Cyberscope

Audit Report

Space Frog X

January 2025

Network	BSC
Address	0xB8Fd1438CA2aF47b940FFF34D13cF58C64ac96A1
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
Critical	TSD	Total Supply Diversion	Unresolved
Minor / Informative	CCR	Contract Centralization Risk	Unresolved
Minor / Informative	DDP	Decimal Division Precision	Unresolved
Minor / Informative	IDI	Immutable Declaration Improvement	Unresolved
Minor / Informative	MMN	Misleading Method Naming	Unresolved
Minor / Informative	MTVU	Misleading Tax Variable Usage	Unresolved
Minor / Informative	PLPI	Potential Liquidity Provision Inadequacy	Unresolved
Minor / Informative	RCS	Redundant Conditional Statements	Unresolved
Minor / Informative	RF	Redundant Function	Unresolved
Minor / Informative	RRA	Redundant Repeated Approvals	Unresolved
Minor / Informative	L02	State Variables could be Declared Constant	Unresolved
Minor / Informative	L04	Conformance to Solidity Naming Conventions	Unresolved
Minor / Informative	L07	Missing Events Arithmetic	Unresolved
Minor / Informative	L11	Unnecessary Boolean equality	Unresolved

●	L13	Divide before Multiply Operation	Unresolved
●	L14	Uninitialized Variables in Local Scope	Unresolved
●	L20	Succeeded Transfer Check	Unresolved

Table of Contents

Risk Classification

The criticality of findings in Cyberscope's smart contract audits is determined by evaluating multiple variables. The two primary variables are:

1. **Likelihood of Exploitation**: This considers how easily an attack can be executed, including the economic feasibility for an attacker.
2. **Impact of Exploitation**: This assesses the potential consequences of an attack, particularly in terms of the loss of funds or disruption to the contract's functionality.

Based on these variables, findings are categorized into the following severity levels:

1. **Critical**: Indicates a vulnerability that is both highly likely to be exploited and can result in significant fund loss or severe disruption. Immediate action is required to address these issues.
2. **Medium**: Refers to vulnerabilities that are either less likely to be exploited or would have a moderate impact if exploited. These issues should be addressed in due course to ensure overall contract security.
3. **Minor**: Involves vulnerabilities that are unlikely to be exploited and would have a minor impact. These findings should still be considered for resolution to maintain best practices in security.
4. **Informative**: Points out potential improvements or informational notes that do not pose an immediate risk. Addressing these can enhance the overall quality and robustness of the contract.

Severity	Likelihood / Impact of Exploitation
Critical	Highly Likely / High Impact
Medium	Less Likely / High Impact or Highly Likely/ Lower Impact
Minor / Informative	Unlikely / Low to no Impact

Review

Contract Name	SpaceFrogX
Compiler Version	v0.8.19+commit.7dd6d404
Optimization	200 runs
Explorer	https://bscscan.com/address/0xb8fd1438ca2af47b940fff34d13cf58c64ac96a1
Address	0xb8fd1438ca2af47b940fff34d13cf58c64ac96a1
Network	BSC
Symbol	SFX
Decimals	18
Total Supply	420.000.000.000
Badge Eligibility	Must Fix Criticals

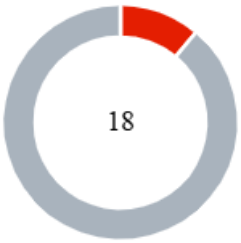
Audit Updates

Initial Audit	23 Jan 2025

Source Files

Filename	SHA256
SpaceFrogX.sol	8189549472edefde6fe4177302a96513d48ea3788315e98bb5316554b1a5d1f7

Findings Breakdown



Severity	Unresolved	Acknowledged	Resolved	Other
Critical	2	0	0	0
Medium	0	0	0	0
Minor / Informative	16	0	0	0

ST - Stops Transactions

Criticality	Critical
Location	SpaceFrogX.sol#L370
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.

```
    if (!exemptFee[sender] && !exemptFee[recipient]) {
        require(tradingEnabled, "Trading not enabled");
    }
```

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.

TSD - Total Supply Diversion

Criticality	Critical
Location	SpaceFrogX.sol#L405
Status	Unresolved

Description

The total supply of a token is the total number of tokens that have been created, while the balances of individual accounts represent the number of tokens that an account owns. The total supply and the balances of individual accounts are two separate concepts that are managed by different variables in a smart contract. These two entities should be equal to each other.

