Audit Report

Pocketinns Smart Contract

Prepared by:

DevGenesis Private Limited

Registered Office:
Building No. 10,
N.R. Rai Singh Chowk,
Barola, Sector-49, Noida - 201301, (U.P.) India

Corporate Office:
A-27/L, Second Floor,
Near Car Market, Sector-16, Noida - 201301, (U.P.) India

CIN: U74999UP2017PTC095066
Email ID: admin@devgenesis.com

Introduction

This is a technical audit for Pocketinns token smart contract. This documents outlines our methodology, limitations and results for our security audit.

Token name– Pocketinns Token

Token Symbol- Pinns

Decimals allowed- 18

Pinns Token Total Supply– 150000000

Synopsis

Overall, the code demonstrates high code quality standards adopted and effective use of concept and modularity. Pocketinns contract development team demonstrated high technical capabilities, both in the design of the architecture and in the implementation.

Code Analysis

Besides, the results of the automated analysis, manual verification was also taken into account. The complete contract was manually analyzed, every logic was checked and compared with the one described in the whitepaper. The manual analysis of code confirms that the Contract doesnot contain any serious susceptibility. No divergence was found between the logic in Smart Contract and the whitepaper.

Scope

This audit is into the technical and security aspects of the Pocketinns smart contract. The key aim of this audit is to ensure that funds contributed to these contracts are not by far attacked or stolen by any third parties. The next aim of this audit is that ensure the coded algorithms work as expected. The audit of Smart Contract also checks the implementation of token mechanism i.e. the Contract must follow all the ERC20 Standards.

DEVGENESIS is one of the parties that independently audited Pocketinns Smart Contract.This audit is purely technical and is not an investment advice. The scope of the audit is limited to the following source code files:

- **DutchAuction.sol**
- **AbstractERC20.sol**
- **StandardERC20.sol**
- **Safemath.sol**

Limitations

Security auditing cannot bare all existing vulnerabilities, and even an audit in which no vulnerabilities are found is not a guarantee for a secure smart contract. However, auditing is there to find out vulnerabilities that were unobserved during development and areas where additional security measures are necessary. Some of the issues may affect the intact smart contract application, while some might lack protection only in certain areas. We therefore carry out a source code review to determine all locations that need to be fixed. **DEVGENESIS** has performed widespread auditing in order to discern as many susceptibilities as possible.

Traditional way of Software Development

The code was provided to the auditors on Github. The codebase was properly version controlled.The code is written for Solidity version 0.4.18.The codebase uses community administered high quality Open Zeppelin framework. This software development practices and components match the expected community standards. The recent audited code is the commit

https://github.com/pocketinns/PocketinnsContracts

About the Project

Project Architechture

The project consists of four files as following:

1. The **AbstractERC20.sol** file contains Token contract that implements the ERC20 standard interface.

2. The **safemath.sol** file contains SafeMath library that implements the OpenZepplin library used for Math operations and validations.
3. The **StandardERC20.sol** file contains PocketinnsToken contract that implements the token contract, supplemented with the several service functions.
4. The **DutchAuction.sol** file contains pinnsDutchAuction contract that implements PocketinnsToken, supplemented with the function for funds receiving(fallback), Token Distribution , Bonus Distribution and several service functions.

Code Logic

The Pocketinns contract is supplemented with the logic of funds receiving with three constraints on time(**16 days**), amount raised (**46 Million USD**) and Hard Cap((**30 Million**) tokens sold.

In order to make it possible to transfer the funds to the contract, the contract owner have to call the **startICO()** function, after this it will be possible to transfer funds to the DutchAuction Contract address. When sending funds to the pinnsDutchAuction contract, **itoBids** service variable records how many ETH was sent by the investor during different time of the complete Dutch Auction.

After the Dutch Auction, the leftover tokens for ICO are Burnt and the Last price per token gets fixed by **finalizeAuction()** function.Tokens can be claimed by the investors after the Auction ends using **claimTokensICO()** function.

Additionally, first four days investors will be given Bonus Tokens as per the funds sent by them. Bonus Token distribution will start once the contract owner calls **startGoodwillDistribution()** function.

Good will Tokens can be claimed by the investors after the Auction ends using **claimGoodwillTokens()** function passing the address of goodwill token receiver as an parameter.All the functions and state variables are well commented using the natspec documentation for the functions which is good to understand quickly how everything is supposed to work.

Testing

Pocketinns smart contract went through rigorous testing, which focused on the security features of the smart contract. During the complete Test phase the security of the Project was prime consideration. Major Task was to find and describe the security issues in the Smart Contract.

Primary checks followed during testing of Smart Contract is to see that if code :

➔ We check the Smart Contracts Logic and compare it with one described in Whitepaper.

➔ The contract code should follow the Conditions and logic as per user request.

➔ We deploy the Contract and run the Tests.

➔ We make sure that Contract does not lose any money/Ether.

