HarryPotterObamaSonic10Inu Analysis

This is a technical paper explaining the code, liquidity pool and holders of **HarryPotterObamaSonic10Inu** and why it can't be rugged.

Author: @Azephiar.

The code of the contract can be found **here.**

When using the term "**querying the contract**" what I mean is using **this** page to query the contract and see what the results are.

It's a safemoon fork. This means the code is the same, or to the least very similar.
10% tax on selling and 10% tax on buying.

CODE ANALYSIS

The first 400 lines of code are just standard libraries and interfaces. We don't care about this.

LINES 401-474
This bunch of lines defines what an owner is, ownership renouncement and owner lock.
This is what the owner is defined as:

```
function owner() public view returns (address) {
    return _owner;
}
```

If we query the contract to know the owner, the result is a dead address. 0x00.
This is because ownership has been renounced as we can tell from **this** transaction.

If we are careful enough we notice a tricky funcion here:

```
function lock(uint256 time) public virtual onlyOwner {
    _previousOwner = _owner;
    _owner = address(0);
    _lockTime = now + time;
    emit OwnershipTransferred(_owner, address(0));
}
```

What's happening here is that the previous owner of the contract is saved in a variable, after that a new owner (0x00) is setted. This means it's possible to actually have a fake-renounce.

This has not happened because this function has never been called before ownership got renounced. You can check this **here**.

Why is this function here? Probably because it's a copy of safemoon and nobody dared to remove it.

Last thing we have to take a look at:

```
modifier onlyOwner() {
    require(_owner == _msgSender(), "Ownable: caller is not the owner");
    _;
}
```

This is a modifier. It's something that can be called with a function and it's widely used in this contract. The modifier is always executed before the function is called with. In this case this means that if the caller is not the owner (the address 0x00) the function will not execute.

To put it in simple terms: **every function that has the modifier onlyOwner() can't be called because the owner is the address 0x00, and nobody has access to that address.**

LINES 474-689
These are just the interfaces that make it possible to connect the token to pancakeswap, nothing is going on here.

LINES 690-739
All happening here is definition and initialization of some variables. For instance:

```
uint256 private _tTotal = 1000000000 * 10**6 * 10**9;
string private _name = "HarryPotterObamaSonic10Inu";
string private _symbol = "BITCOIN";
uint8 private _decimals = 9;
uint256 public _taxFee = 5;
uint256 public _liquidityFee = 5;
uint256 public _maxTxAmount = 5000000 * 10**6 * 10**9;
uint256 private numTokensSellToAddToLiquidity = 500000 * 10**6 * 10**9;
```

This sets name, decimals, supply, ticker. We notice there's also a maximum amount per transaction, which is 0.5% of total supply.

LINES 740-883
Nothing is really going on here. The constructor initializes the contract without anything shady going on and after that there are a bunch of public functions to get basic data about the token like name, symbol, etc.
There are also a bunch of functions for conversion token-reflections.
There's a function for airdrops called deliver.
There's a function for transferring tokens from two accounts excluded from fees.

The old owner excluded nobody from the fees before renouncing so we know as a matter of fact that there are only 3 accounts excluded from fees:

- Pancakeswap router.

- Creator of the contract (excluded himself in the constructor).
- Contract token itself.

LINES 884-896

```
function setTaxFeePercent(uint256 taxFee) external onlyOwner() {
    _taxFee = taxFee;
}

function setLiquidityFeePercent(uint256 liquidityFee) external onlyOwner() {
    _liquidityFee = liquidityFee;
}

function setMaxTxPercent(uint256 maxTxPercent) external onlyOwner() {
    _maxTxAmount = _tTotal.mul(maxTxPercent).div(
        10**2
    );
}
```

This is an interesting piece of code. We can see a bunch of functions that can be used to set fees. Now think about it. What happens if the fee is set to 100%? Right. Nobody can sell tokens anymore. Is this function possible to call? No, because it has the modifier **onlyOwner()**, as discussed above.

We can easily tell all 3 of these functions have the modifier onlyOwner(), conclusion being these functions are useless.