In the contract, the amount that is added to the total supply does not equal the amount that is added to the balances. As a result, the sum of balances is diverse from the total supply.

Specifically, in the `_transfer` function when the `burnAmount` is added to the `_balance` of the `deadWallet`, it is not subtracted from the balance of the `sender`.

```
    _balances[deadWallet] += burnAmount;
```

Recommendation

The total supply and the balance variables are separate and independent from each other. The total supply represents the total number of tokens that have been created, while the balance mapping stores the number of tokens that each account owns. The sum of balances should always equal the total supply.

CCR - Contract Centralization Risk

Criticality	Minor / Informative
Location	SpaceFrogX.sol#L477,486,493,501
Status	Unresolved

Description

The contract's functionality and behavior are heavily dependent on external parameters or configurations. While external configuration can offer flexibility, it also poses several centralization risks that warrant attention. Centralization risks arising from the dependence on external configuration include Single Point of Control, Vulnerability to Attacks, Operational Delays, Trust Dependencies, and Decentralization Erosion.

```
function setSwapTokens(uint256 new_amount) external onlyOwner
{/*..*/}
function enableTrading() external onlyOwner {/*..*/}
function excludeFromFee(address _address) external onlyOwner
{/*..*/}
function recoverBEP20FromContract(address _tokenAddy, uint256
_amount) external onlyOwner {/*..*/}
```

Recommendation

To address this finding and mitigate centralization risks, it is recommended to evaluate the feasibility of migrating critical configurations and functionality into the contract's codebase itself. This approach would reduce external dependencies and enhance the contract's self-sufficiency. It is essential to carefully weigh the trade-offs between external configuration flexibility and the risks associated with centralization.

DDP - Decimal Division Precision

Criticality	Minor / Informative
Location	SpaceFrogX.sol#L405,443
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.

```
   function _transfer(address sender, address recipient,
uint256 amount) internal override {
        //...
        if (providingLiquidity && sender != pair)
Liquify(feeswap, currentTaxes);
        super._transfer(sender, recipient, amount - fee);
        if (fee > 0) {
            if (feeswap > 0) {
                uint256 feeAmount = (amount * feeswap) / 100 ;
                super._transfer(sender, address(this),
feeAmount);
            }
            if (currentTaxes.burn > 0) {
                uint256 burnAmount = (currentTaxes.burn *
amount) / 100;
                _balances[deadWallet] += burnAmount;
                emit Transfer(sender, deadWallet, burnAmount);
            }

        }
    }

        function Liquify(uint256 feeswap, Taxes memory
swapTaxes) private
            //...
            uint256 marketing1Amt = unitBalance * 2 *
swapTaxes.marketing1;
            if (marketing1Amt > 0) {

payable(Marketingwallet).sendValue(marketing1Amt);
            }

            uint256 marketing2Amt = unitBalance * 2 *
swapTaxes.marketing2;
             if (marketing2Amt > 0) {

payable(Desenvolvimento).sendValue(marketing2Amt);
            }

        }
    }
```

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	SpaceFrogX.sol#L302,303
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.

```
        router = _router;
         pair = _pair;
```

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.

MMN - Misleading Method Naming

Criticality	Minor / Informative
Location	SpaceFrogX.sol#L412
Status	Unresolved

Description

Methods can have misleading names if their names do not accurately reflect the functionality they contain or the purpose they serve. The contract uses the method name `Liquify` , however the configuration of the contract shows that the `addLiquidity` function will never be triggered because in both `buytaxes` and `sellTaxes` the `liquidity = 0` . Misleading method names can lead to confusion, making the code more difficult to read and understand. Methods can have misleading names if their names do not accurately reflect the functionality they contain or the purpose they serve.

```
    struct Taxes {
          uint256 marketing1;
          uint256 marketing2;
          uint256 burn;
          uint256 liquidity;
      }

      Taxes public buytaxes = Taxes(4, 2, 1, 0);
      Taxes public sellTaxes = Taxes(4, 2, 1, 0);
      //...
      function Liquify(uint256 feeswap, Taxes memory swapTaxes)
private lockTheSwap {
      //...
              uint256 ethToAddLiquidityWith = unitBalance *
swapTaxes.liquidity;

              if (ethToAddLiquidityWith > 0) {
                  addLiquidity(tokensToAddLiquidityWith,
ethToAddLiquidityWith);
              }
      //...
      }
```

Recommendation

It's always a good practice for the contract to contain method names that are specific and descriptive. The team is advised to keep in mind the readability of the code.

