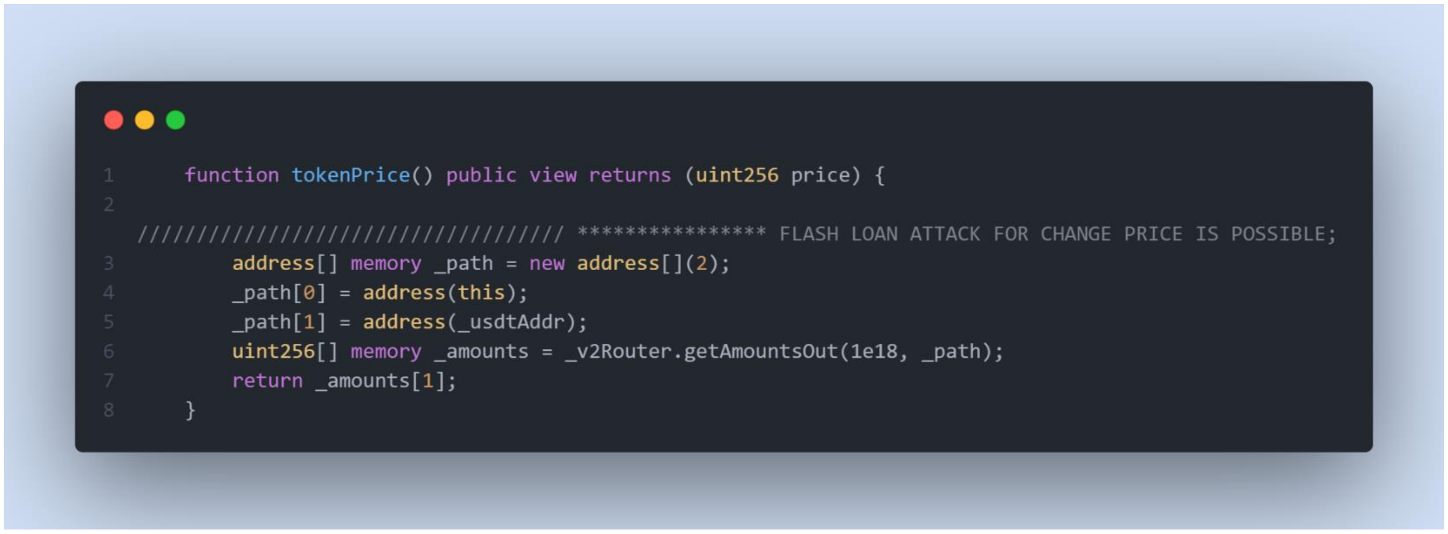
MDC.sol

High severity issues

- **Get token price**

In the function tokenPrice, we are using the DEX contract to calculate the token price and use this price in the function upgradeAmount. The attacker can use the flash loan and manipulate the price of the token.



```
    function tokenPrice() public view returns (uint256 price) {

////////////////////////////////////// *************** FLASH LOAN ATTACK FOR CHANGE PRICE IS POSSIBLE;
        address[] memory _path = new address[](2);
        _path[0] = address(this);
        _path[1] = address(_usdtAddr);
        uint256[] memory _amounts = _v2Router.getAmountsOut(1e18, _path);
        return _amounts[1];
    }
```

✓ **Recommendation**

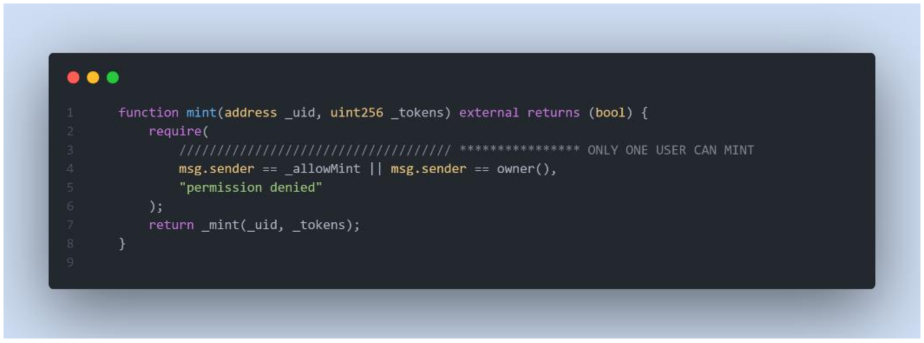
Use oracle service to get the price.

