Cyberscope

Audit Report

Dragon

March 2024

Repository https://github.com/breadNbutter42/Dragon/blob/main/DragonToken.sol

Commit ce6559c3110796c775c52960fd6d95388a5c9c0d

Audited by © cyberscope

Analysis

Critical Medium Minor / Informative Pass

Severity	Code	Description	Status
●	ST	Stops Transactions	Unresolved
●	OTUT	Transfers User's Tokens	Passed
●	ELFM	Exceeds Fees Limit	Passed
●	MT	Mints Tokens	Passed
●	BT	Burns Tokens	Passed
●	BC	Blacklists Addresses	Passed

Diagnostics

Critical Medium Minor / Informative

Severity	Code	Description	Status
●	US	Untrusted Source	Unresolved
●	DDP	Decimal Division Precision	Unresolved
●	IDI	Immutable Declaration Improvement	Unresolved
●	MVN	Misleading Variables Naming	Unresolved
●	PMRM	Potential Mocked Router Manipulation	Unresolved
●	PTRP	Potential Transfer Revert Propagation	Unresolved
●	PVC	Price Volatility Concern	Unresolved
●	RCC	Redundant Constructor Checks	Unresolved
●	RSW	Redundant Storage Writes	Unresolved
●	TUU	Time Units Usage	Unresolved
●	L02	State Variables could be Declared Constant	Unresolved
●	L04	Conformance to Solidity Naming Conventions	Unresolved
●	L09	Dead Code Elimination	Unresolved
●	L11	Unnecessary Boolean equality	Unresolved

●	L13	Divide before Multiply Operation	Unresolved
●	L14	Uninitialized Variables in Local Scope	Unresolved
●	L15	Local Scope Variable Shadowing	Unresolved
●	L16	Validate Variable Setters	Unresolved
●	L17	Usage of Solidity Assembly	Unresolved
●	L19	Stable Compiler Version	Unresolved
●	L20	Succeeded Transfer Check	Unresolved

Table of Contents

Review

Repository	https://github.com/breadNbutter42/Dragon/blob/main/DragonToken.sol
Commit	ce6559c3110796c775c52960fd6d95388a5c9c0d
Badge Eligibility	Must Fix Criticals

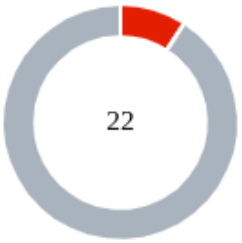
Audit Updates

Initial Audit	06 Mar 2024

Source Files

Filename	SHA256
DragonToken.sol	5b7e80dcc162de4b8f47429b34084a572b72eb017e18a14a482fd11e47a0fb52

Findings Breakdown



Critical	2
Medium	0
Minor / Informative	20

Severity	Unresolved	Acknowledged	Resolved	Other
Critical	2	0	0	0
Medium	0	0	0	0
Minor / Informative	20	0	0	0

ST - Stops Transactions

Criticality	Critical
Location	DragonToken.sol#L5348,5605
Status	Unresolved

Description

The transactions are initially disabled for all users excluding the authorized addresses. The owner can enable the transactions for all users. Once the transactions are enable the owner will not be able to disable them again.

```
    function beforeTokenTransfer(
        address from_,
        address to_,
        uint256 amount_
    ) private {

        if (phasesInitialized == false) {
            //If phases not done being initialized then only allow owner
transfers or LP creation
            require(from_ == owner() || to_ == owner() || to_ ==
uniswapV2Pair, "Buying is not yet open to the public");
            return;
        }

        uint256 tradingPhase_ = tradingPhase();

        if (tradingPhase_ == totalPhases){ //No limits on the last phase
            return;
        }

        if (to_ != uniswapV2Pair && from_ != owner() && to_ != owner()) {
//Don't limit LP creation or selling

            if (tradingPhase_ == 0) {
                revert("Trading phases have not yet started; You are too
early");
            }

            require(allowlisted[to_] <= tradingPhase_, "Not allowlisted
for current phase");
            totalPurchased[to_] += amount_; //Total amount user received
in whale limited phases
            require(totalPurchased[to_] <= maxWeiPerPhase[tradingPhase_],
"Buying too much for current whale limited phase");
        }
    }



    function setPhasesStartTime(uint256 startTime_) external onlyOwner {
        require(startTime_ > 0, "Start time must be greater than 0");
        require(phasesInitialized == false, "Phases initialization is
locked");
        startTime = startTime_;
        emit SettingsChanged(msg.sender, "setPhasesStartTime");
    }
```

Additionally, the contract owner has the authority to stop the sales for all users excluding the owner, as described in detail in sections `US` , `PTRP` and `PMRM` . As a result, the contract might operate as a honeypot.

