SIRIN Crowdsale Audit

Zero Knowledge Labs Auditing Services

AUTHOR: MATTHEW DI FERRANTE & DEAN EIGENMANN

2017-12-08

Audited Material Summary

The audit consists mainly of the `SirinCrowdsale.sol`, `RefundVault.sol` and `SirinSmartToken.sol` contracts. The git commit hash of the reviewed files is `7ef03f846166d64fb12b830bd2e8cbd77577c0e8`.

The majority of custom logic is in `SirinCrowdsale` and `RefundVault`. All other contracts are either taken from OpenZeppelin or are trivial modifications of those contracts.

Overall Security

The contracts have no critical issues that would pose a risk to users or their funds.

SirinCrowdsale.sol

The `SirinCrowdsale` contract implements the crowsale logic. It inherts most of its functionality from OpenZeppelin's `FinalizableCrowsale` and `Crowdsale` contracts (with an addition to enable them to handle SmartTokens and a tranche rate), and uses Bancor's `LimitedTransferSmartToken` as its token implementation.

```
contract SirinCrowdsale is FinalizableCrowdsale
```

Security

No critical issues have been found.

Constructor

```
    function SirinCrowdsale(uint256 _startTime,
    uint256 _endTime,
    address _wallet,
    address _walletFounder,
    address _walletOEM,
    address _walletBounties,
    address _walletReserve,
    SirinSmartToken _sirinSmartToken,
    RefundVault _refundVault)
    public
    Crowdsale(_startTime, _endTime, EXCHANGE_RATE, _wallet,
        _sirinSmartToken) {
        require(_walletFounder != address(0));
        require(_walletOEM != address(0));
        require(_walletBounties != address(0));
        require(_walletReserve != address(0));
        require(_sirinSmartToken != address(0));
        require(_refundVault != address(0));

        walletFounder = _walletFounder;
        walletOEM = _walletOEM;
        walletBounties = _walletBounties;
        walletReserve = _walletReserve;

        token = _sirinSmartToken;
        refundVault  = _refundVault;
    }
```

The constructor performs sanity checking on wallet arguments to ensure they aren't the 0 address, and assigns them to their respective storage variables.

getRate

```
    function getRate() public view returns (uint256) {
        if (now < (startTime.add(24 hours))) {return 1000;}
        if (now < (startTime.add(2 days))) {return 950;}
        if (now < (startTime.add(3 days))) {return 900;}
        if (now < (startTime.add(4 days))) {return 855;}
        if (now < (startTime.add(5 days))) {return 810;}
        if (now < (startTime.add(6 days))) {return 770;}
        if (now < (startTime.add(7 days))) {return 730;}
        if (now < (startTime.add(8 days))) {return 690;}
        if (now < (startTime.add(9 days))) {return 650;}
        if (now < (startTime.add(10 days))) {return 615;}
        if (now < (startTime.add(11 days))) {return 580;}
        if (now < (startTime.add(12 days))) {return 550;}
        if (now < (startTime.add(13 days))) {return 525;}

        return rate;
    }
```

The `getRate` function returns a tranched rate for the crowdsale, which returns a higher rate for earlier contributions, up until 2 weeks past crowdsale launch, beyond which it simply returns `EXCHANGE_RATE` (500).

finalization

```
    function finalization() internal onlyOwner {
        super.finalization();

        // granting bonuses for the pre crowdsale grantees:
        for (uint256 i = 0; i < presaleGranteesMapKeys.length; i++) {
            token.issue(presaleGranteesMapKeys[i], presaleGranteesMap[
                presaleGranteesMapKeys[i]]);
        }

        // Adding 60% of the total token supply (40% were generated during
             the crowdsale)
        // 40 * 2.5 = 100
        uint256 newTotalSupply = token.totalSupply().mul(250).div(100);

        // 10% of the total number of SRN tokens will be allocated to the
            founders and team
        token.issue(walletFounder, newTotalSupply.mul(10).div(100));

        // 10% of the total number of SRN tokens will be allocated to OEMs
            , Operating System implementation,
        // SDK developers and rebate to device and Shield OS users
        token.issue(walletOEM, newTotalSupply.mul(10).div(100));

        // 5% of the total number of SRN tokens will be allocated to
            professional fees and Bounties
        token.issue(walletBounties, newTotalSupply.mul(5).div(100));

        // 35% of the total number of SRN tokens will be allocated to
            SIRIN LABS,
        // and as a reserve for the company to be used for future
            strategic plans for the created ecosystem
        token.issue(walletReserve, newTotalSupply.mul(35).div(100));

        // Re-enable transfers after the token sale.
        token.disableTransfers(false);

        // Re-enable destroy function after the token sale.
        token.setDestroyEnabled(true);

        // Enable ETH refunds and token claim.
        refundVault.enableRefunds();

        // transfer token ownership to crowdsale owner
        token.transferOwnership(owner);

        // transfer refundVault ownership to crowdsale owner
        refundVault.transferOwnership(owner);
    }
```

The `finalization` function is called by the contract owner when the token sale is over. It issues token grants to the presale grantees, and issues tokens to the various wallets.

