# AdEX Smart Contracts Audit

Zero Knowledge Labs Auditing Services

AUTHOR: MATTHEW DI FERRANTE

2017-11-13

## Audited Material Summary

The audit consists of the `ADXExchange` and `ADXRegistry` contracts. The git commit hash of the reviewed files is `6ecc86b`.

### High Level Description

The contracts are fairly clean and build on the OpenZeppelin standard contracts. The `ADXExchange` implements the main functionality, which is an on-chain market for advertisement bids which leverages IPFS to store additional information.

### Security

The contracts are well constructed and the audit has not uncovered any major security flaws. The code is kept concise and simple.

### ADXExchange.sol

The `ADXExchange` contract is a self contained contract that only inherits from `Ownable` and `Drainable`:

```
contract ADXExchange is Ownable, Drainable
```

Besides the constructor, the non-constant functions are `placeBid`, `cancelBid`, `acceptBid`, `giveupBid`, `verifyBid`, `claimBidReward`, and `refundBid`.

### Constructor

```
    function ADXExchange(address _token, address _registry)
    {
        token = ERC20(_token);
        registry = ADXRegistry(_registry);
    }
```

The constructor only assigns the token and registry arguments to storage, typecasting token into an `ERC20` interface and typecasting registry into an `ADXRegistry` interface.

### placeBid

```
    function placeBid(uint _adunitId, uint _target, uint _rewardAmount,
       uint _timeout, bytes32 _peer)
        onlyRegisteredAcc
    {
        bytes32 adIpfs;
        address advertiser;
        address advertiserWallet;

        // NOTE: those will throw if the ad or respectively the account do
            not exist
        (advertiser,adIpfs,,) = registry.getItem(ADUNIT, _adunitId);
        (advertiserWallet,,,) = registry.getAccount(advertiser);

        // XXX: maybe it could be a feature to allow advertisers bidding
           on other advertisers' ad units, but it will complicate things
           ...
        require(advertiser == msg.sender);
```

```
14
15          Bid memory bid;
16
17          bid.id = ++bidsCount; // start from 1, so that 0 is not a valid ID
18          bid.state = BidState.Open; // XXX redundant, but done for code
                clarity
19
20          bid.amount = _rewardAmount;
21
22          bid.advertiser = advertiser;
23          bid.advertiserWallet = advertiserWallet;
24
25          bid.adUnit = _adunitId;
26          bid.adUnitIpfs = adIpfs;
27
28          bid.requiredPoints = _target;
29          bid.requiredExecTime = _timeout;
30
31          bid.advertiserPeer = _peer;
32
33          bidsById[bid.id] = bid;
34          bidsByAdunit[_adunitId].push(bid.id);
35
36          token.transferFrom(advertiserWallet, address(this), _rewardAmount)
                ;
37
38          LogBidOpened(bid.id, advertiser, _adunitId, adIpfs, _target,
                _rewardAmount, _timeout, _peer);
39      }
```

The `placeBid` function is a state-heavy function that takes a bid from an advertiser, and places it in the bids queue if the account is registered and the advertiser has enough tokens. On success the `LogBidOpened` event is fired.

The infromation for the ad unit and advertiser is loaded through the registry.

### cancelBid

```
1       function cancelBid(uint _bidId)
2           onlyRegisteredAcc
3           onlyExistingBid(_bidId)
4           onlyBidOwner(_bidId)
```

```
        onlyBidState(_bidId, BidState.Open)
    {
        var bid = bidsById[_bidId];
        bid.state = BidState.Canceled;
        token.transfer(bid.advertiserWallet, bid.amount);

        LogBidCanceled(bid.id);
    }
```

The `cancelBid` function allows an advertiser to cancel a bid given a bid id, as long as they are calling from an account that is registered and is the owner of the bid, and the bid exists and is in the `Open` state.

On success the tokens are refunded to the advertiser wallet and a `LogBidCanceled` event is emitted.

