FundYourselfNow Sale and Token Contract Audit

June 2017

Dennis Peterson

Discussion

FundYourselfNow requested an audit of their token sale. The code is available at:

https://github.com/PinnacleOne/FYNTokens/tree/master/contracts

I initially evaluated the github commit:

48885ebbc7d3d185e13e7f0ece0f78c381c0ab00

After changes I checked the modified version:

7ddf17ec33ec7adbb6a0f687011f6cb88d8d18f0

Token purchasers pay ether directly to a FundYourselfNow-owned copy of the Ethereum Foundation's standard multisig wallet, an admirable approach since the ether is deposited directly in a well-known, hardened contract, instead of storing contributed ether in a new contract. The wallet is extended primarily by inheriting from a small contract that implements the token sale.

The token contract is a fairly standard ERC20-compatible token. The tokens are tradeable once the sale ends.

The contracts specify at least Solidity 0.4.11, which is current as of this writing.

Disclaimer

The audit makes no statements 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.

Critical Errors

I identified no critical errors.

Moderate Issues

I was concerned about the sale starting immediately upon calling `tokenStart()`, since if initial demand were high, the transaction to actually start the sale could get stuck in traffic. The team resolved this without code changes by simply starting the sale a bit early, which worked fine.

Minor Issues

It could be a little tricky to get the sale to stop early, since it'll require a contribution that exactly reaches the token cap. With high traffic it could be hard to make that happen since the remaining supply will keep changing. But if that gets annoying you can always cut things off by just calling `stopTokenSwap()`.

Broad Concerns

Event Naming

A good practice is to name events with some clear prefix or suffix like "log" or "event." However, the FYN code starts regular functions with lowercase and events with uppercase, which is a decent and fairly common convention (including by ERC20 and the standard multisig wallet used in wallet.sol, so it makes sense to continue it here).

Maintaining State

Arguably the contracts maintain unnecessary state. In my review and some testing, I could find no errors in this, but the code would be more obviously error-free if it maintained fewer state variables.

For example, the modifier `areConditionsSatisfie`d sets `tokenSwa`p to false in several cases. Rather than setting `tokenSwap=false` and checking it in the modifier `isSwapStopped`, it could be more clear to have modifiers like `saleNotExpire`d and `capNotReached`, which directly check the conditions that, in the current code, set the boolean stored variable to `false`.

Related to this, Gavin Wood has advocated "condition-orientated programming," in which modifiers only check conditions, without making updates to state, while function bodies avoid checking conditions and only make state updates:

https://medium.com/@gavofyork/condition-orientated-programming-969f6ba0161a#.2fawrybt8

The FYN code doesn't follow this pattern, making state transitions directly in the `areTokensSatisified` modifier.

If modifiers don't update state, then they can be placed in any order. As it stands, it's important that `areConditionsSatisfied` be used last (as it is, in the current code).

However, there is a tradeoff in making these changes. Tokenswap's `areConditionsSatisfied` doesn't only change its own `tokenSwap` boolean. It also calls `tokenCtr.disableTokenSwapLock()`, activating the token. To remove the need to call `disableTokenSwapLock`, ICO-related constants and functions would need to be included in token.sol. So it may be better to keep the code as is.

Standard wallet

The standard multisig wallet is well-known, public, hardened code, and I see little risk in using it. I checked all the differences between the standard wallet and Wallet.sol.

The file SimpleWallet.sol appears to be a straight import of the standard wallet. I focused on Wallet.sol, which includes the additional crowdsale functionality.

Token Emergency Stops

The token allows a permanent stop of all transfers. If this is done, it is permanent; a new contract has to be created.

A less disruptive approach is to use an upgradeable token design, allowing the owner to stop, redeploy the implementation behind the public address, and restart. However, this gives the owner substantial power. It's arguably a good tradeoff to just have an irrevocable stop, thus requiring users and exchanges to migrate to a new token contract; this provides a social check against abuse. If the owner deploys a new contract with unfair balance adjustments, the community can migrate to a different contract instead.

In the initial version of the code, the stop was triggered by submitting a hash preimage. This was problematic because it was a single secret could not be changed. Now the stop is triggered by the multisig contract; since the multisig does not allow arbitrary calls to token, it was given an emergency stop function.

Sale Stops

The token sale can simply be stopped and restarted at will by the contract owner, but only within the allowed timeframe. Presumably this is also for emergency stop purposes, and in case of a stop, the owners would use standard multisig wallet functions to withdraw and distribute funds. (This functionality is also used to start the sale; see concerns in the "moderate issues" section.)

ERC20 attacks

The token code mitigates both the short address attack, and the race attack against approvals.

Time

The contracts use timestamps rather than blocknums to manage various deadlines. To a small extent, timestamps can be manipulated by miners. However, this is primarily a problem for fine-grained usage of timestamps, and isn't likely to be significantly exploitable when the times are measured in days.

Numerics

FYN makes good use of functions to check for safe arithmetic operations, and I found no overflow issues. These functions would be somewhat more convenient if they were written as, for example `safeAdd(x,y)` instead of `safeToAdd(x,y)`, combining the check and operation in one line. There's sometimes reason to avoid this, if you want to do something besides throw upon overflow, but these contracts just throw anyway.

Line By Line

Wallet.sol

Line 286: underLimit

Returns `false` if the value is zero. A comment notes that this is intentional, because they want multisig approval for zero-valued calls. I don't see a problem with this, just wondering about the rationale.

Line 331: tokenSwap naming

Consider changing the name `tokenSwap` to something that indicates whether `true` means the swap is active.

Line 368: areConditionsSatisfied naming

Most modifiers just check a condition, and prevent the function from running if the condition is false. It'd be nice if the name of this one indicated that it will actually update state; e.g. `endSaleIfConditionsSatisfied`.

Line 489: Minting protection

A line is added to the standard wallet contract which prevents calling arbitrary functions on the token contract, once the token contract address is known. This is an elegant way to prevent arbitrary minting after the sale starts, since the token contract's mint function is marked `onlyFromWallet`.

Token.sol

Lines 38-39: Public variables

If these were named `balanceOf` and `allowance` and made public, the constant accessor functions could be removed.

Line 65: Payload size protection

Kudos on including this. It would be slightly more convenient to multiply the parameter by 32 so each caller doesn't have to do it, but it's fine as is.

Line 167: Approve race protection

Kudos on this as well. As noted in the comment, this isn't perfect protection, but it's the best that can be done for the compliant ERC20 function.

It's worth considering adding noncompliant functions without the vulnerability, e.g. a function that adds to the current approved amount rather than replacing it. However, very few ERC20 contracts do this and users don't expect it.

Line 159: currentSwapRate

Does a stateless calculation every time, which is great.