Medium severity issues

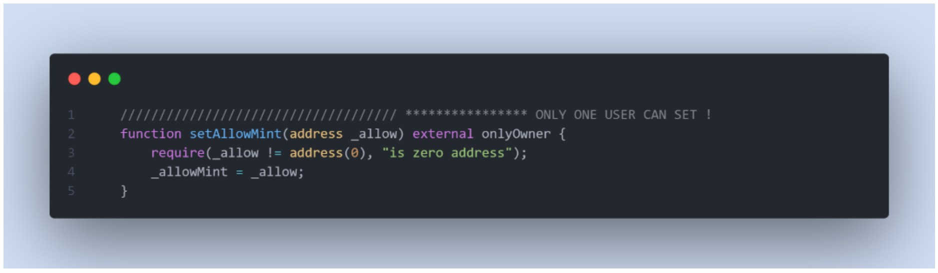
- ### MINTER_ROLE / Centralization

In the function mint, owner or role _allowMint, can mint tokens and distribute those tokens without obtaining the consensus of the community, this could be a centralization risk.

In the function setAllowint, the owner role can set any address as role _ allowMint. allowMint, can mint tokens and distribute those tokens without obtaining the consensus of the community, this could be a centralization risk.



```
    function mint(address _uid, uint256 _tokens) external returns (bool) {
        require(
            //////////////////////////////////// **************** ONLY ONE USER CAN MINT
            msg.sender == _allowMint || msg.sender == owner(),
            "permission denied"
        );
        return _mint(_uid, _tokens);
    }
```



```
    //////////////////////////////////// **************** ONLY ONE USER CAN SET !
    function setAllowMint(address _allow) external onlyOwner {
        require(_allow != address(0), "is zero address");
        _allowMint = _allow;
    }
```

The owner role has access to the below functions and can make changes to contract storage variables. These variables are related to user actions.

