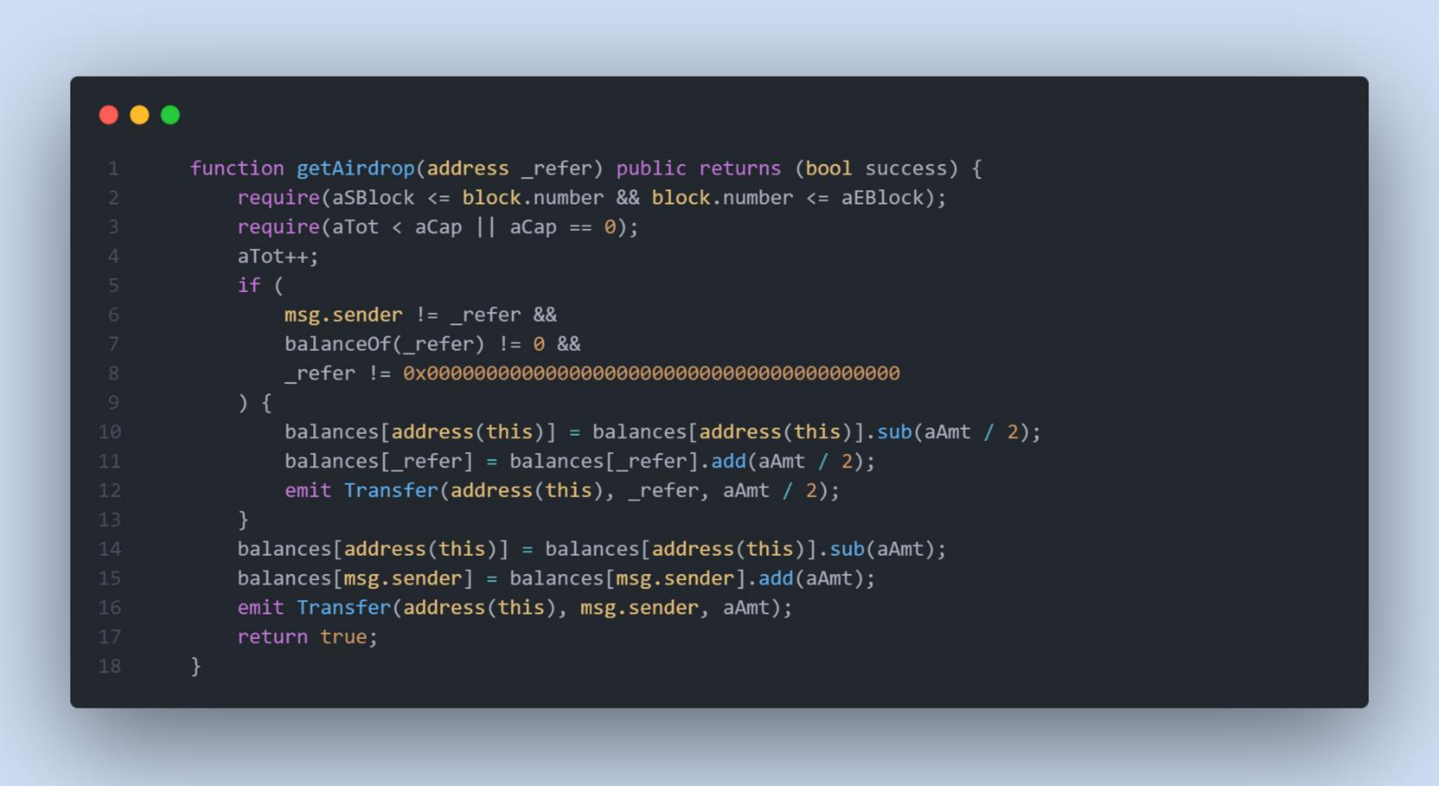
Original audit

https://www.certik.com/projects/spy

Medium severity issues

- ## MISSING UPDATE ATOT AFTER SUCCESSFUL TRANSACTION

In the functions getAirdrop, we are using aTot to make the comparison to aCap at the start of the function but there is not any change in aTot after a successful airdrop transaction.



```
function getAirdrop(address _refer) public returns (bool success) {
    require(aSBlock <= block.number && block.number <= aEBlock);
    require(aTot < aCap || aCap == 0);
    aTot++;
    if (
        msg.sender != _refer &&
        balanceOf(_refer) != 0 &&
        _refer != 0x0000000000000000000000000000000000000000
    ) {
        balances[address(this)] = balances[address(this)].sub(aAmt / 2);
        balances[_refer] = balances[_refer].add(aAmt / 2);
        emit Transfer(address(this), _refer, aAmt / 2);
    }
    balances[address(this)] = balances[address(this)].sub(aAmt);
    balances[msg.sender] = balances[msg.sender].add(aAmt);
    emit Transfer(address(this), msg.sender, aAmt);
    return true;
}
```

Remediation

Similarly, to the tokenSale function (we make update in sTot), Update aTot (airdrop total amount) after any successful airdrop transaction.

