Cyberscope

Audit Report

BFC Smart Contract

February 2023

Type	BEP20
Network	BSC
Address	0xDB296b2a7123BD5a39e9E42107f929c0e960b230
Audited by	© cyberscope

Table of Contents

Review

Contract Name	EDAO
Compiler Version	v0.8.16+commit.07a7930e
Optimization	200 runs
Explorer	https://bscscan.com/address/0xdb296b2a7123bd5a39e9e42107f929c0e960b230
Address	0xdb296b2a7123bd5a39e9e42107f929c0e960b230
Network	BSC

Audit Updates

Initial Audit	06 Feb 2023

Source Files

Filename	SHA256
EDAO.sol	0fa1251db24b5177f569c382da12bb3dc2e868b4314bbc5ce258de1735323979
IERC20.sol	10e4ea87cfa7a34a3f975518094a7f055f39d8215a90c15e1f623d4923e72a09
SafeMath.sol	7e9202b7f153422a579b0870832823f29db8ce0e7b7c5438298042030df029ae

Introduction

The BFC Smart Contract implements a lottery mechanism that allows multiple participants to enter and earn rewards. Upon depositing funds, 0.9% of the amount is deducted as fees, with 0.1% added to the lottery pool. The fee amount is divided into two portions, with 87% being directed to the `feeReceiver` address and 13% to the `feeReceiver13` address.

Roles

Owner

The owner has authority over the following functions:

- `function transferOwnership(address newOwner)`
- `function EmergencyWithdrawal(uint256 _bal)`

User

The user can interact with the following functions:

- `function register(address _referral)`
- `function deposit(uint256 _amount)`
- `function depositBySplit(uint256 _amount)`
- `function transferBySplit(address _receiver, uint256 _amount, uint256 _type)`
- `function lotteryBet(uint256 _number)`
- `function withdraw()`
- `function distributePoolRewards()`
- `function getLottoryWinners(uint256 _day)`
- `function getTeamDeposit(address _userAddr)`
- `function getCurDay()`
- `function getCurCycle()`
- `function getDayInfos(uint256 _day)`
- `function getUserInfos(address _userAddr)`
- `function getBalInfos(uint256 _bal)`
- `function getAllLotteryRecord(uint256 _day, uint256 _number)`
- `function getTeamUsers(address _userAddr, uint256 _layer)`
- `function getUserCycleMax(address _userAddr, uint256 _cycle)`
- `function getDepositors()`
- `function getContractInfos()`

Diagnostics

Critical Medium Minor / Informative

Severity	Code	Description	Status
●	CC	Code Complexity	Unresolved
●	DSM	Data Structure Misuse	Unresolved
●	CR	Circular Referencing	Unresolved
●	RC	Redundant Checks	Unresolved
●	CR	Code Repetition	Unresolved
●	CO	Code Optimization	Unresolved
●	RC	Redundant Calculations	Unresolved
●	MEM	Misleading Error Messages	Unresolved
●	WSR	Winner Selection Randomization	Unresolved
●	OCTD	Transfers Contract's Tokens	Unresolved
●	MZ	Manual Zeroing	Unresolved
●	L02	State Variables could be Declared Constant	Unresolved
●	L04	Conformance to Solidity Naming Conventions	Unresolved

●	L05	Unused State Variable	Unresolved
●	L08	Tautology or Contradiction	Unresolved
●	L11	Unnecessary Boolean equality	Unresolved
●	L13	Divide before Multiply Operation	Unresolved
●	L14	Uninitialized Variables in Local Scope	Unresolved
●	L16	Validate Variable Setters	Unresolved
●	L19	Stable Compiler Version	Unresolved
●	L20	Succeeded Transfer Check	Unresolved

CC - Code Complexity

Criticality	Minor / Informative
Status	Unresolved

Description

The analyzed smart contract exhibits a high level of complexity that is not proportional to its functionality. The code includes several redundant or convoluted logic branches, calculations and variables, making it difficult to understand and potentially increasing the risk of unintended behavior.

Recommendation

The team is advised to revisit the code to simplify and clarify its structure, promoting transparency, readability and reducing the potential for errors. A refactoring of the code would serve to improve its overall functionality and security.

