Parity Second Price Auction (Polkadot Sale) Audit

AUTHOR: MATTHEW DI FERRANTE

2017-10-02

Audited Material Summary

The audit consists of the SecondPriceAuction.sol, FrozenToken.sol and Multicertifier.sol contracts. The git commit hashof the reviewed files is `69b125ca4c6854f47b8f65730ddc1142074e0d71`.

The sale logic consists of a Frozen token with only the ERC20 "transfer" function implemented in the interface, and which only "certified" participants can buy into using the buyin function, feeding an ECDSA signature as one of the arguments. The crowdsale participants must sign a STATEMENT HASH, which consists of the phrase "Ethereum Signed Message: Please take my Ether and try to build Polkadot.". This is to ensure the participants attest to understanding what the sale is for, and that the funds are to be used for this condition.

Other restrictions include only being able to buy from normal accounts (a contract cannot buy tokens), a limit on gasprice and a minimum purchase limit to prevent dust payments. The certification process for participants consists of KYC/AML checks through Parity's ICO Passport Service.

The Sale is an auction type sale, for which the token price is *time dependent* (tokens get cheaper the longer the sale runs), and sale duration is *allocation dependent* (the more tokens that are bought the sooner the sale's deadline).

The `calculateEndTime` and `currentPrice` function breakdowns show a plot of the curves for each respective mechanism. The functions are implemented the way they are (with magic numbers) as the EVM does not have logarithm opcodes.

Overall, the contracts are well constructed and show good design, and I have found no flaws in them, security or otherwise.

SecondPriceAuction.sol

The SecondPriceAuction contract implement the sale logic and is what participants will directly interact with. It interacts with the FrozenToken only through the transfer function.

It has no inheritances and is self contained:

```
contract SecondPriceAuction
```

The contract is well constructed and has no security issues.

Constructor

```
function SecondPriceAuction(
    address _certifierContract,
    address _tokenContract,
    address _treasury,
    address _admin,
    uint _beginTime,
    uint _tokenCap
) public {
        certifier = Certifier(_certifierContract);
        tokenContract = Token(_tokenContract);
        treasury = _treasury;
        admin = _admin;
        beginTime = _beginTime;
        tokenCap = _tokenCap;
        endTime = beginTime + 15 days;
}
```

The constructor only assigns the arguments to their respective state variables, with the extra variable being endTime, which is set to 15 days beyond the sale's start time.

SecondPriceAuction Public Functions

Default Function

```
function() public { assert(false); }
}
```

The default function rejects any calls made to the contract, to prevent payments from participants that have not signed the crowdsale statement.

buyin

```
function buyin(uint8 v, bytes32 r, bytes32 s)
        public
        payable
        when_not_halted
        when_active
        only_eligable(msg.sender, v, r, s)
{
        flushEra();

        uint accounted;
        bool refund;
        uint price;
        (accounted, refund, price) = theDeal(msg.value);

        /// No refunds allowed.
        require (!refund);

        // record the acceptance.
        buyins[msg.sender].accounted += uint128(accounted);
        buyins[msg.sender].received += uint128(msg.value);
        totalAccounted += accounted;
        totalReceived += msg.value;
        endTime = calculateEndTime();
        Buyin(msg.sender, accounted, msg.value, price);

        // send to treasury
        treasury.transfer(msg.value);
}
```

The `buyin` function implements the main logic for the sale. This function, through modifiers, prevents calls if the sale is halted, or not active, and only allows purchasing if the participant is:

- Certified (in the list of allowed participants)
- Has signed the crowdsale statement and fed in the signature
- Not a contract account
- Has sent more than 5 finney with the transactions's gasprice being less than 5 Gwei

If all these checks pass, the function updates the crowdsale "ERA" (new era ticks every 5 minutes), ensures that the buyer is not trying to buy more tokens than available, adds the tokens to the buyer's buyins balance based on `theDeal`'s result and transfers the Ether to the treasury address.

As a side effect the endTime is updated according to `calculateEndTime`'s logic (see below), and the Buyin event is emitted.