Recommendation

The team should carefully manage the private keys of the owner's account. We strongly recommend a powerful security mechanism that will prevent a single user from accessing the contract admin functions. Some suggestions are:

- Introduce a multi-sign wallet so that many addresses will confirm the action.
- Introduce a governance model where users will vote about the actions.

US - Untrusted Source

Criticality	Critical
Location	DragonToken.sol#L5275,5512
Status	Unresolved

Description

The contract uses an external contract in order to determine the transaction's flow. The external contract is untrusted. As a result, it may produce security issues and harm the transactions.

```
   function setExternalFeesProcessor(address contract_) external
onlyOwner {  //Set to the 0 address again to revert this change
        externalFeesProcessor = contract_;
        emit SettingsChanged(msg.sender,
"setExternalFeesProcessor");
    }
...
 function processFees() public {
 ...
    externalFeesProcessing(swapToDRAGONLP_,swapToCtLP_);
    ...
    }

    function externalFeesProcessing(uint256 swapToDRAGONLP_,
uint256 swapToCtLP_) internal { //If we set an external fees
processing contract, then we call it here
        _approve(address(this), externalFeesProcessor,
(swapToDRAGONLP_ + swapToCtLP_)); //Approve external fees processor
to swap our DRAGON tokens and make LP

        (bool success_, string memory error_) =
IProcessFees(externalFeesProcessor).processFeesExternal(swapToDRAGO
NLP_, swapToCtLP_);

        require(success_, error_);
    }
```

Recommendation

The contract should use a trusted external source. A trusted source could be either a commonly recognized or an audited contract. The pointing addresses should not be able to change after the initialization.

DDP - Decimal Division Precision

Criticality	Minor / Informative
Location	DragonToken.sol#L5531
Status	Unresolved

Description

Division of decimal (fixed point) numbers can result in rounding errors due to the way that division is implemented in Solidity. Thus, it may produce issues with precise calculations with decimal numbers.

Solidity represents decimal numbers as integers, with the decimal point implied by the number of decimal places specified in the type (e.g. decimal with 18 decimal places). When a division is performed with decimal numbers, the result is also represented as an integer, with the decimal point implied by the number of decimal places in the type. This can lead to rounding errors, as the result may not be able to be accurately represented as an integer with the specified number of decimal places.

Hence, the splitted shares will not have the exact precision and some funds may not be calculated as expected.

```
uint256 swapToDRAGONLP_ = (contractDRAGONBalance_ * liquidityFee) /
totalFees; //Calculate DRAGON LP fee portion
uint256 farmTokens_ = (contractDRAGONBalance_ * farmFee) / totalFees;
//Calculate farm fee portion
uint256 treasuryTokens_ = (contractDRAGONBalance_ * treasuryFee) /
totalFees; //Calculate treasury fee portion
uint256 swapToCtLP_ = (contractDRAGONBalance_ * communityLPFee) /
totalFees; //Calculate community tokens LP fee portion
```

Recommendation

The team is advised to take into consideration the rounding results that are produced from the solidity calculations. The contract could calculate the subtraction of the divided funds in the last calculation in order to avoid the division rounding issue.

IDI - Immutable Declaration Improvement

Criticality	Minor / Informative
Location	DragonToken.sol#L5105,5107
Status	Unresolved

Description

The contract declares state variables that their value is initialized once in the constructor and are not modified afterwards. The `immutable` is a special declaration for this kind of state variables that saves gas when it is defined.

```
totalPhases
timePerPhase
```

Recommendation

By declaring a variable as immutable, the Solidity compiler is able to make certain optimizations. This can reduce the amount of storage and computation required by the contract, and make it more gas-efficient.

MVN - Misleading Variables Naming

Criticality	Minor / Informative
Location	DragonToken.sol#L5140
Status	Unresolved

Description

Variables can have misleading names if their names do not accurately reflect the value they contain or the purpose they serve. The contract uses some variable names that are too generic or do not clearly convey the information stored in the variable. Misleading variable names can lead to confusion, making the code more difficult to read and understand.

Specifically, the contract is currently using the variable name `basePair` to refer to the address `0xB31f66AA3C1e785363F0875A1B74E27b85FD66c7`. However, this address does not represent a liquidity pair or a token pair as the name `basePair` suggests. Instead, it is the address of the WAVAX token on the Avalanche network. This misalignment between the variable name and the actual entity it represents can lead to confusion and misunderstanding about the contract's functionality, especially regarding how it interacts with the Avalanche ecosystem and DeFi protocols.

```
basePair = 0xB31f66AA3C1e785363F0875A1B74E27b85FD66c7;
```

Recommendation

It's always a good practice for the contract to contain variable names that are specific and descriptive. The team is advised to keep in mind the readability of the code. It is recommended to rename the `basePair` variable to more accurately reflect the entity it represents.