DSM - Data Structure Misuse

Criticality	Minor / Informative
Location	EDAO.sol#L326
Status	Unresolved

Description

The contract uses the valuable `depositors` as an array. This means a user address can be stored in the array more than once, while it should be unique.

```
depositors.push(_userAddr);
```

Recommendation

The contract could use a data structure like a Set, which will ensure the uniqueness of each address.

CR - Circular Referencing

Criticality	Medium
Location	EDAO.sol#L483
Status	Unresolved

Description

Circular referencing is a mutual relationship between two entities where each entity refers to the other. This type of relationship can occur in a Solidity smart contract where two users, "A" and "B", have a mutual reference to each other. The `_updateUplineReward()` function distributes rewards to the referrers. Since the users "A" and "B" can reference each other they can share the calculated rewards repeatedly when the function is executed.

```
function _updateUplineReward(address _userAddr, uint256 _amount) private {
...
}
```

Recommendation

The team is advised to carefully check the implementation and design of the referrers system and to ensure that such dependencies are managed effectively, or not allowed at all. This results in a more efficient and optimized contract, reducing the likelihood of unintended consequences and errors.

RC - Redundant Checks

Criticality	Minor / Informative
Location	EDAO.sol#L382-402
Status	Unresolved

Description

The contract performs redundant checks and, by extension, execute more code segments. Both if-statements listed below manipulate the `staticReward` variable. If the first check passes then the second will, as well, which makes the execution of the first if-statement redundant. It should be noted that this scenario is valid only when `freezeStaticReward` variable is enabled.

```
if(user.totalFreezed > user.totalRevenue){ ... }
...
if(user.totalFreezed.mul(4258).div(1000) >= user.totalRevenue){ ... }
```

Recommendation

The team is advised to take into consideration these segments and rewrite them so the runtime will be more performant. That way it will improve the efficiency and performance of the source code and reduce the cost of executing it.

CR - Code Repetition

Criticality	Minor / Informative
Location	EDAO.sol#L514,625
Status	Unresolved

Description

The contract contains repetitive code segments. There are potential issues that can arise when using code segments in Solidity. Some of them can lead to issues like gas efficiency, complexity, readability, security, and maintainability of the source code. It is generally a good idea to try to minimize code repetition where possible. The reward calculation in the following segments uses them same logic, with changes only to amount and the fee percentage.

```
uint256 reward =
newAmount.mul(invitePercents[i]).div(baseDivider).mul(Reduceproduction()).div(10000);
...
uint256 reward =
lotteryPool.mul(lotteryWinnerPercents[i]).div(baseDivider).mul(Reduceproduction()).di
v(10000);
```

Recommendation

The team is advised to avoid repeating the same code in multiple places, which can make the contract easier to read and maintain. The authors could try to reuse code wherever possible, as this can help to reduce the complexity and size of the contract. For instance, the contract could reuse the common code segments in an internal function in order to avoid repeating the same code in multiple places.

CO - Code Optimization

Criticality	Minor / Informative
Location	EDAO.sol#L516-547
Status	Unresolved

Description

There are code segments that could be optimized. A segment may be optimized so that it becomes a smaller size, consumes less memory, executes more rapidly, or performs fewer operations.

The contract performs many checks, in which, most of them perform the same code if evaluated truthly. This makes the code harder to read and understand, while repeating many code segments. Additionally, the variable `UTDAO` which is stored in the `RewardInfo` struct is the calculated amount of the static reward divided by 12. The static reward is already being store in the `statics` variable of the `RewardInfo` struct. The contract could avoid storing both values in the state.

```
if(userInfo[upline].level > i ){
    upRewards.invited = upRewards.invited.add(reward);
    userInfo[upline].totalRevenue = userInfo[upline].totalRevenue.add(reward);
}else{
    if(userInfo[upline].level ==  4 && i < 5 ){
        upRewards.invited = upRewards.invited.add(reward);
        userInfo[upline].totalRevenue = userInfo[upline].totalRevenue.add(reward);
    }
    ...
}
...
uint256 UTDAO = staticReward.div(12);
userRewards.statics = userRewards.statics.add(staticReward);
userRewards.UTDAO = userRewards.UTDAO.add(UTDAO);
```

Recommendation

The team is advised to take into consideration these segments and rewrite them so the runtime will be more performant. That way it will improve the efficiency and performance of the source code and reduce the cost of executing it. Regarding the `UTDAO` value, a recommended approach would be to only store the `statics` variable, and when the `UTDAO` value is needed it can be calculated as `statics / 12`.