Arithmetic Security

The function only ever adds `msg.value` values and uints returned by `theDeal`, none of which would ever be above 10ˆ30 respectively. The total amount of Ether right now is at most 10ˆ21, and `theDeal` also only adds and multiplies by constant factor, see below.

inject

```
function inject(address _who, uint128 _received)
    public
    only_admin
    only_basic(_who)
    before_beginning
{
        uint128 bonus = _received * uint128(BONUS_SIZE) / 100;
        uint128 accounted = _received + bonus;

        buyins[_who].accounted += accounted;
        buyins[_who].received += _received;
        totalAccounted += accounted;
        totalReceived += _received;
        endTime = calculateEndTime();
        Injected(_who, accounted, _received);
}
```

The `inject` function's purpose is to allow the admin to allocate tokens without depositing Ether. It can only be called by the sale admin before the sale's public start and the tokens can only be allocated to non-contract addresses.

The accounting logic skips `toDeal` and applies a direct bonus of 15%. The inability to call this after the public sale begins prevents price curve inconsistencies from the direct modifications to the `buyins` map.

Arithmetic Security

Similar to `theDeal`, with the exception of multiplying by BONUS_SIZE (15). The values would never go near overflow, as someone would need to send beyond 10^70 Ether, which is more than all Ether than exists by a factor of 10^44.

finalise

```
    function finalise(address _who)
        public
        when_not_halted
        when_ended
        only_buyins(_who)
    {
        // end the auction if we’re the first one to finalise.
        if (endPrice == 0) {
            endPrice = totalAccounted / tokenCap;
            Ended(endPrice);
        }

        // enact the purchase.
        uint total = buyins[_who].accounted;
        uint tokens = total / endPrice;
        totalFinalised += total;
        delete buyins[_who];
        require (tokenContract.transfer(_who, tokens));

        Finalised(_who, tokens);

        if (totalFinalised == totalAccounted) {
            Retired();
        }
    }
}
```

The `finalise` function finalises a buyin for an address. It can only be called if:

- The crowdsale isn’t halted
- The current time is past the crowdsale’s end time
- The address to be finalized has participated previously either directly through `buyin` or indirectly through `inject`’s token allocation.

If the caller is the first to finalise, they also set the endPrice as a side effect.

The amount of tokens transferred from SecondPriceAuction’s contract to the participant is calculated by dividing the participant’s accounted buyin by the endPrice. The participant’s entry in the buyin structure is then deleted to ensure allocation only happens once.

A `Finalised` event is emitted, and if this `finalise` call allocates tokens to the last remaining participant, a `Retired` event is also emitted to signify the end of this contract’s use.

tokenContract.transfer must succeed for this function to execute successfully.

Arithmetic Security

The only way the math in this function can cause an edge case is if `totalAccounted` is less than 10 million (assuming tokenCap is 10 million) - then `endPrice` would be 0. For that to be the case it would mean that less than a fraction of a cent was sent to the crowdsale in total, which is regardless prevented by the dust check modifiers.

setHalted

```
function setHalted(bool _halted) public only_admin { halted = _halted; }
```

The `setHalted` function is only callable by the admin and halts the crowdsale.

This is an emergency function.

drain

```
function drain() public only_admin { reasury.transfer(this.balance); }
```

The `drain` function sends the Ether collected thus far into the treasury. Can only be called by the crowdsale admin. The function will throw if the transfer fails.

This is an emergency function.

calculateEndTime

```
function calculateEndTime() public constant returns (uint) {
    var factor = tokenCap / DIVISOR * USDWEI;
    return beginTime + 18432000 * factor / (totalAccounted + 5 * factor) -
        5760;
}
```

This is a constant function that calculates when the crowdsale should end given the current amount of tokens sold.

The calculation is meant to create a smooth price curve. The figure below shows the price curve with `totalAccounted` as the `x` axis and the sale's end time (in seconds) as the `y` axis.

With the example variables, the delta is about 1 month. The more tokens are bought the sooner the sale will end.

Arithmetic Security

This function cannot overflow as neither `factor` nor `totalAccounted` would ever get high enough. From algebra you can calculate that in a certain base, two numbers multiplied together cannot ever make a number that is longer than the combined length of the two numbers. e.g the bit size of the two numbers combined must be over 2^256 to cause an overflow.

Due to `factor` being constantly calculated assuming `tokenCap` at 10 million and USDWEI at 300*10^18 at time of writing, factor is far less than 2^100. Similarly `totalAccounted` can never be even close to that, due to the fact that it would require more than 1 trillion Ether to have entered the contract.

