ATLANTIS contract overview

Audited by Mars

DISCLAIMER Contract was audited by experienced developers, we aren't giving any guarantees but a detailed process of investigation, please DYOR and immediately contact us on this email crater@marscrater.com in case of an error or misunderstanding from our side

---

- AtlantisBSC
- 0xE1E3511235837Db3fa20A4a8C5bD76727f702231
- https://bscscan.com/address/0xE1E3511235837Db3fa20A4a8C5bD76727f702231

Contract Atlantis was compiled under solidity v0.8.4, with optimization (200 runs). No bugs or warnings were emitted.

Code is a heavily modified version of the Safemoon contract, it has different antiwhale mechanisms, altered liquidity acquisition, and standard reward reflection.

We start our investigation with an antiwhale mechanism, which seems to be unique:

```
if (nukeTheWhales) {
            if(from != owner() && to != owner()) {
                require(amount <= (totalSupply()) * (1) / (10**3), "Transfer amount exceeds the 0.1% of the supply.");
            }
            if(to == address(uniswapV2Pair) || to == address(uniswapV2Router)) {

            uint256 fromBalance = balanceOf(from);
            uint256 threshold = (totalSupply()) * (5) / (10**3);

                if (fromBalance > threshold) {
                    uint _now = block.timestamp;
                    require(amount < fromBalance / (5), "For your protection, max sell is 20% if you hold 0.5% or more of supply.");
                    require( _now - (previousSale[from]) > 1 days, "You must wait a full 24 hours before you may sell again.");
                }
            }
        }
```

This section of code can be found in the _transfer() function. Logic is the following: If a transfer is not from or to the owner’s address, then require() check will revert if the transfer amount exceeds 0.1% of the total supply, same as already seen maxTxAmount mechanism.

Then we have another IF statement, if the transfer is going to pancakeswap pair or router addresses, then there is a threshold, that is total supply multiplied by 5 and divided by 1000, meaning 0.5% of the total supply, if the sender address is holding more than the current threshold value, then the sender can only proceed a sell order that is lower than 20%, and also any sell order will be reverted during 24H period.

Very smart and sophisticated logic, the only flaw is “abusing” power from owner address since it’s excluded from antiwhale and there is no SafeMath used in equations that are notorious for being safe regarding underflow/overflow computational errors.

The next part is the altered liquidity acquisition mechanism, which now instead of routing fee from every transaction to the liquidity pool, is in fact taking a fee for two separate wallets - Game, and Studio.

A reasonable question can be raised, why would someone modify the most core mechanism of contract, potentially creating bugs and also losing auto liquidity acquisition thus making deflation weaker, when it is possible to use a “charity” type contract that will not touch the deflation mechanism mentioned above and still do the job by putting a certain amount of native tokens from every transaction to defined wallet.

And project owner gave me equally reasonable justification: In the case of typical charity tokens, fee’s from transactions are collected in the native currency. To be used for charity or development reasons, it should be liquidated, and liquidating a high amount of tokens can badly affect the market state. So, simply it was a better option to automatically make low (30K) liquidations with the old auto liquidity acquisition mechanism. To explain this even profoundly, first, we should review the original code and then the altered one, so investors with poor or no knowledge in the technical side can understand where is their money going, bear with us, it will be long:

Auto liquidity acquisition mechanism, originally employed by Safemoon was a very smart and sophisticated way to boost token deflation.

From transactions, a certain % is taken as a liquidity fee, and it is reserved in the contract balance. After the balance reaches some amount, defined by the owner (minTokensBeforeSwap), several events will be triggered: the total amount will be split into two, exactly one part stays in native token and another half will be converted into wrapped BNB, then they both are sent to liquidity pool.

```
function swapAndLiquify(uint256 contractTokenBalance) private lockTheSwap {
        // split the contract balance into halves
        uint256 half = contractTokenBalance.div(2);
        uint256 otherHalf = contractTokenBalance.sub(half);

        // capture the contract's current ETH balance.
        // this is so that we can capture exactly the amount of ETH that the
        // swap creates, and not make the liquidity event include any ETH that
        // has been manually sent to the contract
        uint256 initialBalance = address(this).balance;

        // swap tokens for ETH
        swapTokensForEth(half); // <- this breaks the ETH -> HATE swap when swap+liquify is triggered

        // how much ETH did we just swap into?
        uint256 newBalance = address(this).balance.sub(initialBalance);

        // add liquidity to uniswap
        addLiquidity(otherHalf, newBalance);

        emit SwapAndLiquify(half, newBalance, otherHalf);
    }
```

This is the implementation of the logic explained above.