The contract also enables transfers, enables destruction of tokens, turns on refunds in the `RefundVault`, and transfers ownership of the SmartToken and RefundVault contracts to the crowdsale owner.

getTotalFundsRaised

```
    function getTotalFundsRaised() public view returns (uint256) {
        return fiatRaisedConvertedToWei.add(weiRaised);
    }
```

The `getTotalFundsRaised` function returns the fiat amount raised (converted to wei) + the amount of ether raised (in wei).

isActive

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

The `isActive` function simply returns true if the date is between the Crowdsale's start and end time.

addUpdateGrantee

```
    function addUpdateGrantee(address _grantee, uint256 _value) external
       onlyOwner onlyWhileSale{
        require(_grantee != address(0));
        require(_value > 0);

        // Adding new key if not present:
        if (presaleGranteesMap[_grantee] == 0) {
            require(presaleGranteesMapKeys.length < MAX_TOKEN_GRANTEES);
            presaleGranteesMapKeys.push(_grantee);
            GrantAdded(_grantee, _value);
        }
        else {
            GrantUpdated(_grantee, presaleGranteesMap[_grantee], _value);
        }

        presaleGranteesMap[_grantee] = _value;
    }
```

The `addUpdateGrantee` function allows the contract owner to add or update a grantee's allowed grant while the crowdsale is running.

It emits a `GrantAdded` event or a `GrantUpdated` event on success.

deleteGrantee

```
    function deleteGrantee(address _grantee) external onlyOwner
       onlyWhileSale {
        require(_grantee != address(0));
        require(presaleGranteesMap[_grantee] != 0);

        //delete from the map:
        delete presaleGranteesMap[_grantee];

        //delete from the array (keys):
        uint256 index;
        for (uint256 i = 0; i < presaleGranteesMapKeys.length; i++) {
            if (presaleGranteesMapKeys[i] == _grantee) {
                index = i;
                break;
            }
        }
```

```
            presaleGranteesMapKeys[index] = presaleGranteesMapKeys[
                presaleGranteesMapKeys.length - 1];
            delete presaleGranteesMapKeys[presaleGranteesMapKeys.length - 1];
            presaleGranteesMapKeys.length--;

            GrantDeleted(_grantee, presaleGranteesMap[_grantee]);
    }
```

The `deleteGrantee` function allows the contract owner to delete a grantee while the crowdsale is running.

The `presaleGranteesMap` is zero'd out and the key is deleted from `presaleGranteesMapKeys` through swap and array resize.

It emits a `GrantDeleted` event on success.

setFiatRaisedConvertedToWei

```
    function setFiatRaisedConvertedToWei(uint256 _fiatRaisedConvertedToWei
        ) external onlyOwner onlyWhileSale {
            fiatRaisedConvertedToWei = _fiatRaisedConvertedToWei;
            FiatRaisedUpdated(msg.sender, fiatRaisedConvertedToWei);
    }
```

The `setFiatRaisedConvertedToWei` function sets the fiat amount raised. It can only be called by the owner, and only while the crowdsale is running. It emits a `FiatRaisedUpdated` event on success.

claimTokenOwnership

```
    function claimTokenOwnership() external onlyOwner {
            token.claimOwnership();
    }
```

The `claimTokenOwnership` function enables the crowdsale to take ownership of the `SirinSmartToken` contract. It can only be called by the crowdsale owner.

claimRefundVaultOwnership

```
    function claimRefundVaultOwnership() external onlyOwner {
        refundVault.claimOwnership();
    }
```

The `claimRefundVaultOwnership` function is a passthrough to `RefundVault`'s `claimOwnership` function. It can only be called by the crowdsale owner.

buyTokensWithGuarantee

```
    function buyTokensWithGuarantee() public payable {
        require(validPurchase());

        uint256 weiAmount = msg.value;

        // calculate token amount to be created
        uint256 tokens = weiAmount.mul(getRate());
        tokens = tokens.div(REFUND_DIVISION_RATE);

        // update state
        weiRaised = weiRaised.add(weiAmount);

        token.issue(address(refundVault), tokens);

        refundVault.deposit.value(msg.value)(msg.sender, tokens);

        TokenPurchaseWithGuarantee(msg.sender, address(refundVault),
            weiAmount, tokens);
    }
```

The `buyTokensWithGuarantee` function is a public function that enables a user to buy tokens that can be refunded. It enforces `validPurchase` from the `Crowdsale` contract.