LINES 897-955

Just a bunch of utility functions used to convert tokens amount to reflection amount. This allows the automatic distribution of tokens. It's a very clever and efficient way to distribute dividends but I'm not going into details here.

LINES 956-967

```
function calculateTaxFee(uint256 _amount) private view returns (uint256) {
    return _amount.mul(_taxFee).div(
        10**2
    );
}

function calculateLiquidityFee(uint256 _amount) private view returns (uint256) {
    return _amount.mul(_liquidityFee).div(
        10**2
    );
}
```

These functions control the transaction fees. The first one calculates the amount of tokens to be distributed and the second one calculates the amount of tokens to add to the liquidity pool. Both of them are set at 5% and can't be changed, as stated before.

LINES 968-994

Here are 2 functions to remove and restore fees used later in the function to transfer tokens. These are necessary because we don't want to tax the pancakeswap contract, which is why the contract itself is excluded from fees, as stated above.

LINES 996-1040

```
    function _transfer(
        address from,
        address to,
        uint256 amount
    ) private {
        require(from != address(0), "ERC20: transfer from the zero address");
        require(to != address(0), "ERC20: transfer to the zero address");
        require(amount > 0, "Transfer amount must be greater than zero");
        if(from != owner() && to != owner())
            require(amount <= _maxTxAmount, "Transfer amount exceeds the
maxTxAmount.");

        // is the token balance of this contract address over the min number of
        // tokens that we need to initiate a swap + liquidity lock?
        // also, don't get caught in a circular liquidity event.
        // also, don't swap & liquify if sender is uniswap pair.
        uint256 contractTokenBalance = balanceOf(address(this));

        if(contractTokenBalance >= _maxTxAmount)
        {
            contractTokenBalance = _maxTxAmount;
        }

        bool overMinTokenBalance = contractTokenBalance >= numTokensSellToAddToLiquidity;

        if (
            overMinTokenBalance &&
            !inSwapAndLiquify &&
            from != uniswapV2Pair &&
            swapAndLiquifyEnabled
        ) {
            contractTokenBalance = numTokensSellToAddToLiquidity;
            //add liquidity
            swapAndLiquify(contractTokenBalance);
```

```
        }

        //indicates if fee should be deducted from transfer
        bool takeFee = true;

        //if any account belongs to _isExcludedFromFee account then remove the fee
        if(_isExcludedFromFee[from] || _isExcludedFromFee[to]){
            takeFee = false;
        }

        //transfer amount, it will take tax, burn, liquidity fee
        _tokenTransfer(from,to,amount,takeFee);
    }
```

Now this is where the magic happens. This is the function that allows tokens to be traded. Let's analyze it.

The first thing this function does is checking if in the contract are stored more than `numTokensSellToAddToLiquidity` number of tokens. This variable is set to 500000 * 10^6 as stated before.
If this and another bunch of checks necessary to prevent circular swapping are true then that amount of tokens gets added to the liquidity pool. The function `swapAndLiquify(contractTokenBalance)` is what makes this possible, but i'm not going into details on how that happens. After this happens the function `_tokenTransfer()` is called.

LINES 1041-1098
All these functions are what makes it possible to add liquidity to the pool with the tokens stored in the contract. As stated above I'm not going into details on how this exactly happens but I'll explain it very quickly.
The contract sells half of the tokens stored for BNB using the liquidity pool itself, after this is done both the BNB and the tokens are added to the pool. The contract receives then LPs tokens. Now what we have to be careful about is the following: **we have to make sure there is no way for anybody to get the LPs out of there, which in turn would mean that LPs are locked. This would grant liquidity forever.**

We looked at the whole code and we found nothing that allows that, which means we are safe.