RC - Redundant Calculations

Criticality	Minor / Informative
Location	EDAO.sol#L360,395
Status	Unresolved

Description

Redundant calculations can occur in various forms in a smart contract, such as repetitive calculations for the same result, the recalculation of unchanging values, and the repeated execution of complex computations. These redundant calculations can significantly increase the gas cost of executing the smart contract. The contract executes redundant calculations in the segments listed below.

- `_updateLevel()`: The `getTeamDeposit()` function is executed on every iteration, when it could only execute once outside of the loop.
- `_unfreezeCapitalOrReward()`: The operation `user.totalFreezed.mul(4258).div(1000)` is calculated twice.

```
for(uint256 i = user.level; i < levelDeposit.length; i++){
    if(user.maxDeposit >= levelDeposit[i]){
        (uint256 maxTeam, uint256 otherTeam, ) = getTeamDeposit(_userAddr);
        if(maxTeam >= levelInvite[i] && otherTeam >= levelInvite[i] && user.teamNum
>= levelTeam[i]){
            user.level = i + 1;
        }
    }
}
...
if(user.totalFreezed.mul(4258).div(1000) >= user.totalRevenue){
    uint256 leftCapital =
user.totalFreezed.mul(4258).div(1000).sub(user.totalRevenue);
    ...
}
```

Recommendation

The team is advised to take into consideration these segments and rewrite them so the runtime will be more performant. That way it will improve the efficiency and performance of the source code and reduce the cost of executing it.

MEM - Misleading Error Messages

Criticality	Minor / Informative
Location	EDAO.sol#L182,192,197,209,293,294,306,724,729
Status	Unresolved

Description

The contract is using misleading error messages. These error messages do not accurately reflect the problem, making it difficult to identify and fix the issue. As a result, the users will not be able to find the root cause of the error.

```
require(userInfo[msg.sender].maxDeposit == 0, "actived");
require(_amount >= minDeposit && _amount.mod(minDeposit) == 0, "amount err");
require(_amount > 0, "amount err");
require(block.timestamp < lotteryEnd, "today is over");
require(_amount >= minDeposit && _amount <= maxDeposit && _amount.mod(minDeposit) ==
0, "amount err");
require(user.maxDeposit == 0 || _amount >= user.maxDeposit, "too less");
...
```

Recommendation

The team is advised to carefully review the source code in order to address these issues. To accelerate the debugging process and mitigate these issues, the team should use more specific and descriptive error messages.

WSR - Winner Selection Randomization

Criticality	Minor / Informative
Location	EDAO.sol#L637
Status	Unresolved

Description

A lottery is a form of gambling in which players buy tickets to participate and the winners are determined by chance. If the winners are not selected randomly, it would compromise the fairness of the lottery and defeat its purpose, which is to provide an equal chance for all participants to win. The contract does not pick winners randomly, instead it picks the first 10 or less users that have entered the lottery.

```
function getLottoryWinners(uint256 _day) public view returns(address[] memory) {
    uint256 newbies = dayNewbies[_day];
    address[] memory winners = new address[](10);
    uint256 counter;
    for(uint256 i = newbies; i >= 0; i--){
        for(uint256 j = 0; j < allLotteryRecord[_day][i].length; j++ ){
            address lotteryUser = allLotteryRecord[_day][i][j];
            if(lotteryUser != address(0)){
                winners[counter] = lotteryUser;
                counter++;
                if(counter >= 10) break;
            }
        }
        if(counter >= 10 || i == 0 || newbies.sub(i) >= maxSearchDepth) break;
    }
    return winners;
}
```

Recommendation

It is crucial that the smart contract implements a secure and transparent random number generation method to ensure that the winner selection process is unbiased and unpredictable. The contract could use an advanced randomization technique that guarantees an acceptable randomization factor. For instance, the Chainlink VRF (Verifiable Random Function). https://docs.chain.link/docs/chainlink-vrf

OCTD - Transfers Contract's Tokens

Criticality	Minor / Informative
Status	Unresolved

Description

The contract owner has the authority to claim all the balance of the contract. The owner may take advantage of it by calling the `EmergencyWithdrawal` function.

```
function EmergencyWithdrawal(uint256 _bal) public   {
    require(_owner == msg.sender);
    usdt.transfer(msg.sender, _bal);
}
```

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. That risk can be prevented by temporarily locking the contract or renouncing ownership.