MTVU - Misleading Tax Variable Usage

Criticality	Minor / Informative
Status	Unresolved

Description

The contract utilizes a specific tax variable, suggesting that this tax applies solely to the transactions. Contrarily, the implemented logic indiscriminately applies this tax across different transfer types. This misuse of the variable name leads to a misunderstanding of its application, which could result in unexpected tax implications for all transaction types, contrary to the intended exclusive focus based on the variable naming. In this case the `buytaxes` are used in every transfer other than when the recipient is the pair.

```
    Taxes public buytaxes = Taxes(4, 2, 1, 0);

    function _transfer(address sender, address recipient,
uint256 amount) internal override {
        //...
        } else if (recipient != pair) {
            feeswap = buytaxes.liquidity + buytaxes.marketing1
+ buytaxes.marketing2;
            feesum = feeswap + buytaxes.burn;
            currentTaxes = buytaxes;
        }
        //...
    }
```

Recommendation

To ensure the contract operates transparently and aligns with user expectations, it is essential to adjust the tax application logic to correspond with each specific type of transfer. The tax variable naming and functionality should accurately reflect the type of transaction it is intended for. If the tax rate is to be applied universally to all transfers, adopting a more general tax rate variable would be appropriate. This change will ensure that tax implications are clear and predictable, preventing any unintended consequences and enhancing the overall reliability of the contract's financial transactions.

PLPI - Potential Liquidity Provision Inadequacy

Criticality	Minor / Informative
Location	SpaceFrogX.sol#L454
Status	Unresolved

Description

The contract operates under the assumption that liquidity is consistently provided to the pair between the contract's token and the native currency. However, there is a possibility that liquidity is provided to a different pair. This inadequacy in liquidity provision in the main pair could expose the contract to risks. Specifically, during eligible transactions, where the contract attempts to swap tokens with the main pair, a failure may occur if liquidity has been added to a pair other than the primary one. Consequently, transactions triggering the swap functionality will result in a revert.

```
    function swapTokensForETH(uint256 tokenAmount) private {
        //...

router.swapExactTokensForETHSupportingFeeOnTransferTokens(
            tokenAmount,
            0,
            path,
            address(this),
            block.timestamp
        );
    }
```

Recommendation

The team is advised to implement a runtime mechanism to check if the pair has adequate liquidity provisions. This feature allows the contract to omit token swaps if the pair does not have adequate liquidity provisions, significantly minimizing the risk of potential failures.

Furthermore, the team could ensure the contract has the capability to switch its active pair in case liquidity is added to another pair.

Additionally, the contract could be designed to tolerate potential reverts from the swap functionality, especially when it is a part of the main transfer flow. This can be achieved by executing the contract's token swaps in a non-reversible manner, thereby ensuring a more resilient and predictable operation.

RCS - Redundant Conditional Statements

Criticality	Minor / Informative
Location	SpaceFrogX.sol#L419
Status	Unresolved

Description

The contract contains redundant conditional statements that can be simplified to improve code efficiency and performance. Conditional statements that merely return the result of an expression are unnecessary and lead to larger code size, increased memory usage, and slower execution times.

In this case `swapTokens` variable is instanciated as `1260000000 * 10**18` and it can oly be changed to values `>1`. Therefore, the following statement is redundant.

```
   function Liquify(uint256 feeswap, Taxes memory swapTaxes)
private lockTheSwap {
        //...
            if (swapTokens > 1) {
                contractBalance = swapTokens;
            }
        //...
    }

    function setSwapTokens(uint256 new_amount) external
onlyOwner {
        require(new_amount <= 4200000000, "Swap threshold
amount should be lower or equal to 1% of tokens");
        require(new_amount >= 420000000,  "Swap threshold
amount should be greater than or equal to 0.1% of tokens");
        swapTokens = new_amount * 10**decimals();
    }
```

Recommendation

It is recommended to refactor conditional statements that return results by eliminating unnecessary code structures and directly returning the outcome of the expression. This practice minimizes the number of operations required, reduces the code footprint, and optimizes memory and gas usage. Simplifying such statements makes the code more readable and improves its overall performance.