#### acceptBid

```
    function acceptBid(uint _bidId, uint _slotId, bytes32 _peer)
        onlyRegisteredAcc
        onlyExistingBid(_bidId)
        onlyBidState(_bidId, BidState.Open)
    {
        address publisher;
        address publisherWallet;
        bytes32 adSlotIpfs;

        // NOTE: those will throw if the ad slot or respectively the
           account do not exist
        (publisher,adSlotIpfs,,) = registry.getItem(ADSLOT, _slotId);
        (publisherWallet,,,) = registry.getAccount(publisher);

        require(publisher == msg.sender);

        var bid = bidsById[_bidId];

        // should not happen when bid.state is BidState.Open, but just in
           case
        require(bid.publisher == 0);

        bid.state = BidState.Accepted;

        bid.publisher = publisher;
```

```
24          bid.publisherWallet = publisherWallet;
25
26          bid.adSlot = _slotId;
27          bid.adSlotIpfs = adSlotIpfs;
28
29          bid.publisherPeer = _peer;
30
31          bid.acceptedTime = now;
32
33          bidsByAdslot[_slotId].push(_bidId);
34
35          LogBidAccepted(bid.id, publisher, _slotId, adSlotIpfs, bid.
                acceptedTime, bid.publisherPeer);
36      }
```

The `acceptBid` function takes an existing bid in the `Open` state and, if the calling account is registered and the caller is the publisher for the ad slot, acceps the bid and emits a `LogBidAccepted` event.

### giveupBid

```
1       function giveupBid(uint _bidId)
2           onlyRegisteredAcc
3           onlyExistingBid(_bidId)
4           onlyBidAceptee(_bidId)
5           onlyBidState(_bidId, BidState.Accepted)
6       {
7           var bid = bidsById[_bidId];
8           bid.state = BidState.Canceled;
9           token.transfer(bid.advertiserWallet, bid.amount);
10
11          LogBidCanceled(bid.id);
12      }
```

The `giveupBid` function allows an acceptee of an existing bid to cancel their acceptance, setting the bid state to `Canceled`, refunding tokens, and emitting a `LogBidCanceled` event on success.

### verifyBid

```
1       function verifyBid(uint _bidId, bytes32 _report)
2           onlyRegisteredAcc
```

```
        onlyExistingBid(_bidId)
        onlyBidState(_bidId, BidState.Accepted)
    {
        var bid = bidsById[_bidId];

        require(bid.publisher == msg.sender || bid.advertiser == msg.
            sender);

        if (bid.publisher == msg.sender) {
            bid.confirmedByPublisher = true;
            bid.publisherReportIpfs = _report;
        }

        if (bid.advertiser == msg.sender) {
            bid.confirmedByAdvertiser = true;
            bid.advertiserReportIpfs = _report;
        }

        if (bid.confirmedByAdvertiser && bid.confirmedByPublisher) {
            bid.state = BidState.Completed;
            LogBidCompleted(bid.id, bid.advertiserReportIpfs, bid.
                publisherReportIpfs);
        }
    }
```

The `verifyBid` function takes a bid currently in the `Accepted` state and:

- if called by the bid publisher, sets the publisher's confirmation and IPFS report
- if called by the bid advertiser, sets the advertiser's confirmation and IPFS report

If at any point it has been called by both publisher and advertiser, the bid state changes to `Completed` and the contract emits a `LogBidCompleted` event.

### claimBidReward

```
    function claimBidReward(uint _bidId)
        onlyRegisteredAcc
        onlyExistingBid(_bidId)
        onlyBidAceptee(_bidId)
        onlyBidState(_bidId, BidState.Completed)
    {
        var bid = bidsById[_bidId];
```

```
8
9           bid.state = BidState.Claimed;
10
11          token.transfer(bid.publisherWallet, bid.amount);
12
13          LogBidRewardClaimed(bid.id, bid.publisherWallet, bid.amount);
14      }
```

The `claimBidReward` function takes an existing bid in the `Completed` state and if the caller is the publisher, the state of the bid is changed to `Claimed` and the bid's tokens are transferred to the publisher's wallet, and the contract emits a `LogBidRewardClaimed` event on success.