- ## DIVISION BEFORE MULTIPLICATION

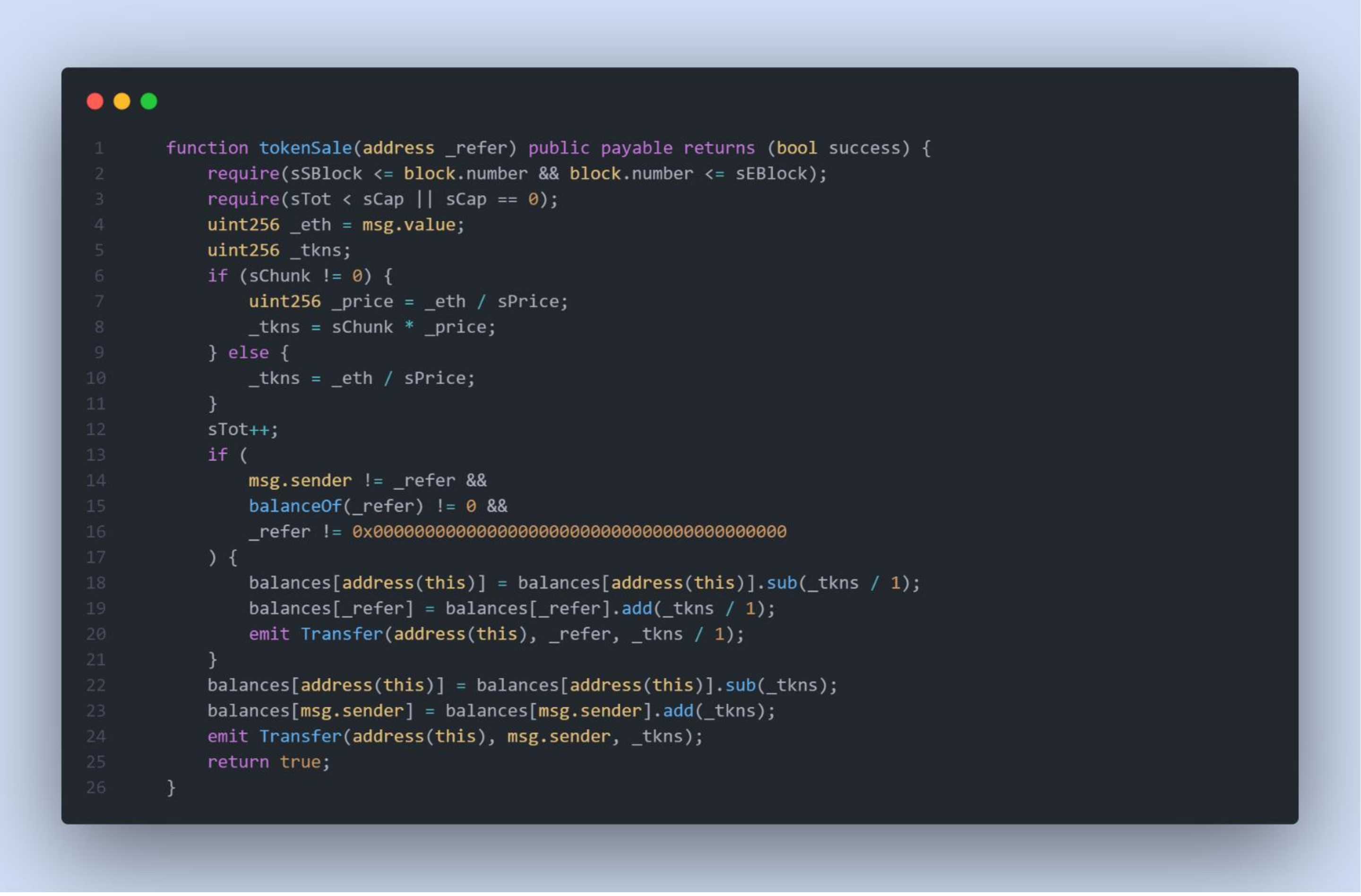
In the function tokenSale and line 237, dividing is happening before multiplying. Let's have a look at a simple mathematical example of this:

A= (10*30*18)/ 30= 180

We are now solving the same equation but performing division before multiplication.

A= (10/30)* 30*18= 179.99999

Almost these two terms are quite close but not the same. In the case of a solidity smart contract, it is actually going to yield 179. Therefore, performing multiplication before division mitigates rounding-off error in a smart contract.



```
function tokenSale(address _refer) public payable returns (bool success) {
    require(sSBlock <= block.number && block.number <= sEBlock);
    require(sTot < sCap || sCap == 0);
    uint256 _eth = msg.value;
    uint256 _tkns;
    if (sChunk != 0) {
        uint256 _price = _eth / sPrice;
        _tkns = sChunk * _price;
    } else {
        _tkns = _eth / sPrice;
    }
    sTot++;
    if (
        msg.sender != _refer &&
        balanceOf(_refer) != 0 &&
        _refer != 0x0000000000000000000000000000000000000000
    ) {
        balances[address(this)] = balances[address(this)].sub(_tkns / 1);
        balances[_refer] = balances[_refer].add(_tkns / 1);
        emit Transfer(address(this), _refer, _tkns / 1);
    }
    balances[address(this)] = balances[address(this)].sub(_tkns);
    balances[msg.sender] = balances[msg.sender].add(_tkns);
    emit Transfer(address(this), msg.sender, _tkns);
    return true;
}
```