Additionally, even if all Ether in existence is sent to the contract, the restult should never underflow or divide by 0, as even in that case the result of `(18432000 * factor)/ (totalAccounted + 5 * factor)` is still ~54480.

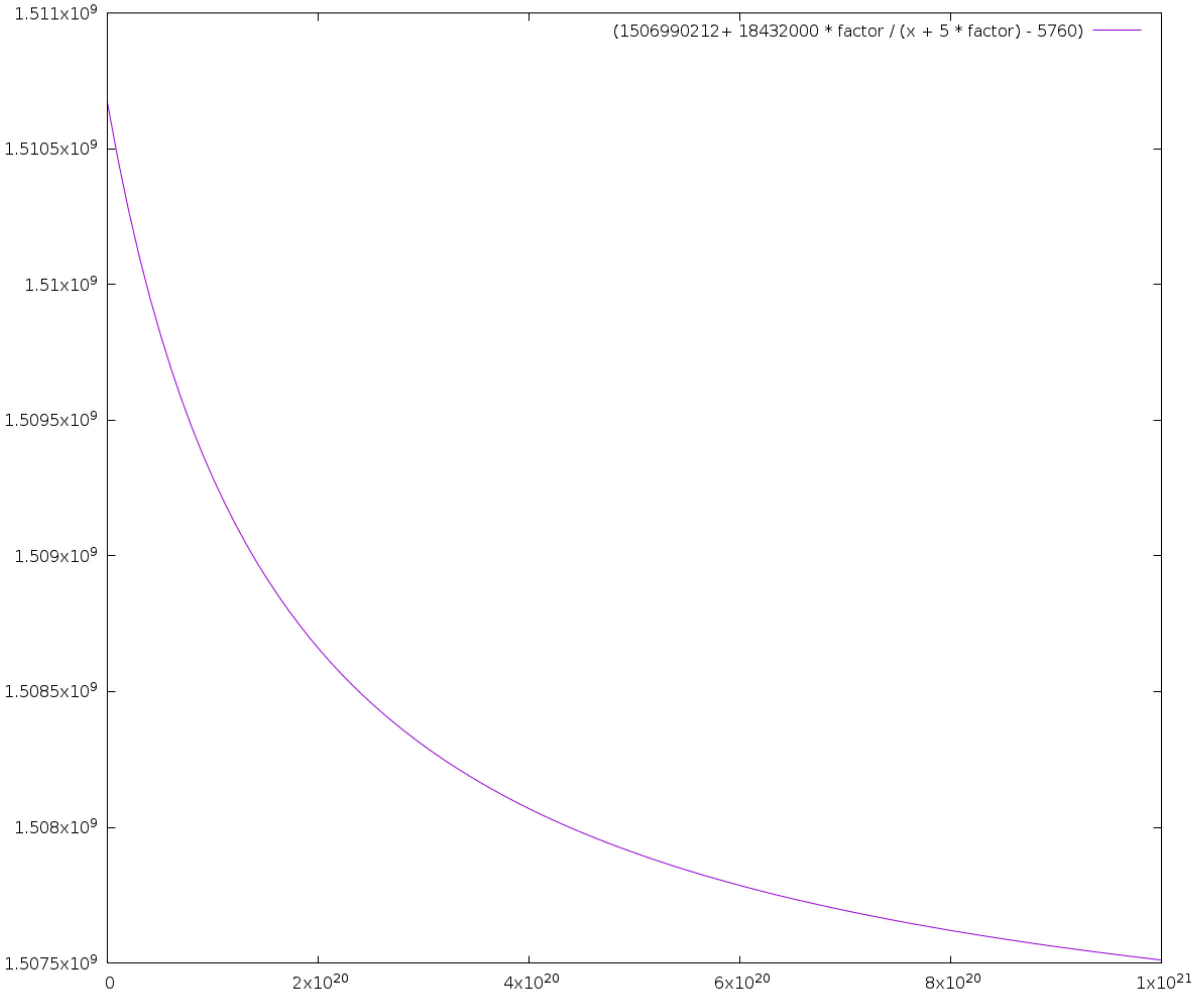

Figure 1: *time-delta*

currentPrice

```
function currentPrice() public constant returns (uint
    weiPerIndivisibleTokenPart) {
    return (USDWEI * 18432000 / (now - beginTime + 5760) - USDWEI * 5) /
        DIVISOR;
}
```

This is a constant function that calculates what the price should be for the next token purchase (or "current" price). Like calculateEndTime, it is meant to be a smooth curve with a sharp dropoff at the beginning. The further from the beginning of the sale, the cheaper the tokens are. Note is only time dependent, and does not care about the number of tokens bought (where as calculateEndTime is allocation dependent).