### refundBid

```
1       function refundBid(uint _bidId)
2           onlyRegisteredAcc
3           onlyExistingBid(_bidId)
4           onlyBidOwner(_bidId)
5           onlyBidState(_bidId, BidState.Accepted)
6       {
7           var bid = bidsById[_bidId];
8           require(bid.requiredExecTime > 0); // you can't refund if you
                haven't set a timeout
9           require(SafeMath.add(bid.acceptedTime, bid.requiredExecTime) < now
                );
10
11          bid.state = BidState.Expired;
12          token.transfer(bid.advertiserWallet, bid.amount);
13
14          LogBidExpired(bid.id);
15      }
```

The `refundBid` function takes an existing bid in the `Accepted` state and if the bid owner is the caller, it allows them to receive a refund on their bid if they've set a timeout, and the timeout has lapsed. The bid is set to the `Expired` state, the tokens are refunded, and a `LogBidExpired` event is emitted.

### Constant / user functions

All `ADXExchange` functions beyond this point are constant functions not used within the smart contract and therefore pose no security risks.

### getBidsFromArr

```
    function getBidsFromArr(uint[] arr, uint _state)
        internal
        returns (uint[] _all)
    {
        BidState state = BidState(_state);

        // separate array is needed because of solidity stupidity (pun
           intended ))) )
        uint[] memory all = new uint[](arr.length);

        uint count = 0;
        uint i;

        for (i = 0; i < arr.length; i++) {
            var id = arr[i];
            var bid = bidsById[id];
            if (bid.state == state) {
                all[count] = id;
                count += 1;
            }
        }

        _all = new uint[](count);
        for (i = 0; i < count; i++) _all[i] = all[i];
    }
```

### getAllBidsByAdunit

```
    function getAllBidsByAdunit(uint _adunitId)
        constant
        external
        returns (uint[])
    {
        return bidsByAdunit[_adunitId];
    }
```

### getBidsByAdunit

```
    function getBidsByAdunit(uint _adunitId, uint _state)
        constant
        external
        returns (uint[])
    {
        return getBidsFromArr(bidsByAdunit[_adunitId], _state);
    }
```

### getAllBidsByAdslot

```
    function getAllBidsByAdslot(uint _adslotId)
        constant
        external
        returns (uint[])
    {
        return bidsByAdslot[_adslotId];
    }
```

### getBidsByAdslot

```
    function getBidsByAdslot(uint _adslotId, uint _state)
        constant
        external
        returns (uint[])
    {
        return getBidsFromArr(bidsByAdslot[_adslotId], _state);
    }
```

### getBid

```
    function getBid(uint _bidId)
        onlyExistingBid(_bidId)
        constant
        external
        returns (
            uint, uint, uint, uint, uint,
            // advertiser (ad unit, ipfs, peer)
```

```
            uint, bytes32, bytes32,
            // publisher (ad slot, ipfs, peer)
            uint, bytes32, bytes32
        )
    {
        var bid = bidsById[_bidId];
        return (
            uint(bid.state), bid.requiredPoints, bid.requiredExecTime, bid
                .amount, bid.acceptedTime,
            bid.adUnit, bid.adUnitIpfs, bid.advertiserPeer,
            bid.adSlot, bid.adSlotIpfs, bid.publisherPeer
        );
    }
```

### getBidReports

```
    function getBidReports(uint _bidId)
        onlyExistingBid(_bidId)
        constant
        external
        returns (
            bytes32, // advertiser report
            bytes32 // publisher report
        )
    {
        var bid = bidsById[_bidId];
        return (bid.advertiserReportIpfs, bid.publisherReportIpfs);
    }
```

### ADXRegistry.sol

The `ADXRegistry` contract stores advertiser and ad items information. It is used heavily by the `ADXExchange` contract.

The contract has no constructor and only directly inherit from the `Ownable` and `Drainable` contracts.