RF - Redundant Function

Criticality	Minor / Informative
Location	SpaceFrogX.sol#L463
Status	Unresolved

Description

The `addLiquidity` is a private function and is never used. In the current configuration of the contract the `buytaxes` and `sellTaxes` structs are initialized with `liquidity = 0`. This means that in the function `Liquify` the `addLiquidity` function will never be used as the `ethToAddLiquidityWith` will always be `0`. Therefore, the `addLiquidity` is redundant.

```
    struct Taxes {
          uint256 marketing1;
          uint256 marketing2;
          uint256 burn;
          uint256 liquidity;
     }

     Taxes public buytaxes = Taxes(4, 2, 1, 0);
     Taxes public sellTaxes = Taxes(4, 2, 1, 0);
     //...
     function Liquify(uint256 feeswap, Taxes memory swapTaxes)
private lockTheSwap {
     //...
              uint256 ethToAddLiquidityWith = unitBalance *
swapTaxes.liquidity;

              if (ethToAddLiquidityWith > 0) {
                  addLiquidity(tokensToAddLiquidityWith,
ethToAddLiquidityWith);
              }
     //...
     }
```

Recommendation

It is recommended to remove unnecessary functions as they will increase the complexity of the contract without adding any value. Removing unused functions also improves code readability. This streamlining can also contribute to better gas efficiency and overall contract optimization.

RRA - Redundant Repeated Approvals

Criticality	Minor / Informative
Location	SpaceFrogX.sol#L453
Status	Unresolved

Description

The contract is designed to `approve` token transfers during the contract's operation by calling the _approve function before specific operations. This approach results in additional gas costs since the approval process is repeated for every operation execution, leading to inefficiencies and increased transaction expenses.

```
    function swapTokensForETH(uint256 tokenAmount) private {
        address[] memory path = new address[](2);
        path[0] = address(this);
        path[1] = router.WETH();
        _approve(address(this), address(router), tokenAmount);

router.swapExactTokensForETHSupportingFeeOnTransferTokens(
            tokenAmount,
            0,
            path,
            address(this),
            block.timestamp
        );
    }
```

Recommendation

Since the approved address is a trusted third-party source, it is recommended to optimize the contract by approving the maximum amount of tokens once in the initial set of the variable, rather than before each operation. This change will reduce the overall gas consumption and improve the efficiency of the contract.

L02 - State Variables could be Declared Constant

Criticality	Minor / Informative
Location	SpaceFrogX.sol#L264,265
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 public Marketingwallet =
(0xF94E2162A6D8D6f4d0142Ab090478Bd4ecfe57C0)
    address public Desenvolvimento =
(0x1B19A9E18DCe1A138B566D93b9Afd3F738FDD582)
```

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	SpaceFrogX.sol#L264,265,271,412,477,493,501
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.

```
address public Marketingwallet =
(0xF94E2162A6D8D6f4d0142Ab090478Bd4ecfe57C0)
address public Desenvolvimento =
(0x1B19A9E18DCe1A138B566D93b9Afd3F738FDD582)
string public Developer

...
```

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/stable/style-guide.html#naming-conventions.

L07 - Missing Events Arithmetic

Criticality	Minor / Informative
Location	SpaceFrogX.sol#L480
Status	Unresolved

Description

Events are a way to record and log information about changes or actions that occur within a contract. They are often used to notify external parties or clients about events that have occurred within the contract, such as the transfer of tokens or the completion of a task.

It's important to carefully design and implement the events in a contract, and to ensure that all required events are included. It's also a good idea to test the contract to ensure that all events are being properly triggered and logged.

```
swapTokens = new_amount * 10**decimals()
```

Recommendation

By including all required events in the contract and thoroughly testing the contract's functionality, the contract ensures that it performs as intended and does not have any missing events that could cause issues with its arithmetic.

L11 - Unnecessary Boolean equality

Criticality	Minor / Informative
Location	SpaceFrogX.sol#L494
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.

```
require(exemptFee[_address] != true,"Account is already
excluded")
```

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	SpaceFrogX.sol#L429,430,436,441
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.