Vulnerabilities Check:

Smart Contract was scanned for commonly known and more specific vulnerabilities.Following are the list of commonly known vulnerabilities that was considered during the audit of the smart contract:

- **TimeStamp Dependence**:

The timestamp of the block can be manipulated by the miner, and so should not be used for critical components of the contract. *Block numbers* and *average block time* can be used to estimate time (suggested). Pocketinns smart contract uses timestamp only to accept the Ether and it is not critically vulnerable to this type of attack.

- **Gas Limit and Loops**:

Loops that do not have a fixed number of iterations, hence due to normal operation, the number of iterations in a loop can grow beyond the block gas limit which can cause the complete contract to be stalled at a certain point. Pocketinns smart contract is free from the gas limit check as the contract code does not contain any loop in its code.

- **Compiler Version:**

Contracts should be deployed with the same compiler version and flags that they have been tested the most with. Locking the pragma helps ensure that contracts do not accidentally get deployed using, for example, the latest compiler which may have higher risks of undiscovered bugs. Pocketinns smart contract is locked to a specific compiler version of **0.14.18** which is good coding practice.

- **ERC20 Standards:**

Pocketinns smart contract follows all the universal ERC20 coding standards and implements all its functions and events in the contract code. This standard is followed in the **AbstarctERC20.sol** file of the Pocketinns smart contract Github link.

- **Redundant fallback function:**

The standard execution cost of a fallback function should be less than 2300 gas, Pocketinns smart contract code is free from this vulnerability.

- **Unchecked math:**

Need to guard uint overflow or security flaws by implementing the proper math logic checks. The Pocketinns smart contract uses the popular SafeMath library for critical operations to avoid arithmetic over- or underflows and safeguard against unwanted behavior. The Safemath library is implemented in **safemath.sol** file. In particular the **balances** variable is updated using the safemath operation.

- **Exception disorder:**

When an exception is thrown, it cannot be caught: the execution stops, the fee is lost. The irregularity in how exceptions are handled may affect the security of contracts.

- **Unsafe type Inference:**

It is not always necessary to explicitly specify the type of a variable, the compiler automatically infers it from the type of the first expression that is assigned to the variable.

- **Reentrancy:**

The reentrancy attack consists of the recursively calling a method to extract ether from a contract if user is not updating the balance of the sender before sending the ether.In Pocketinns smart contract calls to external functions happen after any changes to state variables in the contract so the contract is not vulnerable to a reentrancy exploit. The Pocketinns smart contract does not have any vulnerabilities against reentrancy attack.

- **Dos with (Unexpected) Throw:**

The Contract code can be vulnerable to the Call Depth Attack! So instead, code should have a pull payment system instead of push. The Pocketinns smart contract securely handles the all the payment related scenario thus it is not vulnerable to this attack.

- **Dos with Block Gas LImit:**

In a contract by paying out to everyone at once, contract risk running into the block gas limit. Each Ethereum block can process a certain maximum amount of computation. If one try to go over that, the transaction will fail. Pocketinns smart contract implements the push over pull payment to remove the above issue.

- **Explicit Visibility in functions and state variables:**

Explicit visibility in the function and state variables are provided. Visibility like **external, internal, private** and **public** is used and defined properly.

Features in Smart Contract

A)StandardERC20.sol

1. Transferrable - The tokens can be transferred from one entity to other(like to exchanges for trading). This transfer can be made by using any ethereum wallet which supports ERC20 token standard, for eg. MyEtherWallet, Mist Etc.

Code Line: 73-85

```
function transfer(address _to, uint256 _value)
public
returns (bool)
{
if (balances[msg.sender] < _value) {
// Balance too low
revert();
}
balances[msg.sender] = balances[msg.sender].sub(_value);
balances[_to] = balances[_to].add(_value);
Transfer(msg.sender, _to, _value);
return true;
}
```

2. **Approvable** - Any Token holder if he wills, can approve some other address, who will on his behalf transfer the approved amount of tokens from token holder's to others.

Code Line: 111-118

```
 function approve(address _spender, uint256 _value)public returns (bool success) {
allowed[msg.sender][_spender] = _valuet;
Approval(msg.sender, _spender, _value);
return true;
}
```

3. **Viewable Tokens** - As you already may be aware with the transparent nature of blockchain, all the tokens holders and their exact balance is made clearly visible, on Ethereum blockchain explorers like Etherscan and Ethplorer. Their are

functions which are implemented to return informations like token balance of any particular token holder, the token allowance amount of any particular Token holder, which he has allowed to any other entity(Ethereum address).

Code Line: 138-144

```
function balanceOf(address _owner)public view returns (uint256 balance) {
return balances[_owner];
}
```

4. **Burning of Tokens** - The leftover tokens are meant to be burnt, if all the 30 million Coins are not sold during the DutchAuction process, Thereby decreasing the Total Supply by that amount,this is done only only once the Auction is finalised smart contract.