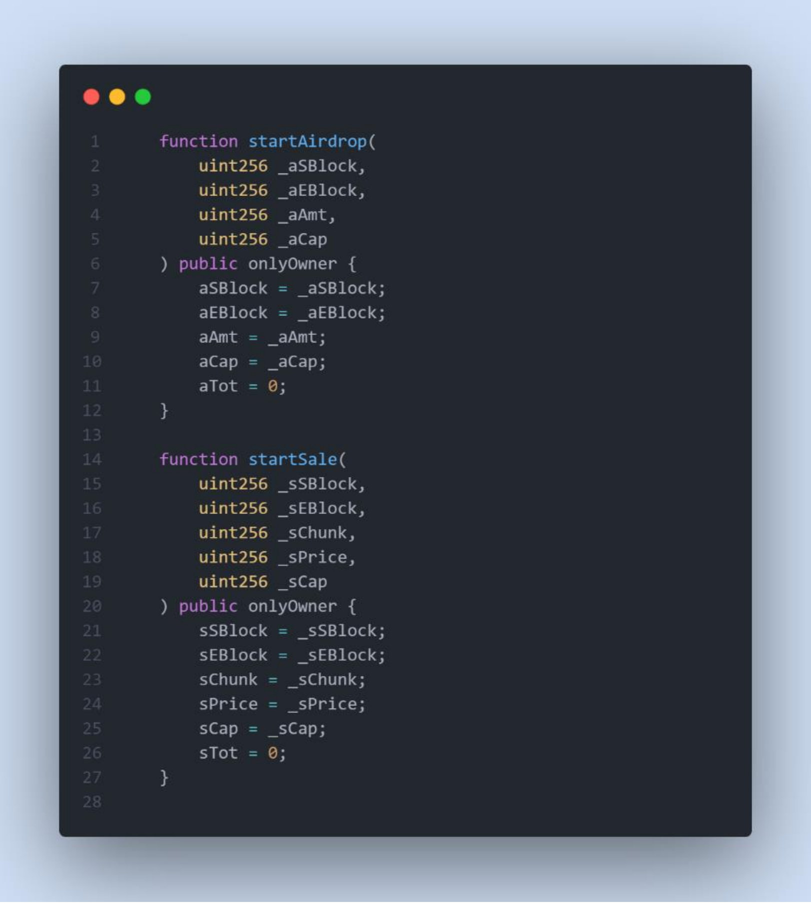
Remediation
Always do your multiplication before division!

- ## VALUE VALIDATION CHECK

In the functions shown below, the user updates variables related to user transactions. We need to first check the argument in the function and then update the variable.

taxFeeUpdate(uint256 amount);

burnFeeUpdate(uint256 amount);



```
function startAirdrop(
    uint256 _aSBlock,
    uint256 _aEBlock,
    uint256 _aAmt,
    uint256 _aCap
) public onlyOwner {
    aSBlock = _aSBlock;
    aEBlock = _aEBlock;
    aAmt = _aAmt;
    aCap = _aCap;
    aTot = 0;
}

function startSale(
    uint256 _sSBlock,
    uint256 _sEBlock,
    uint256 _sChunk,
    uint256 _sPrice,
    uint256 _sCap
) public onlyOwner {
    sSBlock = _sSBlock;
    sEBlock = _sEBlock;
    sChunk = _sChunk;
    sPrice = _sPrice;
    sCap = _sCap;
    sTot = 0;
}
```

Remediation

We need to check the argument in the function before making any changes to the contract. The argument should not be 0 or it should not be more than the maximum.

require(_ amount >= maximumAmountAllowed , “message”);

require(_ amount <= minimumAmountAllowed , “message”);

Low severity issues

VARIABLES THAT COULD BE DECLARED AS IMMUTABLE

In function startAirdrop and startSale, the owner is initializing some variables, these variables should not get changed in the future. By making the change in these variables, centralization risk can happen.

Remediation

We advise using the constructor function and adding these variables as immutable in the contract. Immutable state variables can be assigned during contract creation but will remain constant throughout the lifetime of a deployed contract. A big advantage of immutable variables is that reading them is significantly cheaper than reading from regular state variables since they will not be stored in storage.

MISSING ERROR MESSAGES

The require can be used to check for conditions and throw an exception if the condition is not met.

Remediation

We advise adding error messages to the linked require statements.

IMPROPER USAGE OF PUBLIC AND EXTERNAL TYPE

public functions that are never called by the contract could be declared as external. external functions are more efficient than public functions.

Remediation

Consider using the external attribute for public functions that are never called within the contract.