setV2Pair, unsetV2Pair, setAllowMint, setSwaptime, setExcluded, setMaxDao

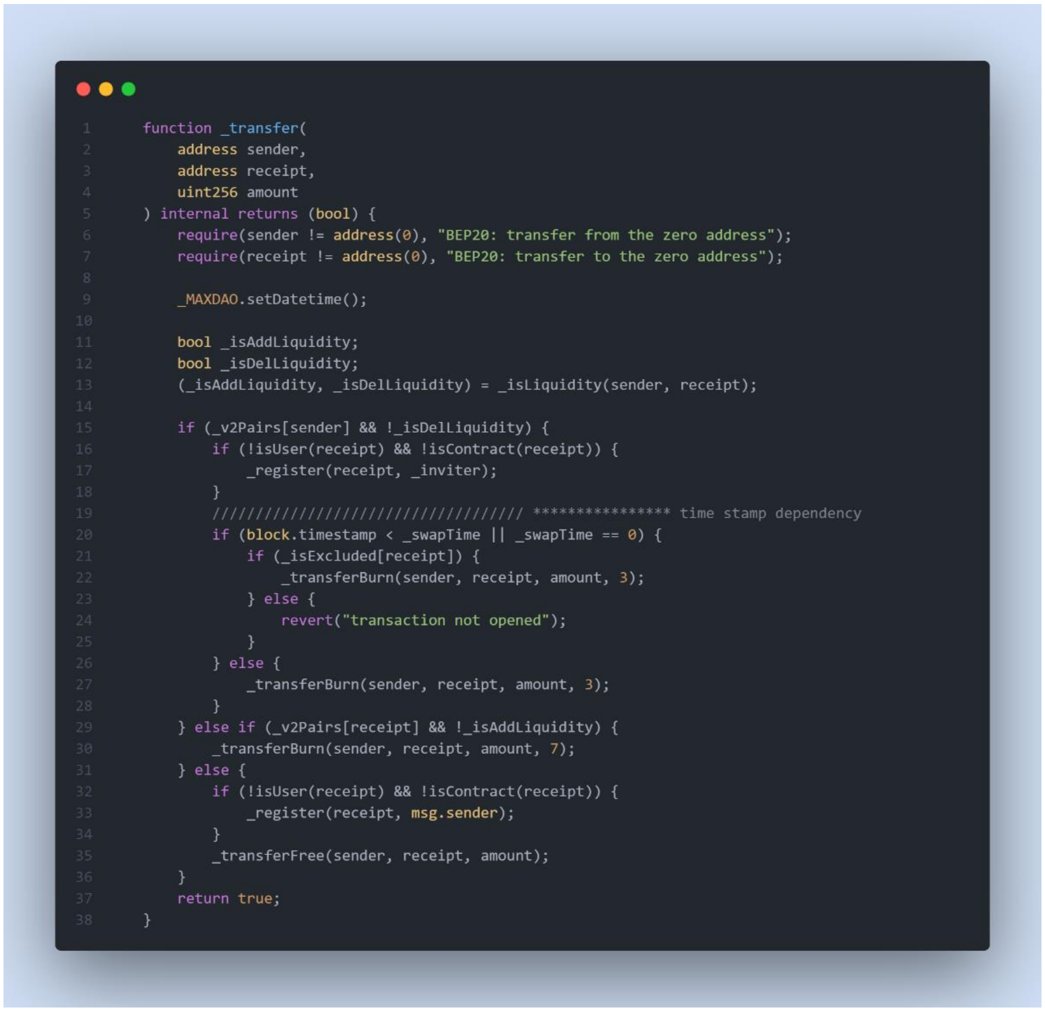
- ✓ **Recommendation**

The risk describes the current project design and potentially makes iterations to improve the security operation and level of decentralization, which in most cases cannot be resolved entirely at the present stage. We advise the client to carefully manage the privileged account's private key to avoid any potential risks of being hacked. In general, we strongly recommend centralized privileges or roles in the protocol be improved via a decentralized mechanism or smart-contract-based accounts with enhanced security practices, e.g., multi-signature wallets

Low severity issues

- **Timestamp Dependency**

There is a Timestamp Dependence problem in _transfer. The timestamp of a block, accessed by now or block.timestamp can be manipulated by a miner.



```
function _transfer(
    address sender,
    address receipt,
    uint256 amount
) internal returns (bool) {
    require(sender != address(0), "BEP20: transfer from the zero address");
    require(receipt != address(0), "BEP20: transfer to the zero address");

    _MAXDAO.setDatetime();

    bool _isAddLiquidity;
    bool _isDelLiquidity;
    (_isAddLiquidity, _isDelLiquidity) = _isLiquidity(sender, receipt);

    if (_v2Pairs[sender] && !_isDelLiquidity) {
        if (!isUser(receipt) && !isContract(receipt)) {
            _register(receipt, _inviter);
        }
        //////////////////////////////////// **************** time stamp dependency
        if (block.timestamp < _swapTime || _swapTime == 0) {
            if (_isExcluded[receipt]) {
                _transferBurn(sender, receipt, amount, 3);
            } else {
                revert("transaction not opened");
            }
        } else {
            _transferBurn(sender, receipt, amount, 3);
        }
    } else if (_v2Pairs[receipt] && !_isAddLiquidity) {
        _transferBurn(sender, receipt, amount, 7);
    } else {
        if (!isUser(receipt) && !isContract(receipt)) {
            _register(receipt, msg.sender);
        }
        _transferFree(sender, receipt, amount);
    }
    return true;
}
```

- ✓ **Recommendation**

Use oracle service to get the time.

• Missing Emit Events

There should always be events emitted in the sensitive functions.