Code Line: 61 - 68

```
function burnLeftItoTokens(uint _burnValue)
public
onlyForDutchAuctionContract
{
totalSupply = totalSupply.sub(_burnValue);
balances[dutchAuctionAddress] = balances[dutchAuctionAddress].sub(_burnValue);
}
```

B)DutchAuction.sol

1. **Ownable-** The smart contract and the tokens implemented it are owned by a particular entity (ethereum public address). To use those tokens the owner will have to sign with his private key. As we deploy the smart contract on Ethereum blockchain, initially all the tokens will be owned by the owner of the smart contract i.e. the entity offering the crowdsale (we walk you through the procedure to deploy the token, so you will own all the tokens before crowdsale starts).

Code Line: 75-80 and 88-95

Modifier:

```
modifier onlyOwner() {
if (msg.sender != owner) {
revert();
}
 _;
}
```

Constructor :

```
function pinnsDutchAuction(uint256 EtherPriceFactor)
public
{
```

```
require(EtherPriceFactor != 0);
owner = msg.sender;
stage = Stages.AuctionDeployed;
priceFactor = EtherPriceFactor;
}
```

2. **SetWhiteList Address:** Once the investor is whitelisted by KYC process , the whitelisting of the investors addresses in the Smart Contract is done only by the owner of the smart contract.The Whitelisted Address are only eligible to claim their tokens once the Token distribution starts.

Code Line: 196-198

```
function setWhiteListAddresses(address _investor) external isOwner
{
whitelisted[_investor] = true;
}
```

3. **Current Price in the Auction:** During the Dutch Auction process the price change occurs in every 12 hours. After sending the funds Investor can view the Current price running in the DutchAuction.

Code Line: 138-150

```
funcion getCurrentPrice() public
{
totalTokensSold = ((totalReceived.mul(priceFactor.mul(100))).div(currentPerTokenPrice));
uint256 priceCalculationFactor = (block.timestamp.sub(startItoTimestamp)).div(43200);
if(priceCalculationFactor <=16)
{
currentPerTokenPrice = (price1).sub(priceCalculationFactor.mul(100));
}
else if (priceCalculationFactor > 16 && priceCalculationFactor <= 31)
{
currentPerTokenPrice = (price2).sub((((priceCalculationFactor.mul(100)).sub(1600))).div(2));
}
}
```

4. **Claimable Tokens:** After the Dutch Auction ends, Price per token is fixed and these tokens can be claimed after the Auction ends.

Code Line: 180-192

```
function claimTokensICO(address receiver) public
atStage(Stages.AuctionEnded)
isValidPayload
{
if (receiver == 0)
receiver = msg.sender;
require(whitelisted[receiver]);
if(itoBids[receiver] >0)
{
uint256 tokenCount = (itoBids[receiver].mul(priceFactor.mul(100))).div(finalPrice);
itoBids[receiver] = 0;
pinnsToken.transfer(receiver, tokenCount);
}
}
```

5. **Goodwill Tokens/ Bonus Tokens :** Additional tokens will be distributed to the first four days investors as per their contribution. The Good will tokens with respect to investor is recorded in the Smart contract code, to later claim these tokens. Owner of the Contract will start distribution which can later be claimed by the investors.

Code Line: 196-203 and 205-218

```
 function  startGoodwillDistribution()
external
atStage(Stages.AuctionEnded)
isOwner
{
require (pinnsToken.balanceOf(address(this)) != 0);
stage = Stages.goodwillDistributionStarted;
}


function claimGoodwillTokens(address receiver)
atStage(Stages.goodwillDistributionStarted)
public
isValidPayload
{
if (receiver == 0)
receiver = msg.sender;
if(goodwillBonusStatus[msg.sender] == true)
{
goodwillBonusStatus[msg.sender] = false;
uint bonus = bonusTokens[msg.sender];
pinnsToken.transfer(msg.sender, bonus);
}}
```

Risk

The Pocketinns Smart Contract has no the risk of losing any amounts of ethers in case of external attack or a bug, as contract does not takes any kind of funds from the user. If anyone tries to send any amount of ether to the contract address, the transaction will cancel itself and no ether comes to the contracts.

The flow of tokens from this Pocketinns contract can be controlled using a script running on the backend and visually through EtherScan.io. By using EtherScan.io working of code can be verified which will lead the compiled code getting matched with the bytecode of deployed smart contract in the blockchain.

Therefore, there is no anomalous gap in the Pocketinns smart contract, tokens can be claimed by the investors after the DutchAuction as per the amount paid by them during DutchAuction. As all the funds are held with owner's address/Smart Contract address, so any possible losses due to flaws in the Pocketinns smart contract is not possible to occur.

Conclusion

In this report, we have concluded about the security of Pocketinns Smart Contract. The smart contract has been analyzed under different facets. Code quality is very good, and well modularized. We found that Pocketinns smart contract adapts a very good coding practice and have clean, documented code. Smart Contract logic was checked and compared with the one described in the whitepaper. No discrepancies were found.