The figure below illustrates the fact that within the first ~6 hours from the crowdsale (a 20000 second step), the price drops by a factor of ~5.

Arithmetic Security

The only dynamic value in this equation is now, and since that is UNIX time, it will not go over 2ˆ32. USDWEI is many orders of magnitude larger, hence there is no risk of underflows or 0 results.

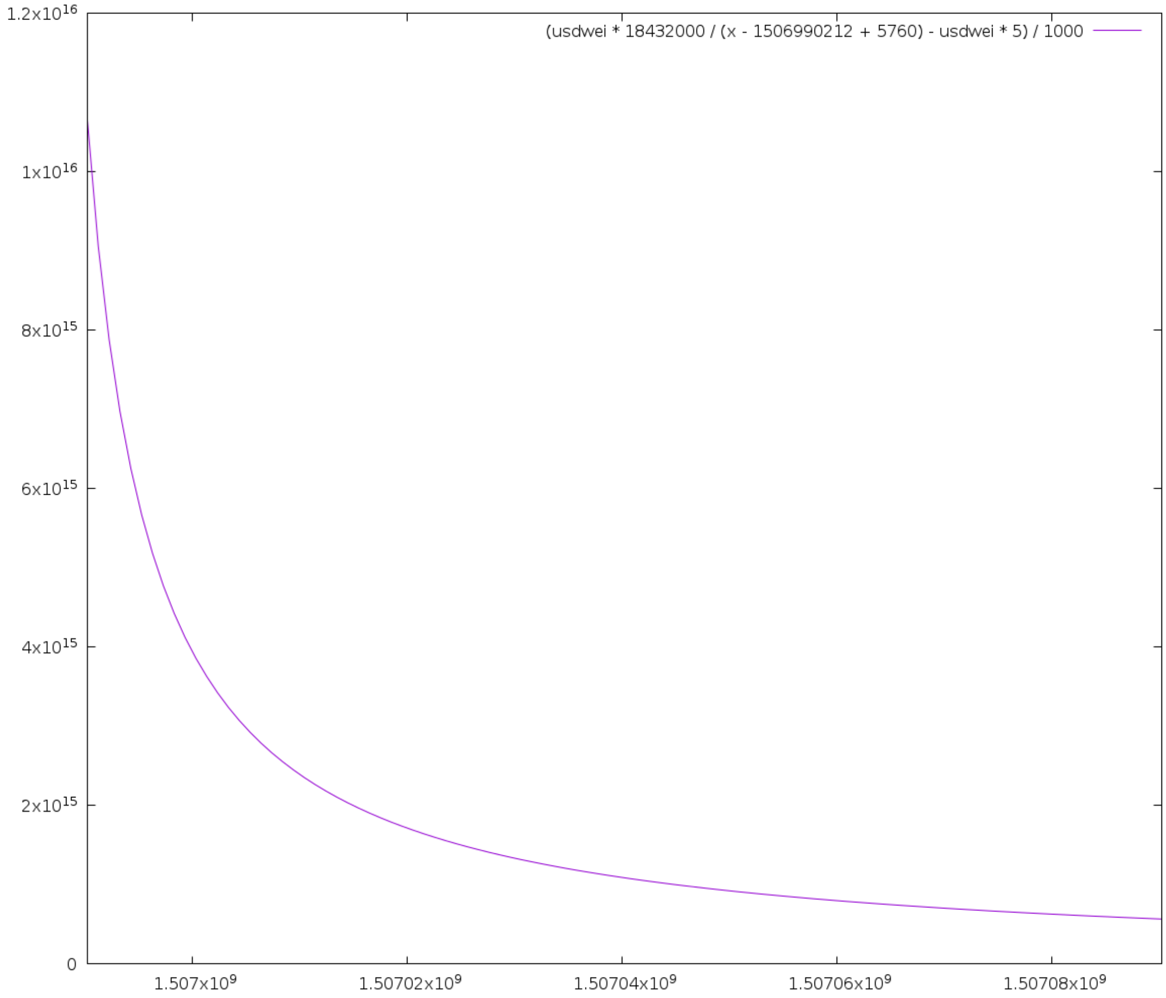

Figure 2: *price-delta*

tokensAvailable

```
function tokensAvailable() public constant returns (uint tokens) {
    return tokenCap - totalAccounted / currentPrice();
}
```