PMRM - Potential Mocked Router Manipulation

Criticality	Minor / Informative
Location	DragonToken.sol#L5175
Status	Unresolved

Description

The contract includes a method that allows the owner to modify the router address and create a new pair. While this feature provides flexibility, it introduces a security threat. The owner could set the router address to any contract that implements the router's interface, potentially containing malicious code. In the event of a transaction triggering the swap functionality with such a malicious contract as the router, the transaction may be manipulated.

```
   function setMainDex(address router_, address basePair_)
external onlyOwner {
    //There is an inherent risk using a third party dex, so we
have the ability to change dexs.
        router = router_;
        basePair = basePair_;
        IUniswapV2Router02 uniswapV2Router_ =
IUniswapV2Router02(router);
        uniswapV2Router = uniswapV2Router_; //Uses the
interface created above
        address uniswapV2Pair_ =
IUniswapV2Factory(uniswapV2Router.factory()).getPair(address(th
is), basePair); //This pair must already exist
        require(uniswapV2Pair_ != address(0),
        "LP Pair must be created first, with a valid basePair_
input into the setDex function"); //An invalid LP pair or
basePair_ will return 0 address.
        uniswapV2Pair = uniswapV2Pair_; //Set to pair grabbed
from the factory call above
        uint256 length_ = communityTokens.length;

        for (uint256 i = 0; i < length_; i++){
            uniswapV2Pair_ =
IUniswapV2Factory(uniswapV2Router.factory()).getPair(address(th
is), communityTokens[i]);
            require(uniswapV2Pair_ != address(0), "All
CT/DRAGON LP Pairs must be created first on main dex, to create
LPs on");
        }

        emit SettingsChanged(msg.sender, "setMainDex");
    }
```

Recommendation

The team should carefully manage the private keys of the owner's account. We strongly recommend a powerful security mechanism that will prevent a single user from accessing the contract admin functions.

Temporary Solutions:

These measurements do not decrease the severity of the finding

- Introduce a time-locker mechanism with a reasonable delay.
- Introduce a multi-signature wallet so that many addresses will confirm the action.

- Introduce a governance model where users will vote about the actions.

Permanent Solution:

- Renouncing the ownership, which will eliminate the threats but it is non-reversible.

PTRP - Potential Transfer Revert Propagation

Criticality	Minor / Informative
Location	DragonToken.sol#L5515
Status	Unresolved

Description

The contract sends funds to a `externalFeesProcessor` as part of the transfer flow. This address can either be a wallet address or a contract. If the address belongs to a contract then it may revert from incoming payment. As a result, the error will propagate to the token's contract and revert the transfer.

```
(bool success_, string memory error_) =
IProcessFees(externalFeesProcessor).processFeesExternal(swapToDRAGO
NLP_, swapToCtLP_);

require(success_, error_);
```

Recommendation

The contract should tolerate the potential revert from the underlying contracts when the interaction is part of the main transfer flow. This could be achieved by not allowing set contract addresses or by sending the funds in a non-revertable way.

PVC - Price Volatility Concern

Criticality	Minor / Informative
Location	DragonToken.sol#L5524
Status	Unresolved

Description

The contract accumulates tokens from the taxes to swap them for ETH. The variable `swapTokensAtAmount` sets a threshold where the contract will trigger the swap functionality. However the contract will accumulate fees until the `processFees` function is called. As a result the contract will swap a huge amount of tokens for ETH.

It is important to note that the price of the token representing it, can be highly volatile. This means that the value of a price volatility swap involving Ether could fluctuate significantly at the triggered point, potentially leading to significant price volatility for the parties involved.

```
function processFees() public {  //Swap DRAGON fees for
community tokens, make into LP, and burn LP
        uint256 contractDRAGONBalance_ =
balanceOf(address(this));
        bool canSwap_ = contractDRAGONBalance_ >=
swapTokensAtAmount; //Must have the minimum amount of fees
collected to process them

        if (canSwap_ && !swapping) {
            ...
            if (externalFeesProcessor != address(0)) { //If we
have added an external fees processing contract then use that
instead

externalFeesProcessing(swapToDRAGONLP_,swapToCtLP_);
            } else {
                swapAndBurnLP(swapToDRAGONLP_, swapToCtLP_);
            }
```

Recommendation

The contract could ensure that it will not sell more than a reasonable amount of tokens in a single transaction. A suggested implementation could check that the maximum amount

should be less than a fixed percentage of the exchange reserves. Hence, the contract will guarantee that it cannot accumulate a huge amount of tokens in order to sell them.