LINES 1099-1151
```
    function _tokenTransfer(address sender, address recipient, uint256 amount,bool takeFee) private {
        if(!takeFee)
            removeAllFee();

        if (_isExcluded[sender] && !_isExcluded[recipient]) {
            _transferFromExcluded(sender, recipient, amount);
```

```
        } else if (!_isExcluded[sender] && _isExcluded[recipient]) {
            _transferToExcluded(sender, recipient, amount);
        } else if (!_isExcluded[sender] && !_isExcluded[recipient]) {
            _transferStandard(sender, recipient, amount);
        } else if (_isExcluded[sender] && _isExcluded[recipient]) {
            _transferBothExcluded(sender, recipient, amount);
        } else {
            _transferStandard(sender, recipient, amount);
        }

        if(!takeFee)
            restoreAllFee();
    }
```

This is the actual function that transfers the tokens. It's easy to notice that all of this does is remove the fees if necessary. After that it calls 1 of 4 functions based on 2 facts: if the sender and the receiver are excluded or not from the fees. All of these 4 functions are very similar, here we are only going to examine the standard one because it's the most common.

```
    function _transferStandard(address sender, address recipient, uint256 tAmount)
private {
        (uint256 rAmount, uint256 rTransferAmount, uint256 rFee, uint256
tTransferAmount, uint256 tFee, uint256 tLiquidity) = _getValues(tAmount);
        _rOwned[sender] = _rOwned[sender].sub(rAmount);
        _rOwned[recipient] = _rOwned[recipient].add(rTransferAmount);
        _takeLiquidity(tLiquidity);
        _reflectFee(rFee, tFee);
        emit Transfer(sender, recipient, tTransferAmount);
    }
```

The first thing this function does is calling `_getValues(tAmount)` this returns the amount of fees and the amount to transfer both in tokens and reflections. After that the balances are adjusted. Then the fees are distributed to the liquidity pool and the token holders.

The code ends here. There is nothing shady going on in the code, except for the fact that it is written in a complex way and I'm quite sure the same moon developers smoke a bunch of crack before doing that.

This does not mean the token can't be rugpulled.

LIQUIDITY CHECK

The risk here is that if anybody owns a high percentage of the liquidity pool they can take it all out at the same time effectively preventing people from trading at all.

We can find a list of LPs holders **here.**

At the moment of writing 82.1405% of the LPs are owned by the 0x00 contract, as we know this is not accessible. Which means the majority of LPs are locked. Good.

The other 17.85% is owned by a contract (about 16%) and a person (about 1.8%). I have no clue what that contract is about, but even if both the person and the contract sells all of their LPs the token will still be tradable.

HOLDERS CHECK

Last thing to check is how much of the coins the top holders own. If it's too much in the hands of few the risk of a coordinated sell is too high to be trusted.
We can find the list of holders **here.**

The address 0xdead000000000000000042069420694206942069 contains 60.0146% of the tokens. These tokens are burned because this address is clearly not accessible.
Now keep in mind one thing: burning tokens is not actually a good thing.

Let me explain.
The second address owns 4.9491% of the supply. But consider this: 60.01% are burned, which actually means the second address owns actually more than 4.94% of the circulating supply, how much? We can find that using a simple proportion.

100:(100%-60.01%)=X:4.9491%

X= (4.9491%*100)/(100%-60.01%) = 12.37%

So the second biggest holder actually owns **12.37%** of the circulating supply. That's a lot, but not too much.

How much do the top 20 holders own?
I did the math for you: **54,38%**.
That's a lot, but not too much especially if the holders don't know each other.

There is no actual way to check if the addresses owning a token are related to each other or owned by the same individual. What we can do instead is take a look at the transactions that happened right after the contract was deployed. We can do that **here** by going to the last page.

As we can see the first 2 transactions after the contract was deployed are the following:

- Half the supply is burned
- Half the supply is locked in the liquidity pool

That's all we know.

ANALYSIS BY @AZEPHIAR

Who am I and why should you trust me?
I'm not going to reveal my identity and I'm not interested in anybody trusting me.

Why did I write this?
I wrote this thing in 2 hours, Why did I do it? I analyzed the contract by myself because I decided to invest into it. Since I'm invested in it I want the project to be as successful as possible. I'm in no way affiliated with the creators of the project. This file it's far from perfect. If you have any questions you can reach out to me.

Contacts

TG: @Azephiar