AdEx Contract Audit

June, 2017

Dennis Peterson

Contents

Introduction **3**

Files Audited **3**

Disclaimer **3**

Executive Summary **3**

General Discussion **4**

Moderate and Minor Issues **4**
- Gas cost attack 4
- Token deposits 4
- Ownership 5
- Prebuyer shenanigans 5

Line By Line Comments **5**
- 98: isNotHalted 5
- Line 104: PreBuy event lacks address 5
- Line 108: Constructor 5
- Lines 134-139: Initial balances 6
- Line 146: Transfer no-op 6
- Line 147: Throw transfer 7
- Line 190: immediate send on purchase 7
- Line 192-193: processPurchase 7
- Lines 195-196, 219: no safeMath 7
- Lines 214-215: preBuy calls and potential shenanigans 7
- Line 227: is_crowdfund_period 8
- Lines 236-254: grantVested 8

Appendix - Short Address Attack **8**

Contact **9**

Introduction

AdEx asked us to audit their smart contract code for their token and ICO. New Alchemy reviewed the system from a technical perspective looking for bugs in their codebase, and evaluated the security tradeoffs they made in the design.

Files Audited

The code is at the following github repository:

We evaluated the github hash:

51652fd206036fa6ced9286b9a30a2b472748f489a39b16076d88777f2608111

They import OpenZeppelin code with the github hash:

7b9c1429d918a3cf685a1e85fd497d9cc3cf350e

NewAlchemy audited a previous version of the OpenZeppelin code. For this audit we reviewed the current code, making sure new vulnerabilities have not been introduced, and paying close attention to the interaction of OpenZeppelin with new AdEx code.

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.

Executive Summary

We found no critical errors, and several minor issues. In general the code is good quality, showing awareness of common security pitfalls. The project uses the Truffle framework and includes unit tests in Javascript.

AdEx made no changes to Zeppelin code, which for the most part is a good decision. However, it's not without tradeoffs. Making the included version of the Zeppelin code work in this context required some trickiness, which is generally a bad idea in smart contracts. It's better if the code is so clear and simple that it obviously works. Some complexity could have been avoided by using more recent Zeppelin code.

There's some potential for attackers to inflict unnecessary gas cost on users. This is an inevitable consequence of the vesting feature.

General Discussion

The code uses Solidity version 0.4.11, which is the latest stable version at time of audit. The OpenZeppelin code is slightly older than Zeppelin's latest version.

The AdEx code uses SafeMath functions in most places.

The code uses a common convention of capitalizing event names. Generally we prefer making event names even more clear, with a prefix like "LOG". However, the ERC20 standard uses capitalization, and since this contract is an ERC20, it makes sense to continue the same pattern for its custom events.

A very recent discovery is that short address protection can cause transfers from hardware wallets to fail, due to parameter padding by the EVM. This is only a problem if the protection requires an exact length, instead of only failing when the data is too short. Fortunately, the Zeppelin code does the latter, so AdEx should be fine.

https://blog.coinfabrik.com/smart-contract-short-address-attack-mitigation-failure/

Moderate and Minor Issues

Gas cost attack

The AdEx token inherits from Zeppelin's VestedToken contract. Consequently, any user can send "grants," which transfer tokens that are only spendable over time according to their vesting schedule. This is an intended design feature for all users, but there is a downside.

With vested tokens, it's more complicated to determine a user's spendable balance; the balance function has to loop through all the grants the user has received, calculating how much remains unvested for each one.

The code allows every user to receive up to twenty vested transfers. Each of them increases the gas cost for every transfer that user does. An attacker who wishes to harass a user can call `grantVestedTokens()` with tiny amounts of ETH, and permanently increase the gas cost for every transfer by that address.

A way to mitigate this is to reduce the number `MAX_GRANTS_PER_ADDRESS` to a lower number than twenty, if it's likely that a lower number would be sufficient for users.

On the other hand, this issue may not cause much trouble in actual practice. A user who is attacked can simply transfer their entire balance to a new address, spending less gas than the attacker.

Token deposits

There is no way to prevent users from depositing ERC20 tokens to your contract's address, and a significant number of tokens have been stranded this way. It's always a good idea to add a function that allows you to withdraw those tokens. For example:

```
function withdrawERC20(address tokenaddr) onlyOwner {
    ERC20 token = ERC20(tokenaddr);
    uint bal = token.balance();
    token.transfer(msg.sender, bal);
}
```

Ownership

The contract does not provide for changing addresses of the owner, AdEx team, or the multisig which receives funds. This would be a serious concern if these addresses had long-term uses. However, they appear to be used only for the sale, so it's unlikely to be a problem as long as AdEx does a reasonably good job securing their keys.

Prebuyer shenanigans

It's possible for the first prebuyer to buy the entire prebuy allocation. Since three specific prebuyers will be specified, this may not be the agreement. We presume they are known entities, so if they misbehaved they could be dealt with the old-fashioned way.

Line By Line Comments

98: isNotHalted

Always a good idea for sales to implement emergency halts like this. This modifier is not applied to token transfers, it can only halt crowdsale contributions.

A slight improvement would be to use a different, high-security address for resuming the sale, in case the reason for the halt was a compromise of the owner address. Given that this is only relevant during the crowdsale, it's not a big deal.

Line 104: PreBuy event lacks address

Adding the address would probably be helpful, in the unlikely event of "shenanigans."

Line 108: Constructor

Having constructor arguments will cause some hassle in verifying the contract on Etherscan; a more convenient approach is for the constructor to just set `owner = msg.sender` and then use a separate initializer function with an `onlyOwner` modifier.

Lines 134-139: Initial balances

These four lines set the initial balances to four addresses, for a total of 100,000,000 tokens as designed. The rest of the contract only transfers existing tokens, rather than creating new ones.

