Ledger, each a separate upgradable contract. The FunFair whitepaper claims the Controller may be upgraded multiple times, the Ledger perhaps once. The Token contract is designed to not require upgrading.

6.2.1 Token Reclaiming

Sensibly, the Token contract allows reclamation of tokens sent to it. However the Controller and Ledger do not. I don't agree with this decision. Peterson's law is clear; tokens will end up in them.

That said, neither of the contracts will be easily accessible to even a moderately interested observer. The Token contract would need to be inspected, and addresses followed in order to determine the

direct addresses of the Controller and Ledger. I believe this restricts the likely senders of tokens to trolls.

Nevertheless, I bet both trolls and FunFair will regret not being able to reclaim wayward tokens from these two contracts and I recommend that when the next Controller or Ledger upgrade happens that these contracts become `TokenReceivable` as well.

6.2.2 Line 92: Eight decimals

Eight decimals is a sane number. I don't believe 18 decimals is helpful for most ERC20 tokens.

6.2.3 Line 136: Approvals and Zero

The contract will throw if an allowance attempt is made and the current allowance isn't zero. This is provably more correct than the vulnerable-to-races original version popularized by the ERC20 spec, but it will probably confuse some people. Overall I'd say leave it this way, but it's not great to `throw` with no way to tell a user what happened – a flaw in the Ethereum blockchain, not this code.

6.2.4 Lines 192, 196, 200 - Consistency

Why are `balanceOf` and `allowance` decorated `onlyToken` but `totalSupply` is not? It makes sense to me that you could argue one way or the other, but both ways seems wrong to me.

6.2.5 Line 318: LogMint

I'd kind of prefer to see a `Transfer` event here, based on existing industry support for token browsers.

6.2.6 Line 325: safeadd and multimint

Multiminting is rad; thanks to Griff and Jordi for the initial explanation. These contracts make what is, in my opinion, the right decision and do not use `safeAdd` for minting in order to save gas.

For all calls to `multimint` this is demonstrably the right decision, because at most there are 96 bits of value to be added per iteration, and `uint`'s overflow at 2^255 or so.

That said, the owner could overflow the `totalSupply` with a large argument to `mint` directly. This would be noticeable immediately, and is only possible during the minting period before `mintingStopped` is set true. I rate this as a negligible concern.

6.2.7 Line 341: (Heart)Burn

This gives me nausea. The only thing stopping someone from massively fucking up the `totalSupply` on the ledger is that `safeSub` will `throw` in line 342 before the `totalSupply` is decremented.

If `safeSub` didn't `throw` then the user could call `burn` with any `amount` and lower the `totalSupply`. I would just like a comment here that says "I, the developer, thought about this and confirmed it's okay."

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 soup-to-nuts services for companies and individuals engaging in the Token ecosystem from strategy to software to marketing. Please contact us at hello@newalchemy.io!