MZ - Manual Zeroing

Criticality	Minor / Informative
Location	EDAO.sol#L222,223,224,225,229,230
Status	Unresolved

Description

Zeroing memory is related to the management of storage objects. When a storage object is no longer needed, it is important to reset its value to the default zero value of its type in order to free up storage space. However, the process of zeroing memory can be challenging, as it requires writing custom code to explicitly set each byte of the storage object to zero. This can be time-consuming, error-prone, and may result in higher gas costs.

```
userRewards.statics = 0;
userRewards.invited = 0;
userRewards.level5Released = 0;
userRewards.lotteryWin = 0;
userRewards.capitals = 0;
userRewards.UTDAO = 0;
```

Recommendation

The team is advised to replace all zeroing operations with a delete operation which is more readable and gas efficient.

L02 - State Variables could be Declared Constant

Criticality	Minor / Informative
Location	EDAO.sol#L23
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 Owner
```

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	EDAO.sol#L7,23,25,26,27,28,29,30,31,32,33,34,35,36,38,39,40,41,50,51,52,55,56,61,151,153,166,174,181,189,205,240,261,637,655,680,684,688,692,696,700,722,728
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 WETH() external pure returns (address);
address Owner
uint256 private constant baseDivider = 10000
uint256 private constant feePercents = 90
uint256 private constant minDeposit = 100e18
uint256 private constant maxDeposit = 5000e18
uint256 private constant baseDeposit = 1000e18
uint256 private constant splitPercents = 2100
uint256 private constant lotteryPercents = 900
uint256 private constant transferFeePercents = 1000
uint256 private constant OverTime = 0
uint256 public MAXBal = 0
uint256 public MAXrebackBal = 0
uint256 private constant dayRewardPercents = 80

...
```

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.

L05 - Unused State Variable

Criticality	Minor / Informative
Location	EDAO.sol#L23,33
Status	Unresolved

Description

An unused state variable is a state variable that is declared in the contract, but is never used in any of the contract's functions. This can happen if the state variable was originally intended to be used, but was later removed or never used.

Unused state variables can create clutter in the contract and make it more difficult to understand and maintain. They can also increase the size of the contract and the cost of deploying and interacting with it.

```
address Owner
uint256 private constant OverTime = 0
```

Recommendation

To avoid creating unused state variables, it's important to carefully consider the state variables that are needed for the contract's functionality, and to remove any that are no longer needed. This can help improve the clarity and efficiency of the contract.

L08 - Tautology or Contradiction

Criticality	Minor / Informative
Location	EDAO.sol#L768
Status	Unresolved

Description

A tautology is a logical statement that is always true, regardless of the values of its variables. A contradiction is a logical statement that is always false, regardless of the values of its variables.

Using tautologies or contradictions can lead to unintended behavior and can make the code harder to understand and maintain. It is generally considered good practice to avoid tautologies and contradictions in the code.

```
i >= 0
```

Recommendation

The team is advised to carefully consider the logical conditions is using in the code and ensure that it is well-defined and make sense in the context of the smart contract.

L11 - Unnecessary Boolean equality

Criticality	Minor / Informative
Location	EDAO.sol#L453
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.

```
order.isUnfreezed == false &&
      block.timestamp >= order.unfreeze &&
      _amount >= order.amount
```

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	EDAO.sol#L396,423,424,425,460,610,740
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 staticReward = order
        .amount
        .mul(dayRewardPercents)
        .mul(dayPerCycle)
        .div(timeStep)
        .div(baseDivider)
        .mul(Reduceproduction())
        .div(10000)
```

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	EDAO.sol#L386,589,767,792
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.

```
bool isNewbie
uint256 newAmount
uint256 counter
uint256 maxTeam
```

Recommendation

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

L16 - Validate Variable Setters

Criticality	Minor / Informative
Location	EDAO.sol#L196,197,200,916
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.

```
feeReceiver = _feeReceiver
feeReceiver13 = _feeReceiver13
defaultRefer = _defaultRefer
_owner = newOwner
```

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.

L19 - Stable Compiler Version

Criticality	Minor / Informative
Location	EDAO.sol#L2
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 ^0.8.6;
```

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	EDAO.sol#L238,311,333,424,425,693,921
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.

```
usdt.transferFrom(msg.sender, address(this), _amount)
usdt.transfer(msg.sender, withdrawable)
DAO.transfer(sender, DAObal)
usdt.transfer(feeReceiver, totalFee.mul(87).div(100))
usdt.transfer(feeReceiver13, totalFee.mul(13).div(100))
usdt.transfer(defaultRefer, _bal)
usdt.transfer(msg.sender, _bal)
```

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.