```
uint256 unitBalance = deltaBalance / (denominator -
swapTaxes.liquidity)
uint256 ethToAddLiquidityWith = unitBalance *
swapTaxes.liquidity
```

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	SpaceFrogX.sol#L374,375,377
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 feeswap
uint256 feesum
Taxes memory currentTaxes
```

Recommendation

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

L20 - Succeeded Transfer Check

Criticality	Minor / Informative
Location	SpaceFrogX.sol#L505
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.

```
IBEP20(_tokenAddy).transfer(Marketingwallet, _amount)
```

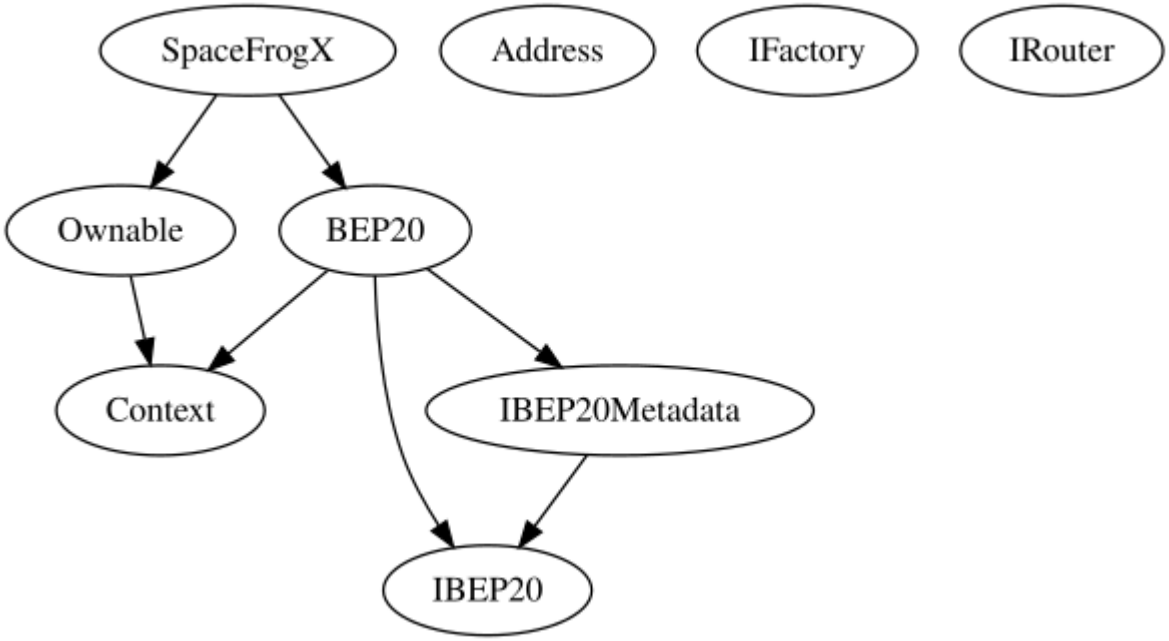
Recommendation

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.