- function mint(address _uid, uint256 _tokens) external returns (bool);
- function transfer(address recipient, uint256 amount);
- …

✓ Recommendation

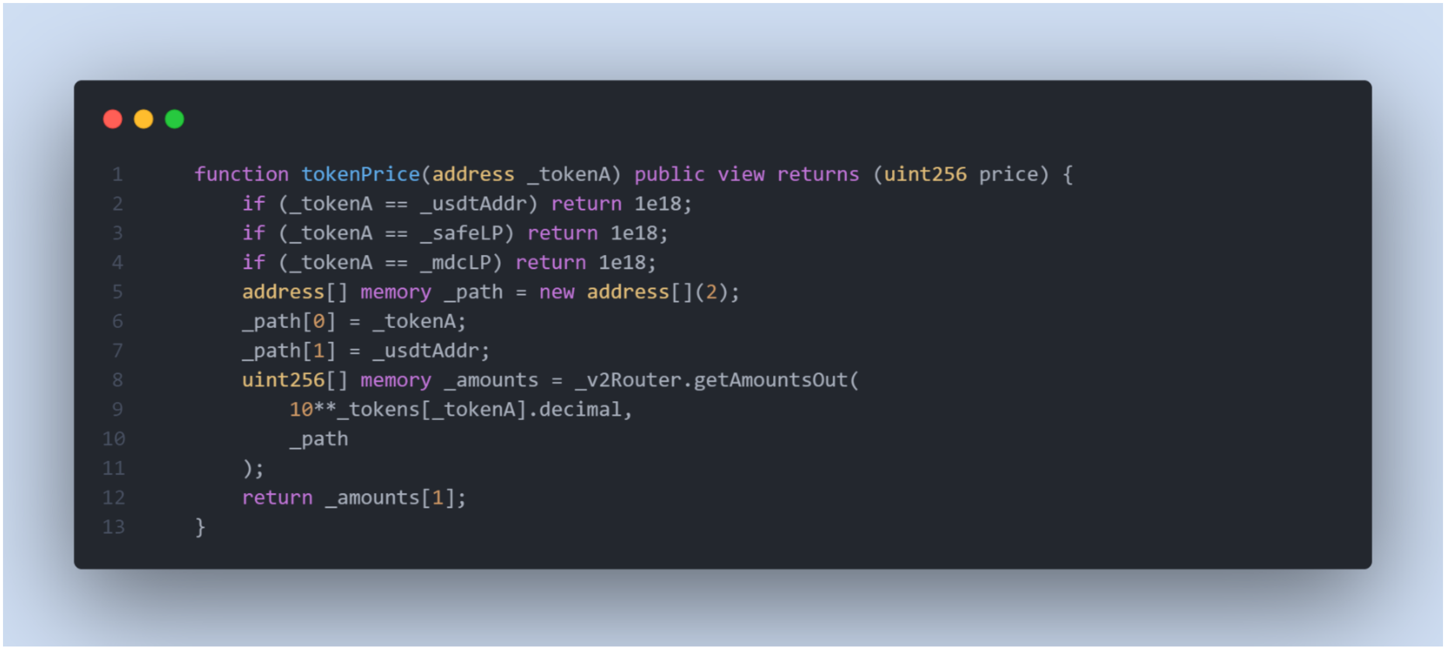
It is recommended emitting events for sensitive functions.

MaxDAO.sol

High severity issues

- **Get token price**

In the function tokenPrice, we are using the DEX contract to calculate the token price and use this price in the function related to deposit in platform. The attacker can use the flash loan and manipulate the price of the token.



```
function tokenPrice(address _tokenA) public view returns (uint256 price) {
    if (_tokenA == _usdtAddr) return 1e18;
    if (_tokenA == _safeLP) return 1e18;
    if (_tokenA == _mdcLP) return 1e18;
    address[] memory _path = new address[](2);
    _path[0] = _tokenA;
    _path[1] = _usdtAddr;
    uint256[] memory _amounts = _v2Router.getAmountsOut(
        10**_tokens[_tokenA].decimal,
        _path
    );
    return _amounts[1];
}
```