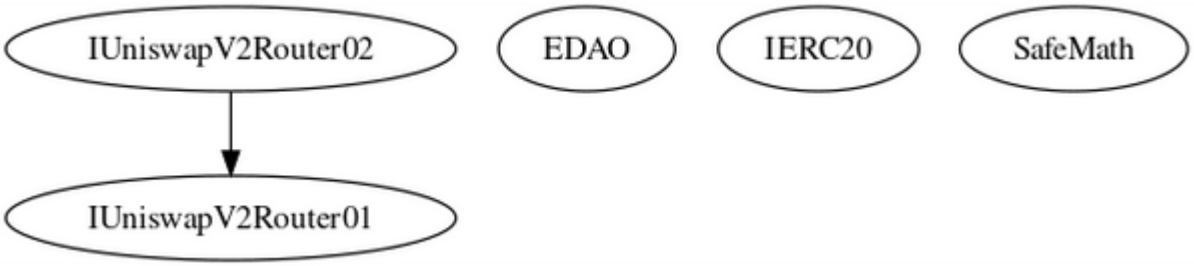
Functions Analysis

Contract	Type	Bases		
	Function Name	**Visibility**	**Mutability**	**Modifiers**
IUniswapV2Router01	Interface			
	factory	External		-
	WETH	External		-
IUniswapV2Router02	Interface	IUniswapV2Router01		
	swapExactTokensForTokensSupportingFeeOnTransferTokens	External	✓	-
EDAO	Implementation			
		Public	✓	-
	Reduceproduction	Public		-
	register	External	✓	-
	deposit	External	✓	-
	depositBySplit	External	✓	-
	transferBySplit	External	✓	-
	lotteryBet	External	✓	-
	withdraw	External	✓	-
	UsdtForERC20	Private	✓	
	UsdtForERC20toblack	Private	✓	
	distributePoolRewards	External	✓	-
	_deposit	Private	✓	
	_distributeDeposit	Private	✓	
	_updateLevel	Private	✓	
	_unfreezeCapitalOrReward	Private	✓	