The rate for purchase of refundable tokens is 50% of non-refundable tokens. The invested Ether is deposited into the vault, instead of being forwarded to the crowdsale wallet. The tokens are issued to the `RefundVault` as opposed to being issued to the purchaser directly.

A `TokenPurchaseWithGuarantee` event is emitted on success.

SirinSmartToken.sol

This contract simply instantiates a Bancor Limited Transfer SmartToken:

```
contract SirinSmartToken is LimitedTransferBancorSmartToken
```

Security

This contract has no security issues.

Constructor

```
    function SirinSmartToken() public {
        //Apart of 'Bancor' computability - triggered when a smart token
           is deployed
        NewSmartToken(address(this));
    }
```

Only the constructor is populated, which is just a passthrough to the Bancor contract's constructor.

RefundVault.sol

This contract is an extension of OpenZeppelin's `RefundVault` that enables it to handle token refunds. The functions also have some extra modifiers/restrictions placed on them.

The audit for this contract will focus on the main extension of functionality - `refundETH`, `claimTokens` and `claimAllTokens`.

refundETH

```
    function refundETH(uint256 ETHToRefundAmountWei) isInRefundTimeFrame
       isRefundingState public {
        require(ETHToRefundAmountWei != 0);

        uint256 depositedTokenValue = depositedToken[msg.sender];
        uint256 depositedETHValue = depositedETH[msg.sender];

        require(ETHToRefundAmountWei <= depositedETHValue);

        uint256 refundTokens = ETHToRefundAmountWei.mul(
           depositedTokenValue).div(depositedETHValue);

        assert(refundTokens > 0);

        depositedETH[msg.sender] = depositedETHValue.sub(
            ETHToRefundAmountWei);
        depositedToken[msg.sender] = depositedTokenValue.sub(refundTokens)
            ;

        token.destroy(address(this),refundTokens);
        msg.sender.transfer(ETHToRefundAmountWei);

        RefundedETH(msg.sender, ETHToRefundAmountWei);
    }
```

The `refundETH` function allows an investor to get a refund for their SRN token purchase. This can only be called in the refund time frame, and only if the vault is in the refund state.

The function takes a wei argument, proportionally destroys the amount of tokens that value corresponds to, and refunds the ETH to the caller.

On success a `RefundedETH` event is emitted.

claimTokens

```

    function claimTokens(uint256 tokensToClaim) isRefundingOrCloseState
        public {
        require(tokensToClaim != 0);

        address investor = msg.sender;
        require(depositedToken[investor] > 0);

        uint256 depositedTokenValue = depositedToken[investor];
        uint256 depositedETHValue = depositedETH[investor];

        require(tokensToClaim <= depositedTokenValue);

        uint256 claimedETH = tokensToClaim.mul(depositedETHValue).div(
            depositedTokenValue);

        assert(claimedETH > 0);

        depositedETH[investor] = depositedETHValue.sub(claimedETH);
        depositedToken[investor] = depositedTokenValue.sub(tokensToClaim);

        token.transfer(investor, tokensToClaim);
        if(state != State.Closed) {
            etherWallet.transfer(claimedETH);
        }

        TokensClaimed(investor, tokensToClaim);
    }
```

The `claimTokens` function allows an investor to claim their tokens from the `RefundVault`. It can only be called if the crowdsale is in the `Closed` or `Refunding` state.

On success, a `TokensClaimed` event is submitted, and the `RefundVault` transfers tokens to the `investor`. A caller of this function may not claim more token than are deposited for a certain investor.

claimAllInvestorTokensByOwner

```
    function claimAllInvestorTokensByOwner(address investor) isCloseState
        onlyOwner public {
        uint256 depositedTokenValue = depositedToken[investor];
        require(depositedTokenValue > 0);


        token.transfer(investor, depositedTokenValue);

        TokensClaimed(investor, depositedTokenValue);
    }
```

The `claimAllInvestorTokensByOwner` function can be called by the owner when the vault is closed to transfer any deposited tokens to an investor that has an outstanding balance in the `depositedTokens` map.

The function emits a `TokensClaimed` event on success.

claimAllTokens

```
    function claimAllTokens() isRefundingOrCloseState public  {
        uint256 depositedTokenValue = depositedToken[msg.sender];
        claimTokens(depositedTokenValue);
    }
```

The `claimAllTokens` function simply calls `claimTokens` with the entire deposited balance as the value argument.

It can only be called if the vault is in the refunding or close states.

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/