Only two non-constant functions are implemented in this contract - `register` and `registerItem`:

### register

```
    function register(bytes32 _name, address _wallet, bytes32 _ipfs,
        bytes32 _sig, bytes32 _meta)
        external
    {
        require(_wallet != 0);
        // XXX should we ensure _sig is not 0? if so, also add test

        var isNew = accounts[msg.sender].addr == 0;

        var acc = accounts[msg.sender];

        if (!isNew) require(acc.signature == _sig);
        else acc.signature = _sig;

        acc.addr = msg.sender;
        acc.wallet = _wallet;
        acc.ipfs = _ipfs;
        acc.name = _name;
        acc.meta = _meta;

        if (isNew) LogAccountRegistered(acc.addr, acc.wallet, acc.ipfs,
            acc.name, acc.meta, acc.signature);
        else LogAccountModified(acc.addr, acc.wallet, acc.ipfs, acc.name,
            acc.meta, acc.signature);
    }
```

The `register` function either:

- registers a new account if it has not been registered before, and emits a `LogAccountRegistered` event on success
- updates an already existing account, as long as a valid signature is given, and emits a `LogAccountModified` event on success

### registerItem

```
    function registerItem(uint _type, uint _id, bytes32 _ipfs, bytes32
        _name, bytes32 _meta)
        onlyRegistered
    {
        // XXX _type sanity check?
        var item = items[_type][_id];
```

```
6
7           if (_id != 0)
8               require(item.owner == msg.sender);
9           else {
10              // XXX: what about overflow here?
11              var newId = ++counts[_type];
12
13              item = items[_type][newId];
14              item.id = newId;
15              item.itemType = ItemType(_type);
16              item.owner = msg.sender;
17
18              accounts[msg.sender].items[_type].push(item.id);
19          }
20
21          item.name = _name;
22          item.meta = _meta;
23          item.ipfs = _ipfs;
24
25          if (_id == 0) LogItemRegistered(
26              item.owner, uint(item.itemType), item.id, item.ipfs, item.name
                    , item.meta
27          );
28          else LogItemModified(
29              item.owner, uint(item.itemType), item.id, item.ipfs, item.name
                    , item.meta
30          );
31      }
```

The `registerItem` function works much like the `register` function, except it works on items instead of accounts:

- If the item does not exist yet, it adds it to the map and emits a `LogItemRegistered` event
- If the item already exists, it checks that the caller is the item owner, updates the item's information and emits a `LogItemModified` event

It can only be called by registered accounts.

### isRegistered

```
1       function isRegistered(address who)
2           public
```

```
        constant
        returns (bool)
    {
        var acc = accounts[who];
        return acc.addr != 0;
    }
```

The `isRegistered` function returns true if the `who` address is in the `accounts` map and that account has a non-zero `addr` set, and false otherwise.

### getAccount

```
    function getAccount(address _acc)
        constant
        public
        returns (address, bytes32, bytes32, bytes32)
    {
        var acc = accounts[_acc];
        require(acc.addr != 0);
        return (acc.wallet, acc.ipfs, acc.name, acc.meta);
    }
```

The `getAccount` function returns a tuple consisting of a wallet address, an IPFS hash, a name and a meta tag if the account exists, and throws otherwise.

### getAccountItems

```
    function getAccountItems(address _acc, uint _type)
        constant
        public
        returns (uint[])
    {
        var acc = accounts[_acc];
        require(acc.addr != 0);
        return acc.items[_type];
    }
```

The `getAccountItems` function returns the number of items for an account, or throws if the account is not registered.

### getItem

```
    function getItem(uint _type, uint _id)
        constant
        public
        returns (address, bytes32, bytes32, bytes32)
    {
        var item = items[_type][_id];
        require(item.id != 0);
        return (item.owner, item.ipfs, item.name, item.meta);
    }
```

The `getItem` getter returns a tuple consisting of an address, an IPFS hash, a name, and a meta tag if the id/type combination exists in the map, and throws otherwise.

### hasItem

```
    function hasItem(uint _type, uint _id)
        constant
        public
        returns (bool)
    {
        var item = items[_type][_id];
        return item.id != 0;
    }
```

The `hasItem` function is a simple checker that returns true if the `items` map has an item of type `_type` and id `_id`.

## Disclaimer

This audit concerns only the correctness of the Smart Contracts listed, and is not to be taken as an endorsement of the platform, team, or company.

## Audit Attestation

This audit has been signed by the key provided on https://keybase.io/mattdf - and the signature is available on https://github.com/mattdf/audits/