Another question can be raised, who is paying for all the transactions that should happen after the contract balance triggers the swapAndLiquify function.

The answer is - The trader who initiates the sell order if the part taken from his/her transaction will fill up the contract balance to the point where swapAndLiquify should get triggered.

In this case of Atlantis, this part is heavily modified so, instead of liquidity acquisition, the two transfers are happening - the BNB part is sent to Studio wallet, AKA treasury wallet, which will be used for future deflations, marketing plans, etc. And another part stays in native currency and is sent to Game wallet, which will be used as an in-game currency. To finally answer the question raised before we went to the rabbit hole, instead of making big liquidations manually, this system will assure low and stabilized liquidations, which will have a less severe effect on price and market movement.

```
function swapAndLiquify(uint256 contractTokenBalance) private lockTheSwap {
        // split the contract balance into halves
        uint256 half = contractTokenBalance.div(2);
        uint256 otherHalf = contractTokenBalance.sub(half);

        // capture the contract's current ETH balance.
        // this is so that we can capture exactly the amount of ETH that the
        // swap creates, and not make the liquidity event include any ETH that
        // has been manually sent to the contract
        uint256 initialBalance = address(this).balance;

        // swap tokens for ETH
        swapTokensForEth(half); // <- this breaks the BNB -> A swap when swap+liquify is triggered

        // how much ETH did we just swap into?
        uint256 newBalance = address(this).balance.sub(initialBalance);

        payable(_studioWallet).transfer(newBalance);

        removeAllFee();
        (uint256 rAmount, uint256 rTransferAmount,, uint256 tTransferAmount,,) = _getValues(otherHalf, address(0));
        _rOwned[address(this)] = _rOwned[address(this)].sub(rAmount);
        _rOwned[_gameWallet] = _rOwned[_gameWallet].add(rTransferAmount);
        emit Transfer(address(this), _gameWallet, tTransferAmount);
        restoreAllFee();

        emit SwapAndLiquify(half, newBalance, otherHalf);
    }
```

This is the part, taken from the Atlantis contract that employs an altered liquidation mechanism. Everything is the same but the liquification is changed to standard reflection.

```
_rOwned[address(this)] = _rOwned[address(this)].sub(rAmount);
_rOwned[_gameWallet] = _rOwned[_gameWallet].add(rTransferAmount);
```

otherHalf (rAmount) that is exactly 50% from the 7% fee taken from transactions deducted from contract balance, then assigned to _gameWallet.

```
 payable(_studioWallet).transfer(newBalance);
```

While BNB part is transferred to _studioWallet with standard transfer function from ERC20.

Other features and addons:

```
function setTaxFee(uint8 _newTaxFee) external onlyOwner {
        require(_newTaxFee <= 5, "TOO_MUCH");
        _taxFee = _newTaxFee;
    }

    function setLiquidityFee(uint8 _newLiquidityFee) external onlyOwner {
        require(_newLiquidityFee <= 10, "TOO_MUCH");
        _liquidityFee = _newLiquidityFee;
    }
```

Both altered liquidity and tax fees are adjustable by the owner, but they are capped by require() check (5% and 10%), assuring that owner can't set abusive values.

```
function setPancakeRouter(address _router) public onlyOwner {
        _pancakeRouter = _router;
    }
```

Pancake router address is adjustable as well, which can come in handy in case of pancake upgrading to a newer version, but can be used to completely block transactions.

There is also a flaw, detected by our audit:

```
function includeInReward(address account) external onlyOwner() {
        require(_isExcluded[account], "Account is already excluded");
        for (uint256 i = 0; i < _excluded.length; i++) {
            if (_excluded[i] == account) {
                _excluded[i] = _excluded[_excluded.length - 1];
                _tOwned[account] = 0;
                _isExcluded[account] = false;
                _excluded.pop();
                break;
            }
        }
    }
```

In this function (and in several other functions), there is a for loop that will traverse through _excluded, this operation might get reverted with OUT_OF_GAS error if the list too long. But, as the owner assured us, this list will contain only several addresses which will not harm the contract.

To conclude, the contract contains minor security issues but remember, always DYOR! Nothing is guaranteed.

APPROVED

BY MARSCRATER

Check out Mars Craters

Website

Telegram

We also accept donations, BEP20 only 0xb4Cf0a7dCB90Fe975C1d2F9716F88CB28648540e