RCC - Redundant Constructor Checks

Criticality	Minor / Informative
Location	DragonToken.sol#L5106
Status	Unresolved

Description

The contract is declaring and initializing its state variables within its constructor, including setting `totalPhases` to 8, `timePerPhase` to 480 seconds, and calculating `swapTokensAtAmount` based on the total supply. Following these initializations, the contract uses require statements to ensure that these variables are greater than 0. However, these checks are redundant given the static values assigned to `totalPhases` and `timePerPhase`, and the fact that `swapTokensAtAmount` is derived from the total supply. This redundancy not only increases the gas cost of deploying the contract but also clutters the constructor with unnecessary validations.

```
constructor(){
...
totalPhases = 8; //Last phase is the public non whale limited phase
require(totalPhases > 0, "Total phases must be greater than 0");
timePerPhase = 480; //Seconds per each phase, 8 minutes is 480 seconds
require(timePerPhase > 0, "Time per phase must be greater than 0");
...
swapTokensAtAmount = totalSupply() / (100000); //Minimum amount of
fees collected in contract before we procesFees(), so users don't
waste gas.
require(swapTokensAtAmount > 0, "swapTokensAtAmount cannot be 0 wei,
totalSupply() is too low");
...
}
```

Recommendation

It is recommended to remove the redundant `require` statements from the constructor that check if `totalPhases`, `timePerPhase`, and `swapTokensAtAmount` are

greater than 0. Given that these variables are directly assigned non-zero values within the constructor, these checks serve no practical purpose and only add to the complexity and deployment cost of the contract. Simplifying the constructor by eliminating these unnecessary validations will improve the contract's efficiency and readability, aligning with best practices for smart contract development.

RSW - Redundant Storage Writes

Criticality	Minor / Informative
Location	DragonToken.sol#L5255
Status	Unresolved

Description

The contract modifies the state of the following variables without checking if their current value is the same as the one given as an argument. As a result, the contract performs redundant storage writes, when the provided parameter matches the current state of the variables, leading to unnecessary gas consumption and inefficiencies in contract execution.

```
    function excludeFromFees(address account_) public onlyOwner
{ //Exclude address from transfer fees
        isExcludedFromFees[account_] = true;
        emit ExcludeFromFees(account_);
    }
```

Recommendation

The team is advised to implement additional checks within to prevent redundant storage writes when the provided argument matches the current state of the variables. By incorporating statements to compare the new values with the existing values before proceeding with any state modification, the contract can avoid unnecessary storage operations, thereby optimizing gas usage.

TUU - Time Units Usage

Criticality	Minor / Informative
Location	DragonToken.sol#L5107
Status	Unresolved

Description

The contract is using arbitrary numbers to form time-related values. As a result, it decreases the readability of the codebase and prevents the compiler to optimize the source code.

```
timePerPhase = 480; //Seconds per each phase, 8 minutes is 480 seconds
```

Recommendation

It is a good practice to use the time units reserved keywords like `seconds` , `minutes` , `hours` , `days` and `weeks` to process time-related calculations.

It's important to note that these time units are simply a shorthand notation for representing time in seconds, and do not have any effect on the actual passage of time or the execution of the contract. The time units are simply a convenience for expressing time in a more human-readable form.

L02 - State Variables could be Declared Constant

Criticality	Minor / Informative
Location	DragonToken.sol#L5017
Status	Unresolved

Description

State variables can be declared as constant using the constant keyword. This means that the value of the state variable cannot be changed after it has been set. Additionally, the constant variables decrease gas consumption of the corresponding transaction.

```
Address = 0x000000000000000000000000000000000000dEaD; //Burn LP by send
```

Recommendation

Constant state variables can be useful when the contract wants to ensure that the value of a state variable cannot be changed by any function in the contract. This can be useful for storing values that are important to the contract's behavior, such as the contract's address or the maximum number of times a certain function can be called. The team is advised to add the constant keyword to state variables that never change.

L04 - Conformance to Solidity Naming Conventions

Criticality	Minor / Informative
Location	DragonToken.sol#L1189,3139,3150,3422,3788,4568,4630,4631,4658,4704,5456,5473,5658,5664,5670,5677,5683,5689,5695
Status	Unresolved

Description

The Solidity style guide is a set of guidelines for writing clean and consistent Solidity code. Adhering to a style guide can help improve the readability and maintainability of the Solidity code, making it easier for others to understand and work with.