✓ **Recommendation**

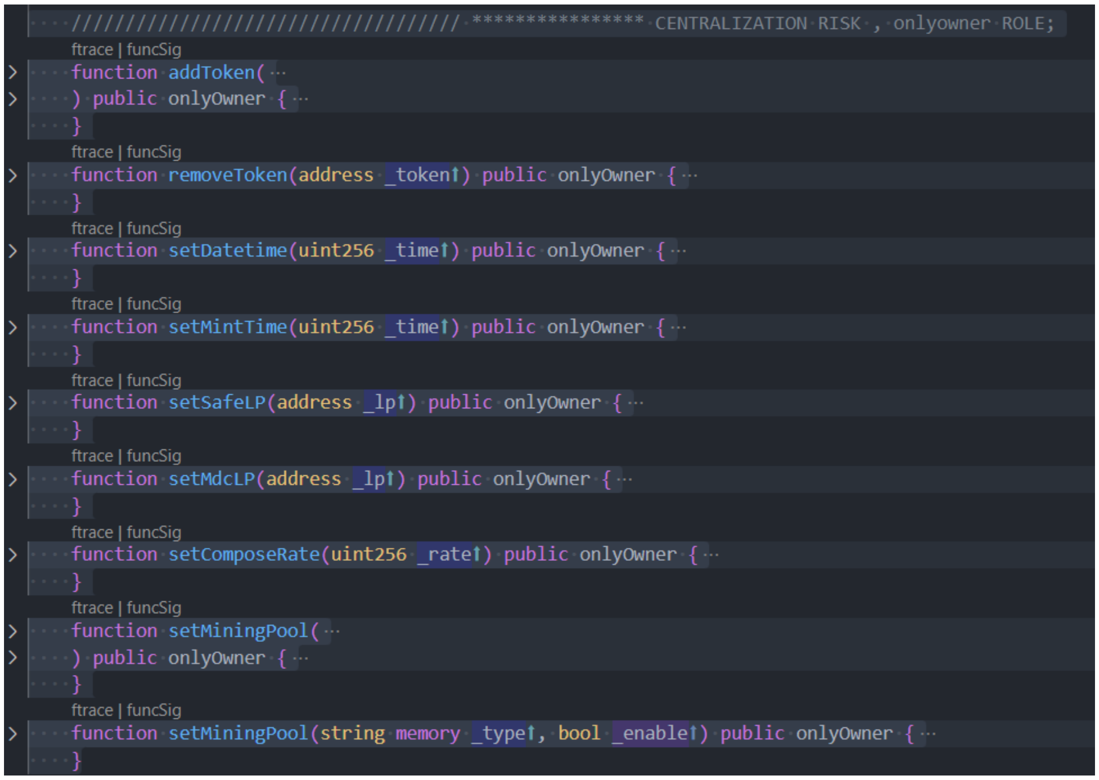
Use oracle service to get the price.

Medium severity issues

- ### OWNER_ROLE / Centralization

In the functions listed below, the only owner role has access to the functions and can make changes to contract storage variables. These variables are related to user actions.

For example, in one scenario, the owner can add a token contract containing a Malicious code or the owner can remove the token from the contract without the users' permission.



```
    /////////////////////////////////// *************** CENTRALIZATION RISK , onlyowner ROLE;
    ftrace | funcSig
    function addToken(…
    ) public onlyOwner {…
    }
    ftrace | funcSig
    function removeToken(address _token↑) public onlyOwner {…
    }
    ftrace | funcSig
    function setDatetime(uint256 _time↑) public onlyOwner {…
    }
    ftrace | funcSig
    function setMintTime(uint256 _time↑) public onlyOwner {…
    }
    ftrace | funcSig
    function setSafeLP(address _lp↑) public onlyOwner {…
    }
    ftrace | funcSig
    function setMdcLP(address _lp↑) public onlyOwner {…
    }
    ftrace | funcSig
    function setComposeRate(uint256 _rate↑) public onlyOwner {…
    }
    ftrace | funcSig
    function setMiningPool(…
    ) public onlyOwner {…
    }
    ftrace | funcSig
    function setMiningPool(string memory _type↑, bool _enable↑) public onlyOwner {…
    }
```