The `tokensAvailable` function returns the number of tokens left for allocation by taking the amount accounted for divided by the current price.

Arithmetic Security

Given that we know `currentPrice` won't ever won't return less than ~10ˆ15 as shown by the curve, this function is safe. The crowdsale would end or run out of tokens before the value can underflow. It's worth noting a slight edge case, where since tokenCap represents whole tokens (and not divisible units), if there are 2 tokens left and someone buys 1.5, then the remaining 0.5 tokens can never be purchased as tokensAvailable will return 0 due to the whole number calculation.

maxPurchase

```
function maxPurchase() public constant returns (uint spend) {
    return tokenCap * currentPrice() - totalAccounted;
}
```

The `maxPurchase` function returns the largest purchase that is available in the sale.

This function is not used by another functions but it is publicly callable and constant, so it is only an informational to know the maximum allowed value for the next purchase.

Arithmetic Security

Because this function is not called by any other function in the contract it poses no security risk.

theDeal

```
function theDeal(uint _value)
    public
    constant
    when_active
    returns (uint accounted, bool refund, uint price)
{

    uint _bonus = bonus(_value);

    price = currentPrice();
    accounted = _value + _bonus;

    uint available = tokensAvailable();
    uint tokens = accounted / price;
    refund = (tokens > available);
}
```

The theDeal function is the main logic for setting prices of tokens in the auction. If the crowdsale is not active, the function returns nothing.

It returns a tuple consisting of: - accounted, calculated by adding sent wei by bonus's value - refund, set to true if tokens that would be received are more than what is available - price, taken from currentPrice's calculation

Arithmetic Security

No overflows are possible due to the bounds of _value and bonus() as described in buyin and bonus breakdowns.

bonus

```
function bonus(uint _value)
    public
    constant
    when_active
    returns (uint extra)
{
    if (now < beginTime + BONUS_DURATION) {
        return _value * BONUS_SIZE / 100;
    }
    return 0;
}
```

The bonus function returns a 15% bonus to the amount of tokens purchased per Ether if the tokens are purchased within the BONUS_DURATION (1st hour) of the crowdsale starting.

If it is called after BONUS_DURATION has elapsed or after the sale has ended, then it returns 0.

Arithmetic Security

The value returned by bonus cannot overflow as _value will never be big enough for that to happen, and due to dust prevention it will never be low enough to flatten to 0..

isActive

```
function isActive() public constant returns (bool) { return now >= beginTime && now < endTime; }
```

The `isActive` function checks that that the time is between the set start and end times for the sale. It makes up the `when_active` modifier.

allFinalised

```
function allFinalised() public constant returns (bool) { return now >= endTime && totalAccounted == totalFinalised; }
```

The function `allFinalised` returns true when the sale is past its end time and all allocations have been finalized.

isBasicAccount

```
    function isBasicAccount(address _who) internal constant returns (bool) {
        uint senderCodeSize;
        assembly {
            senderCodeSize := extcodesize(_who)
        }
        return senderCodeSize == 0;
    }
```

The `isBasicAccount` function uses the EXTCODESIZE opcode to check that the length of the code stored at the address is 0 - effectively making sure it's a non-contract account. This function makes up the `only_basic` modifier.

SecondPriceAuction Private Functions

flushEra

```
function flushEra() private {
    uint currentEra = (now - beginTime) / ERA_PERIOD;
    if (currentEra > eraIndex) {
        Ticked(eraIndex, totalReceived, totalAccounted);
    }
    eraIndex = currentEra;
}
```

The `flushEra` function just updates the eraIndex if 5 or more minutes have elapsed since the last buyin and emits a `Ticked` event. It is only called by the `buyin` function.