The followings are a few key points from the Solidity style guide:

1. Use camelCase for function and variable names, with the first letter in lowercase (e.g., myVariable, updateCounter).
2. Use PascalCase for contract, struct, and enum names, with the first letter in uppercase (e.g., MyContract, UserStruct, ErrorEnum).
3. Use uppercase for constant variables and enums (e.g., MAX_VALUE, ERROR_CODE).
4. Use indentation to improve readability and structure.
5. Use spaces between operators and after commas.
6. Use comments to explain the purpose and behavior of the code.
7. Keep lines short (around 120 characters) to improve readability.

```
function CLOCK_MODE() external view returns (string memory);

EIP712Name() internal view returns (string memory) {
        return _name.toStringWithFallback(_nameFallback);
    }

function _EIP712Version() internal view returns (string memory)
{
    return _version.toStringWithFallback(_versionFallback);
}
function DOMAIN_SEPARATOR() external view returns (bytes32);

...
```

Recommendation

By following the Solidity naming convention guidelines, the codebase increased the readability, maintainability, and makes it easier to work with.
Find more information on the Solidity documentation
https://docs.soliditylang.org/en/v0.8.17/style-guide.html#naming-convention.

L09 - Dead Code Elimination

Criticality	Minor / Informative
Location	DragonToken.sol#L67,84,101,118,135,169,186,203,220,237,254,271,288,305,322,339,356,373,390,407,424,441,458,475,509,543,560,577,591,609,627,645,663,681,699,717,735,753,771,789,807,825,843,861,879,897,915,933,951,969,987,1005,1023,1041,1059,1077,1095,1113,1131,1149,1163,1405,1415,1425,1435,1445,1465,1475,1599,1622,1629,1637,1646,1736,1762,1778,1862,1912,1965,1976,2014,2027,2057,2067,2098,2138,2146,2155,2162,2172,2186,2199,2243,2251,2261,2274,2297,2306,2319,2326,2334,2366,2390,2410,2445,2455,2500,2584,2631,2639,2649,2662,2685,2694,2707,2714,2722,2754,2778,2798,2833,2858,2865,2874,2893,2900,2935,2954,2968,3212,3245,3256,3268,3615,3842,3961,3965,4311,4326
Status	Unresolved

Description

In Solidity, dead code is code that is written in the contract, but is never executed or reached during normal contract execution. Dead code can occur for a variety of reasons, such as:

- Conditional statements that are always false.
- Functions that are never called.
- Unreachable code (e.g., code that follows a return statement).

Dead code can make a contract more difficult to understand and maintain, and can also increase the size of the contract and the cost of deploying and interacting with it.

```
function toUint248(uint256 value) internal pure returns
(uint248) {
        if (value > type(uint248).max) {
            revert SafeCastOverflowedUintDowncast(248, value);
        }
        return uint248(value);
    }

function toUint240(uint256 value) internal pure returns
(uint240) {
        if (value > type(uint240).max) {
            revert SafeCastOverflowedUintDowncast(240, value);
        }
        return uint240(value);
    }

...
```

Recommendation

To avoid creating dead code, it's important to carefully consider the logic and flow of the contract and to remove any code that is not needed or that is never executed. This can help improve the clarity and efficiency of the contract.

L11 - Unnecessary Boolean equality

Criticality	Minor / Informative
Location	DragonToken.sol#L5354,5607,5614,5629
Status	Unresolved

Description

Boolean equality is unnecessary when comparing two boolean values. This is because a boolean value is either true or false, and there is no need to compare two values that are already known to be either true or false.

it's important to be aware of the types of variables and expressions that are being used in the contract's code, as this can affect the contract's behavior and performance. The comparison to boolean constants is redundant. Boolean constants can be used directly and do not need to be compared to true or false.

```
if (phasesInitialized == false)
require(phasesInitialized == false, "Phases initialization is
locked");
require(phasesInitialized == false, "Phases initialization is
locked");
require(phasesInitialized == false, "Phases initialization is
already locked");
```

Recommendation

Using the boolean value itself is clearer and more concise, and it is generally considered good practice to avoid unnecessary boolean equalities in Solidity code.