The owner gets both `ALLOC_TEAM` and `ALLOC_CROWDSALE`. The team allocation gets transferred to the team, the crowdsale allocation goes to crowdsale purchasers and whatever remains goes to the fund.

Line 146: Transfer no-op

In the version of OpenZeppelin used by AdEx, the transfer function checks for the correct length of `msg.data`, in order to prevent the "short address attack" (see appendix). This is good protection, but there is a problem: if a contract calls `transfer()` from another function, which has a different number of parameters, the length check will throw an exception. Since Zeppelin's code is intended for use in child contracts, they ended up removing the length check from their latest code.

AdEx calls `transfer()` from the function `grantVestedTokens()`, which is called from two places. One is `grantVested()`, which happens to have two parameters. But another is `preBuy()`, which has no parameters. At first glance it seems the latter should fail.

To prevent this, AdEx added line 146:

```
if (\_to == msg.sender) return;
```

When `preBuy()` is called, the "to" address is the sender, so the transfer fails without throwing an exception. Normally this would mean the recipient gets no tokens, except that `preBuy()` also calls `processPurchase()`, which does its own transfer.

From one perspective this is a neat hack to avoid having to change Zeppelin code, while still keeping the length check. But it's convoluted and fragile. It disables functionality in a certain circumstance, while sometimes in that circumstance the functionality is needed, so in those cases it's added back with additional code. From a software engineering perspective this is frightening, and would likely eventually break any code that was subject to change. We don't recommend using this method in other contracts. However, deployed contracts are not subject to change, and this contract does appear to work; I checked it repeatedly since I kept getting the shivers, but could not find a scenario that was incorrect.

The scenarios are as follows:

Public buy: happens in fallback function, during the sale. Calls `processPurchase` which does its own balance adjustment, without calling `transfer`.

Presale buy: happens before public sale starts. Calls `processPurchase` to transfer tokens to the buyer. Then calls `grantVestedTokens`, which will call `transfer` with an incorrect parameter length, but with `msg.sender == \_to` so the transaction does not throw.

Team/fund distribution: after sale is complete, owner calls `grantVested` which calls `grantVestedTokens` twice. The `msg.sender` is not the recipient, so the `msg.sender` check is no protection. But this function happens to have two parameters so the length check passes.

For future contracts, a cleaner approach might be to use the latest version of Zeppelin, and instead of adding line 146, add if desired, add back the `msg.sender.length` check (keeping in mind the potential issue with multisig wallets).

Line 147: Throw transfer

This line causes any token transfer to fail if the crowdsale has not completed. This is good, but would be even better if transfers were enabled manually after review of everyone's balances, just in case some undiscovered bug causes incorrect balances. However, since there's no manual minting, it's not as critical as in some other sales.

Note the wrinkle here: `transfer` actually does have to be called before the sale ends! (See the "presale buy" scenario above.) But line 146 keeps things running, returning from the function without doing anything before line 147 kills the transaction.

Line 190: immediate send on purchase

When ETH is deposited, it is immediately sent to the multisig address. All contributions will fail if the multisig has an expensive fallback; no reputable multisig has an expensive fallback, so this should be fine. (If it were not fine, failing immediately is by far the best outcome anyway, given that the multisig address cannot be changed.)

Line 192-193: processPurchase

This is where `processPurchase` does its own transfer of token balances, so it doesn't have to call the inherited transfer function.

Lines 195-196, 219: no safeMath

The few places where safe math functions are not used. No chance of overflow, since the quantities are limited by ether deposit amounts.

Lines 214-215: preBuy calls and potential shenanigans

Here we see the call to `processPurchase`, followed by `grantVestedTokens`, as described above. Note that the rate is higher in the first call, resulting in more tokens. We presume this is intentional, so these buyers have some tokens which vest, and others available immediately.

In the first call, the second parameter is the remaining funds available to prebuyers overall; this prevents prebuyers from extracting more funds than they're entitled to from the general public. However, there's nothing to prevent the first prebuyer who calls from taking the entire allocation.

Line 227: is_crowdfund_period

This modifier on the fallback checks whether `etherRaised >= hardcapInEth`. It will throw if the ETH hardcap has been exceeded, but will not prevent the cap from being exceeded by the last transaction. This isn't necessarily bad, just something to be aware of.

The modifier also checks whether `ADXSold >= ALLOC_CROWDSALE`. It doesn't appear that the sold amount could actually exceed the allocated amount, so just equal would also work, but there's no harm leaving it as is.

Lines 236-254: grantVested

This is the scenario which would fail, if it did not have two parameters.

One of those parameters is `\_adexTeamAddress`, which presumably should be the same as the variable `adexTeamAddress` set in the constructor; apparently the parameter is there just to avoid trouble with the transfer. There's nothing to make sure these addresses are the same, but if the owner chooses to pass in a different address it's only an internal matter for AdEx. Be *very careful* to pass in the correct addresses! With the constructor, the contract can be abandoned if there's a mistake; with this function there's no recourse.

The fund address is not set in the constructor, and is only passed in here.

This function would be less stressful for the owner if both addresses were set in the constructor, carefully checked by multiple parties before the point of no return, then checked for accuracy with a bit of extra code in `grantVested`; keep the parameters but throw if they don't match what was set previously.

A simpler bit of protection would be to at least check that neither address is `0x0`.

Appendix - 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 at least 68:

```
if (msg.data.length < 68) throw;
```

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:

```
modifier onlyPayloadSize(uint numwords) {
    if (msg.data.length < numwords * 32 + 4) throw;
    _;
}

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. As mentioned above, it's important not to require that the length be *exactly* equal, since the EVM can pad parameters.

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

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!