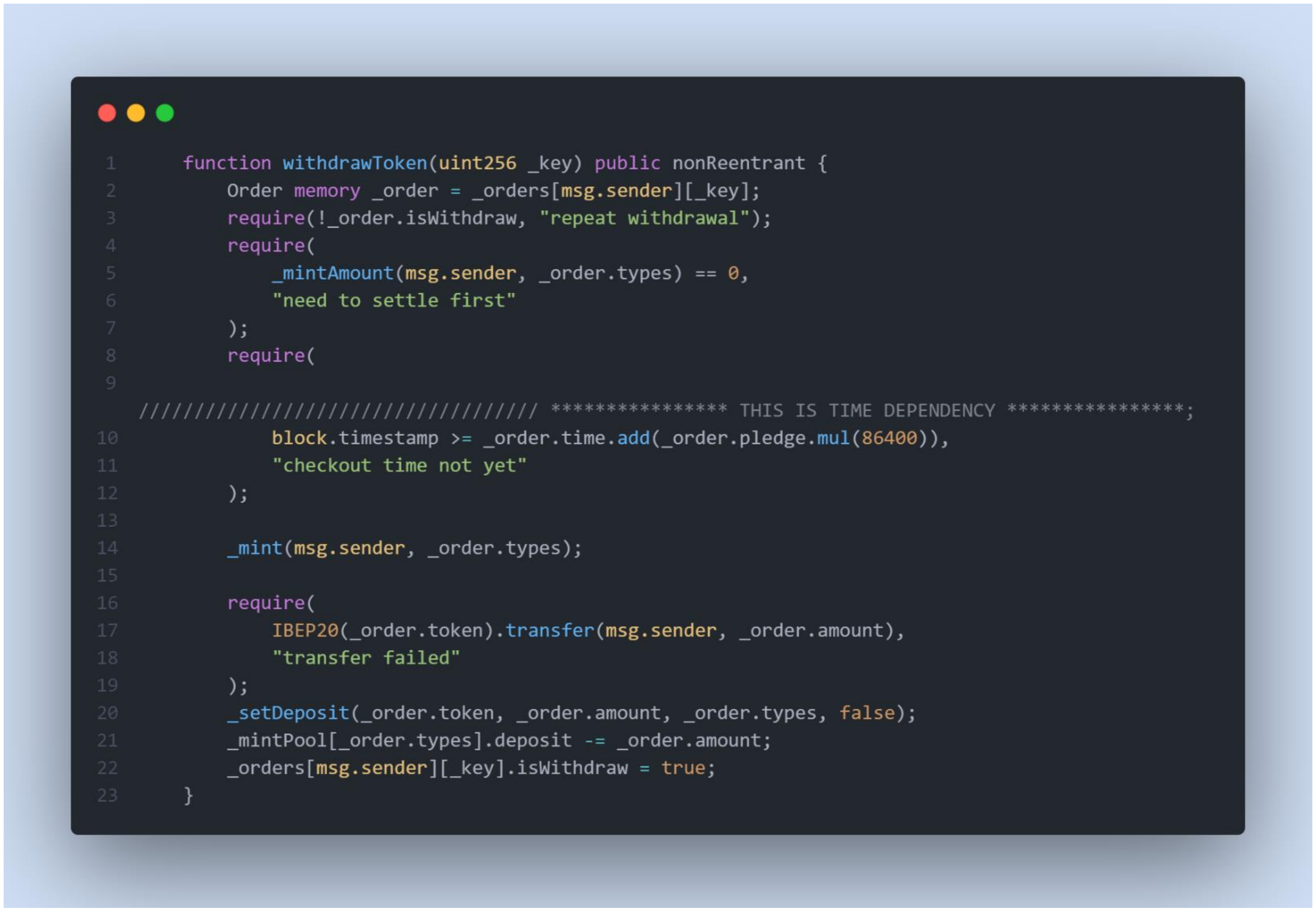
- ✓ ### Recommendation

The risk describes the current project design and potentially makes iterations to improve the security operation and level of decentralization, which in most cases cannot be resolved entirely at the present stage. We advise the client to carefully manage the privileged account's private key to avoid any potential risks of being hacked. In general, we strongly recommend centralized privileges or roles in the protocol be improved via a decentralized mechanism or smart-contract-based accounts with enhanced security practices, e.g., multi-signature wallets.

Low severity issues

- **Timestamp Dependency**

There is a Timestamp Dependence problem in withdrawToken. The timestamp of a block, accessed by now or block.timestamp can be manipulated by a miner.