L13 - Divide before Multiply Operation

Criticality	Minor / Informative
Location	DragonToken.sol#L1824,1827,1839,1843,1844,1845,1846,1847,1848,1854
Status	Unresolved

Description

It is important to be aware of the order of operations when performing arithmetic calculations. This is especially important when working with large numbers, as the order of operations can affect the final result of the calculation. Performing divisions before multiplications may cause loss of prediction.

```
denominator := div(denominator, twos)
inverse *= 2 - denominator * inverse
```

Recommendation

To avoid this issue, it is recommended to carefully consider the order of operations when performing arithmetic calculations in Solidity. It's generally a good idea to use parentheses to specify the order of operations. The basic rule is that the multiplications should be prior to the divisions.

L14 - Uninitialized Variables in Local Scope

Criticality	Minor / Informative
Location	DragonToken.sol#L3926,5429,5449,5466,5489
Status	Unresolved

Description

Using an uninitialized local variable can lead to unpredictable behavior and potentially cause errors in the contract. It's important to always initialize local variables with appropriate values before using them.

```
uint256 oldValue, uint256 newValue
bytes memory error_
```

Recommendation

By initializing local variables before using them, the contract ensures that the functions behave as expected and avoid potential issues.

L15 - Local Scope Variable Shadowing

Criticality	Minor / Informative
Location	DragonToken.sol#L4527
Status	Unresolved

Description

Local scope variable shadowing occurs when a local variable with the same name as a variable in an outer scope is declared within a function or code block. When this happens, the local variable "shadows" the outer variable, meaning that it takes precedence over the outer variable within the scope in which it is declared.

```
EIP712(name, "1")
```

Recommendation

It's important to be aware of shadowing when working with local variables, as it can lead to confusion and unintended consequences if not used correctly. It's generally a good idea to choose unique names for local variables to avoid shadowing outer variables and causing confusion.

L16 - Validate Variable Setters

Criticality	Minor / Informative
Location	DragonToken.sol#L5177,5178,5276,5654
Status	Unresolved

Description

The contract performs operations on variables that have been configured on user-supplied input. These variables are missing of proper check for the case where a value is zero. This can lead to problems when the contract is executed, as certain actions may not be properly handled when the value is zero.

```
router = router_;
basePair = basePair_;
externalFeesProcessor = contract_;
to_.transfer(amount_); //Can only send Avax from our contract,
any user's wallet is safe
```

Recommendation

By adding the proper check, the contract will not allow the variables to be configured with zero value. This will ensure that the contract can handle all possible input values and avoid unexpected behavior or errors. Hence, it can help to prevent the contract from being exploited or operating unexpectedly.

L17 - Usage of Solidity Assembly

Criticality	Minor / Informative
Location	DragonToken.sol#L1407,1417,1427,1437,1447,1457,1467,1477,1551,1785,2414,2608,2802,2839,2937,2983,3220
Status	Unresolved

Description

Using assembly can be useful for optimizing code, but it can also be error-prone. It's important to carefully test and debug assembly code to ensure that it is correct and does not contain any errors.

Some common types of errors that can occur when using assembly in Solidity include Syntax, Type, Out-of-bounds, Stack, and Revert.

```
assembly {
            r.slot := slot
        }

assembly {
            r.slot := store.slot
...
            mstore(add(str, 0x20), sstr)
        }

assembly {
                let mm := mulmod(x, y, not(0))
                prod1 := sub(sub(mm, prod0), lt(mm, prod0))
            }

...
```

Recommendation

It is recommended to use assembly sparingly and only when necessary, as it can be difficult to read and understand compared to Solidity code.

L19 - Stable Compiler Version

Criticality	Minor / Informative
Location	DragonToken.sol#L22,1177,1196,1259,1269,1319,1351,1488,1613,1659,2077,2210,2814,2910,2998,3160,3337,3429,3594,3625,3726,3980,4062,4090,4408,4493,4576
Status	Unresolved

Description

The `^` symbol indicates that any version of Solidity that is compatible with the specified version (i.e., any version that is a higher minor or patch version) can be used to compile the contract. The version lock is a mechanism that allows the author to specify a minimum version of the Solidity compiler that must be used to compile the contract code. This is useful because it ensures that the contract will be compiled using a version of the compiler that is known to be compatible with the code.

```
pragma solidity ^0.8.20;
pragma solidity ^0.8.20;
...
```

Recommendation

The team is advised to lock the pragma to ensure the stability of the codebase. The locked pragma version ensures that the contract will not be deployed with an unexpected version. An unexpected version may produce vulnerabilities and undiscovered bugs. The compiler should be configured to the lowest version that provides all the required functionality for the codebase. As a result, the project will be compiled in a well-tested LTS (Long Term Support) environment.