FrozenToken.sol

The FrozenToken contract implements the “token” for the crowdsale. The only ERC20 interface function it supports is `transfer`, and all participant tokens are frozen by default and cannot be moved without an unfreeze from the contract owner.

The contract inherits only from Owned, which implements the standard owner pattern:

```
contract FrozenToken is Owned
```

Constructor

```
function FrozenToken(uint _totalSupply, address _owner)
public
        when_non_zero(_totalSupply)
{
        totalSupply = _totalSupply;
        owner = _owner;
        accounts[_owner].balance = totalSupply;
        accounts[_owner].liquid = true;
}
```

The constructor ensures that no Ether was sent when creating the contract, and enforces a non-zero starting supply.

The entire supply is allocated to the owner, and its token balance is unfrozen (“liquid”) by default.

balanceOf

```
function balanceOf(address _who) public constant returns (uint256) {
        return accounts[_who].balance;
}
```

Simply returns the balance of account associated to _who. Standard ERC20 behaviour.

makeLiquid

```
function makeLiquid(address _to)
        public
        when_liquid(msg.sender)
        returns(bool)
{
        accounts[_to].liquid = true;
        return true;
}
```

The makeLiquid function unfreezes an account.

Due to the when_liquid modifier, it can only be called by an account that is itself unfrozen. No ether can be sent to this function.

transfer

```
function transfer(address _to, uint256 _value)
        public
        when_owns(msg.sender, _value)
        when_liquid(msg.sender)
        returns(bool)
{
        Transfer(msg.sender, _to, _value);
        accounts[msg.sender].balance -= _value;
        accounts[_to].balance += _value;

        return true;
}
```

The FrozenToken's transfer function acts like the usual ERC20 transfer, except that the account of msg.sender must be in the unfrozen state (liquid set to true), otherwise the function will throw.

Arithmetic Security

As above, there will just not be enough tokens in existence to cause an overflow. The `when_owns` ensures that the sender owns at least `_value` so underflow is not possible either.

Default Function

```
function() public {
        assert(false);
}
```

The default function simply throws, such that no Ether can be sent to this contract by mistake.

MultiCertifier.sol

The last contract is MultiCertifier, which implements an interface that SecondPriceAuction calls to check whether a participant (a caller of `buyin`) is allowed to purchase tokens.

It also allows the owner to add and remove certifier “delegates”, which are able to manipulate the mapping of certified addresses.

The contract is logically simple, consisting of only four non-constant functions `certify`, `revoke`, `addDelegate`, and `removeDelegate` and four modifiers for constraint checking.

The contract inherits from Owned and Certifier:

```
contract MultiCertifier is Owned, Certifier
```

Which implement the Owned pattern and define the Certifier interface, respectively.

Constructor

The contract has no constructor beyond the one it inherits from Owned.

certify

```
function certify(address _who)
        only_delegate
        only_uncertified(_who)
```

```
{
        certs[_who].active = true;
        certs[_who].certifier = msg.sender;
        Confirmed(_who, msg.sender);
}
```

The `certify` function certifies an address, if and only if it hasn't already been certified and if the caller is on the list of delegates, or the contract owner. It emits a `Confirmed` event on success.

revoke

```
function revoke(address _who)
        only_certifier_of(_who)
        only_certified(_who)
{
        certs[_who].active = false;
        Revoked(_who, msg.sender);
}
```

The `revoke` function allows a certififer to revoke certification for a certified address, if and only if they are the ones who certified it (with the exception of thecontract owner, who can revoke arbitrarily).

addDelegate

```
function addDelegate(address _new) only_owner { delegates[_new] = true; }
```

The `addDelegate` function can be used by the owner to add a new delegate address with certification abilities.

removeDelegate

```
function removeDelegate(address _old) only_owner { delete delegates[_old];
    }
```

The `removeDelegate` function can be used by the owner to add a remove a delegate address from the list of certifiers.

certified

```
function certified(address _who) constant returns (bool) { return certs[_who].active; }
```

This is a constant function that returns whether an address has been certified. This is the only function in this contract called by SecondPriceAuction.

getCertifier

```
function getCertifier(address _who) constant returns (address) { return certs[_who].certifier;
```

This is a constant function that returns the certifier for a certain address.