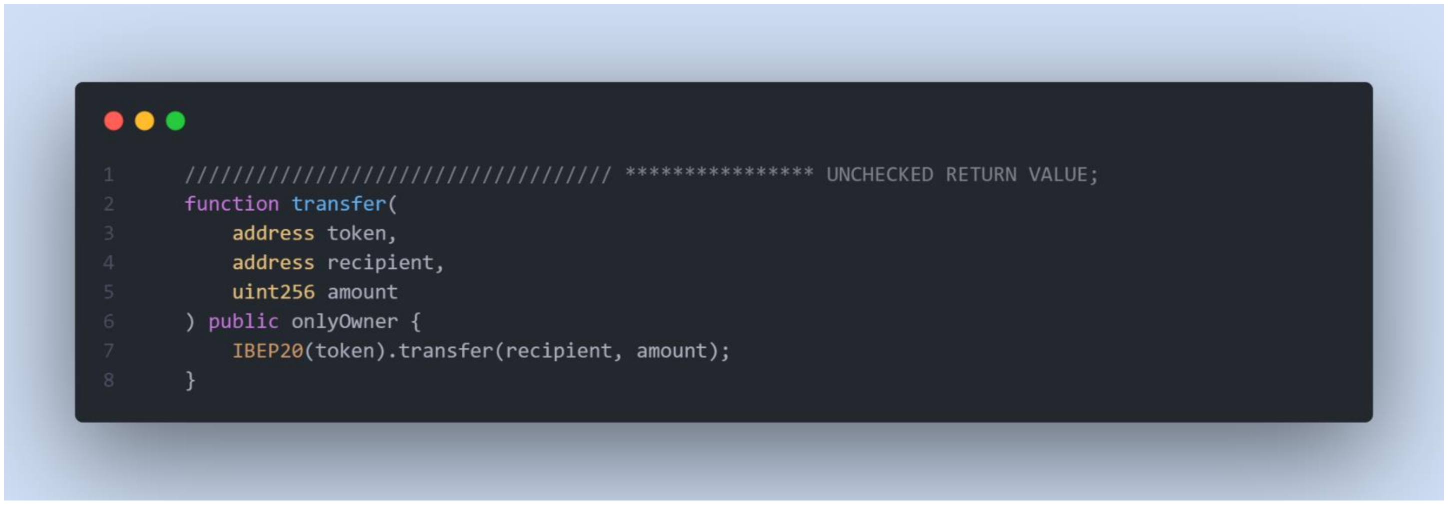
```
    function withdrawToken(uint256 _key) public nonReentrant {
        Order memory _order = _orders[msg.sender][_key];
        require(!_order.isWithdraw, "repeat withdrawal");
        require(
            _mintAmount(msg.sender, _order.types) == 0,
            "need to settle first"
        );
        require(

////////////////////////////////////// *************** THIS IS TIME DEPENDENCY ***************;
            block.timestamp >= _order.time.add(_order.pledge.mul(86400)),
            "checkout time not yet"
        );

        _mint(msg.sender, _order.types);

        require(
            IBEP20(_order.token).transfer(msg.sender, _order.amount),
            "transfer failed"
        );
        _setDeposit(_order.token, _order.amount, _order.types, false);
        _mintPool[_order.types].deposit -= _order.amount;
        _orders[msg.sender][_key].isWithdraw = true;
    }
```

✓ **Recommendation**

Use oracle service to get the time.

• Unchecked returned value

In the functions below, the contract is using the transfer function without checking returned value.



```
//////////////////////////////////// **************** UNCHECKED RETURN VALUE;
function transfer(
    address token,
    address recipient,
    uint256 amount
) public onlyOwner {
    IBEP20(token).transfer(recipient, amount);
}
```

✓ Recommendation

We recommend checking the returned value or using safeERC20library.

• Missing Emit Events

There should always be events emitted in the sensitive functions.

- function depositAlone at line 745.
- function transfer at line 720.
- function depositCompose at line 768.
- …

✓ Recommendation

It is recommended emitting events for sensitive functions.