L20 - Succeeded Transfer Check

Criticality	Minor / Informative
Location	DragonToken.sol#L5660,5666
Status	Unresolved

Description

According to the ERC20 specification, the transfer methods should be checked if the result is successful. Otherwise, the contract may wrongly assume that the transfer has been established.

```
Ierc20Token(contract_).transferFrom(address(this), to_,
amount_); //Can only transfer from our own contract
Ierc20Token(contract_).transfer(to_, amount_); //Since
interfaced contract looks at msg.sender then this can only send
from our own contract
```

Recommendation

The contract should check if the result of the transfer methods is successful. The team is advised to check the SafeERC20 library from the Openzeppelin library.

Functions Analysis

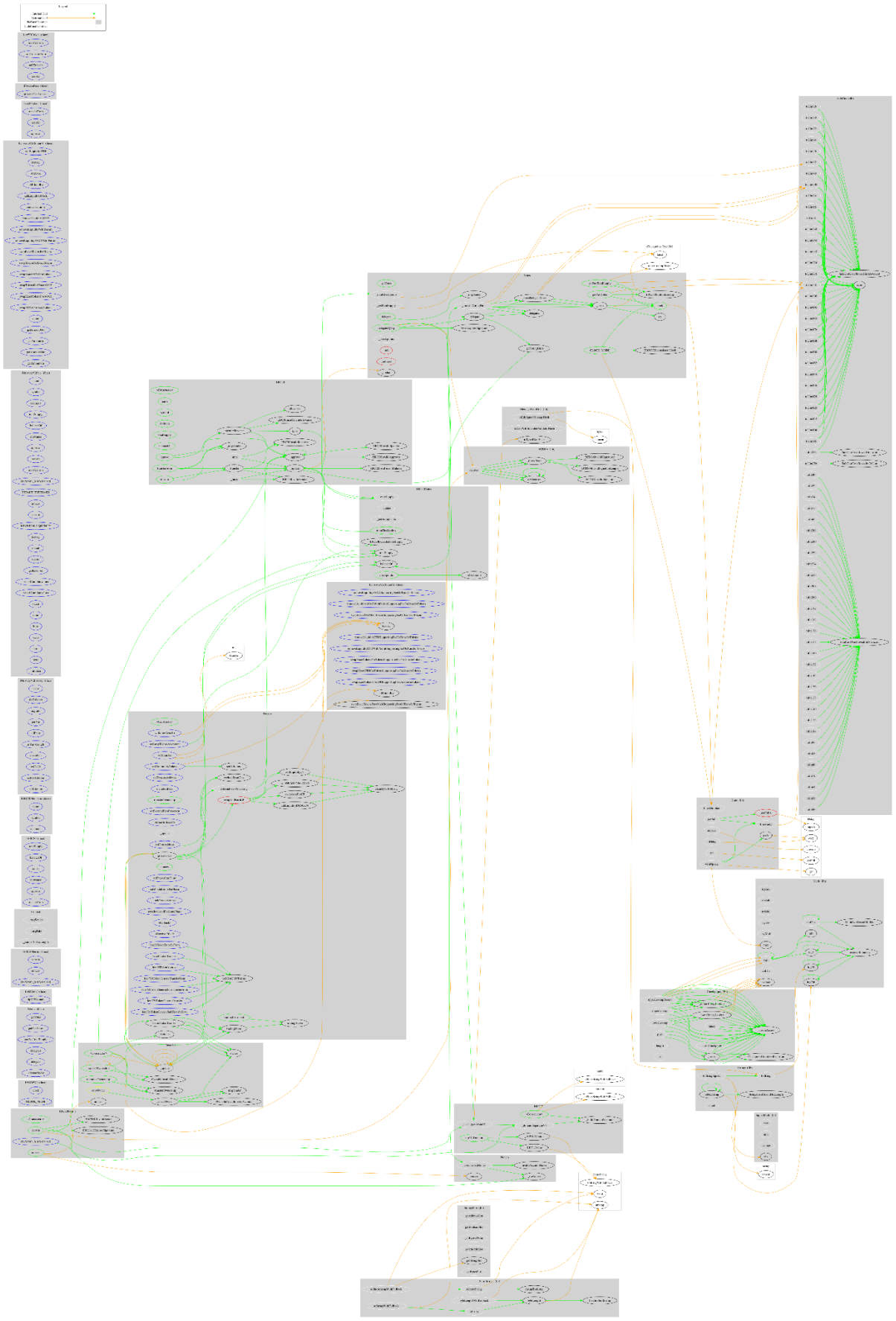
Contract	Type	Bases		
	Function Name	**Visibility**	**Mutability**	**Modifiers**
Dragon	Implementation	ERC20, ERC20Permit, ERC20Votes, Ownable		
		Public	✓	ERC20 ERC20Permit Ownable
	setMainDex	External	✓	onlyOwner
	setCommunityTokens	External	✓	onlyOwner
	setCtRouters	Public	✓	onlyOwner
	setSwapTokensAtAmount	External	✓	onlyOwner
	setTreasuryAddress	External	✓	onlyOwner
	setFarmAddress	External	✓	onlyOwner
	excludeFromFees	Public	✓	onlyOwner
	includeInFees	External	✓	onlyOwner
	transferOwnership	Public	✓	onlyOwner
	setExternalFeesProcessor	External	✓	onlyOwner
	setFeesInBasisPts	External	✓	onlyOwner
	_update	Internal	✓	
	beforeTokenTransfer	Private	✓	
	swapAndBurnLP	Private	✓	