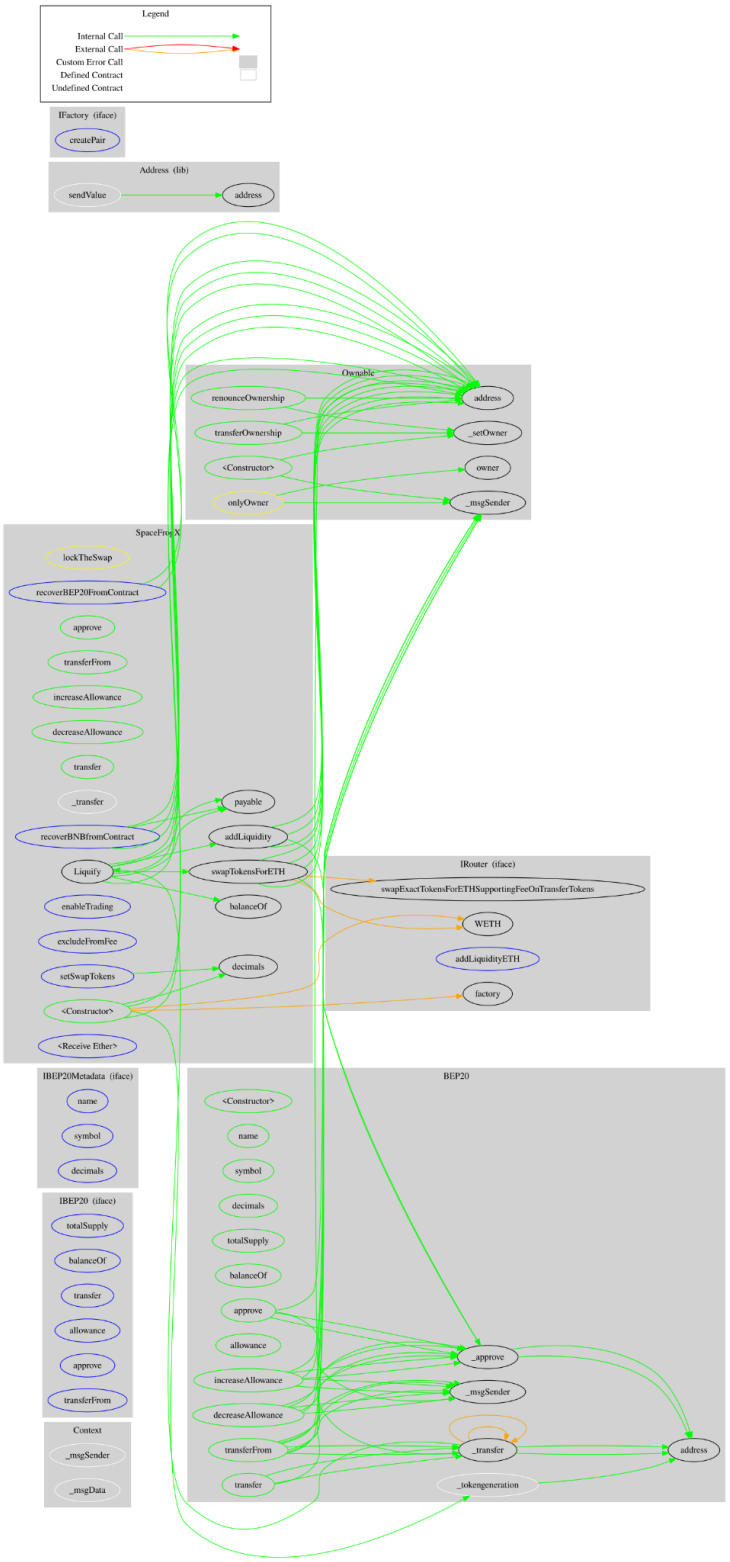
Function Analysis

Contract	Type	Bases		
	Function Name	**Visibility**	**Mutability**	**Modifiers**
SpaceFrogX	Implementation	BEP20, Ownable		
		Public	✓	BEP20
	approve	Public	✓	-
	transferFrom	Public	✓	-
	increaseAllowance	Public	✓	-
	decreaseAllowance	Public	✓	-
	transfer	Public	✓	-
	_transfer	Internal	✓	
	Liquify	Private	✓	lockTheSwap
	swapTokensForETH	Private	✓	
	addLiquidity	Private	✓	
	setSwapTokens	External	✓	onlyOwner
	enableTrading	External	✓	onlyOwner
	excludeFromFee	External	✓	onlyOwner
	recoverBEP20FromContract	External	✓	onlyOwner
	recoverBNBfromContract	External	✓	-
		External	Payable	-

Inheritance Graph



Flow Graph


Legend
Internal Call
External Call
Custom Error Call
Defined Contract
Undefined Contract
IFactory (iface)
createPair
Address (lib)
sendValue
address
Ownable
renounceOwnership
transferOwnership
<Constructor>
onlyOwner
address
_setOwner
owner
_msgSender
SpaceFrogX
lockTheSwap
recoverBEP20FromContract
approve
transferFrom
increaseAllowance
decreaseAllowance
transfer
_transfer
recoverBNBfromContract
Liquify
enableTrading
excludeFromFee
setSwapTokens
<Constructor>
<Receive Ether>
payable
addLiquidity
swapTokensForETH
balanceOf
decimals
IRouter (iface)
swapExactTokensForETHSupportingFeeOnTransferTokens
WETH
addLiquidityETH
factory
IBEP20Metadata (iface)
name
symbol
decimals
IBEP20 (iface)
totalSupply
balanceOf
transfer
allowance
approve
transferFrom
Context
_msgSender
_msgData
BEP20
<Constructor>
name
symbol
decimals
totalSupply
balanceOf
approve
allowance
increaseAllowance
decreaseAllowance
transferFrom
transfer
_approve
_msgSender
_transfer
_tokengeneration
address

Summary

Space Frog X 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. There is also a limit of max 7% 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

cyberscope.io