	swapDRAGONForAVAX	Private	✓	
	swapAvaxForCT	Private	✓	
	addLiquidityDRAGON	Private	✓	
	addLiquidityCT	Private	✓	
	cleanBytesToString	Internal		
	externalFeesProcessing	Internal	✓	
	processFees	Public	✓	-
	nonces	Public		-
	tradingActive	Public		-
	tradingRestricted	Public		-
	tradingPhase	Public		-
	setPhasesStartTime	External	✓	onlyOwner
	setWhaleLimitsPerPhase	External	✓	onlyOwner
	lockPhasesSettings	External	✓	onlyOwner
	setAllowlistedForSomePhase	External	✓	onlyOwner
		External	Payable	-
		External	Payable	-
	withdrawAvaxTo	External	✓	onlyOwner
	Ierc20TokenTransferFrom	External	✓	onlyOwner
	Ierc20TokenTransfer	External	✓	onlyOwner
	Ierc20TokenApprove	External	✓	onlyOwner
	Ierc721TokenGenericTransferFrom	External	✓	onlyOwner
	Ierc721TokenGenericSafeTransferFrom	External	✓	onlyOwner

	Ierc721TokenGenericTransfer	External	✓	onlyOwner
	Ierc721TokenGenericSafeTransferData	External	✓	onlyOwner
	noDRAGONTokens	Internal		

Inheritance Graph

Flow Graph



Summary

Dragon contract implements a token mechanism. This audit investigates security issues, business logic concerns and potential improvements. There are some functions that can be abused by the owner like stop transactions. A multi-wallet signing pattern will provide security against potential hacks. Temporarily locking the contract or renouncing ownership will eliminate all the contract threats. There is also a limit of max 8% fees.

Disclaimer

The information provided in this report does not constitute investment, financial or trading advice and you should not treat any of the document's content as such. This report may not be transmitted, disclosed, referred to or relied upon by any person for any purposes nor may copies be delivered to any other person other than the Company without Cyberscope's prior written consent. This report is not nor should be considered an "endorsement" or "disapproval" of any particular project or team. This report is not nor should be regarded as an indication of the economics or value of any "product" or "asset" created by any team or project that contracts Cyberscope to perform a security assessment. This document does not provide any warranty or guarantee regarding the absolute bug-free nature of the technology analyzed, nor do they provide any indication of the technologies proprietors' business, business model or legal compliance. This report should not be used in any way to make decisions around investment or involvement with any particular project. This report represents an extensive assessment process intending to help our customers increase the quality of their code while reducing the high level of risk presented by cryptographic tokens and blockchain technology.

Blockchain technology and cryptographic assets present a high level of ongoing risk Cyberscope's position is that each company and individual are responsible for their own due diligence and continuous security Cyberscope's goal is to help reduce the attack vectors and the high level of variance associated with utilizing new and consistently changing technologies and in no way claims any guarantee of security or functionality of the technology we agree to analyze. The assessment services provided by Cyberscope are subject to dependencies and are under continuing development. You agree that your access and/or use including but not limited to any services reports and materials will be at your sole risk on an as-is where-is and as-available basis Cryptographic tokens are emergent technologies and carry with them high levels of technical risk and uncertainty. The assessment reports could include false positives false negatives and other unpredictable results. The services may access and depend upon multiple layers of third parties.

About Cyberscope

Cyberscope is a blockchain cybersecurity company that was founded with the vision to make web3.0 a safer place for investors and developers. Since its launch, it has worked with thousands of projects and is estimated to have secured tens of millions of investors' funds.

Cyberscope is one of the leading smart contract audit firms in the crypto space and has built a high-profile network of clients and partners.

The Cyberscope team

https://www.cyberscope.io