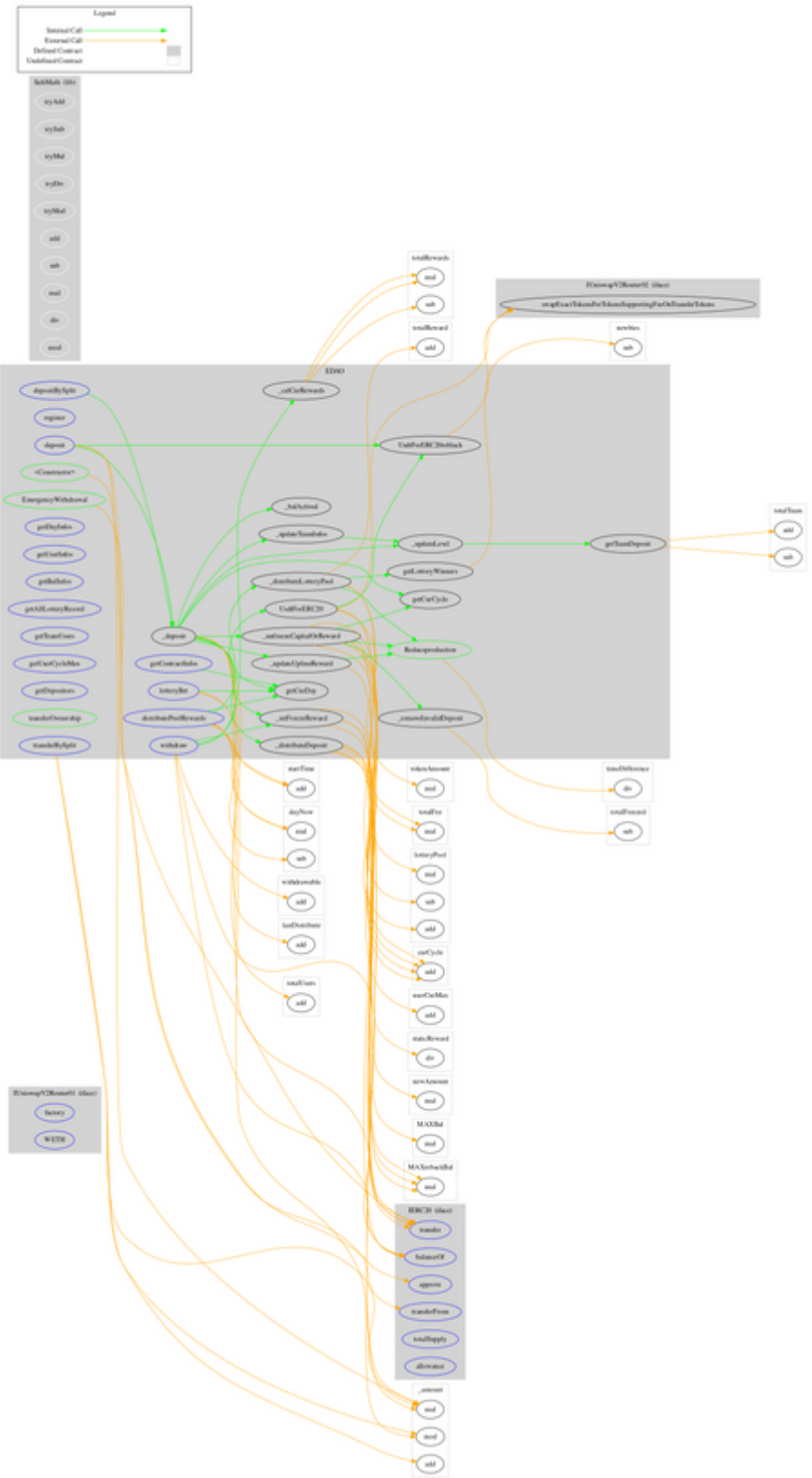
	_removeInvalidDeposit	Private	✓	
	_updateTeamInfos	Private	✓	
	_updateUplineReward	Private	✓	
	_balActived	Private	✓	
	_setFreezeReward	Private	✓	
	_calCurRewards	Private		
	_distributeLotteryPool	Private	✓	
	getLottoryWinners	Public		-
	getTeamDeposit	Public		-
	getCurDay	Public		-
	getCurCycle	Public		-
	getDayInfos	External		-
	getUserInfos	External		-
	getBalInfos	External		-
	getAllLotteryRecord	External		-
	getTeamUsers	External		-
	getUserCycleMax	External		-
	getDepositors	External		-
	getContractInfos	External		-
	transferOwnership	Public	✓	-
	EmergencyWithdrawal	Public	✓	-
IERC20	Interface			
	totalSupply	External		-
	balanceOf	External		-
	transfer	External	✓	-
	allowance	External		-
	approve	External	✓	-
	transferFrom	External	✓	-

SafeMath	Library			
	tryAdd	Internal		
	trySub	Internal		
	tryMul	Internal		
	tryDiv	Internal		
	tryMod	Internal		
	add	Internal		
	sub	Internal		
	mul	Internal		
	div	Internal		
	mod	Internal		
	sub	Internal		
	div	Internal		
	mod	Internal		

Inheritance Graph



Flow Graph


Legend
Internal Call
External Call
Defined Contract
Undefined Contract
SafeMath (lib)
try Add
try Sub
try Mul
try Div
try Mod
add
sub
mul
div
mod
EDO
deposit By Split
register
deposit
<Constructor>
Emergency Withdrawal
getDayIndex
getUserIndex
getBalInfo
getAllLotteryRecord
getTeamUsers
getUserCycleMax
getDepositors
transferOwnership
transferBySplit
_setCycleRewards
UnliFeeERC20toMark
_updateLevel
getTeamDeposit
totalTeam
getLotteryWinners
getUserCycle
UnliFeeERC20
_setExecuteCapitalOfReward
Reduceproduction
_updateUplineReward
_deposit
getContractInfo
lotteryBet
distributePoolRewards
withdraw
getCycDay
_setExecuteReward
_distributeDeposit
_removeInvalidDeposit
startTime
dayNow
withdrawable
lastDistribute
totalUsers
tokenAmount
totalFee
lotteryPool
cutCycle
userCycMax
statsReward
newAmount
MAXBid
MAXtotalBid
IERC20 (iface)
transfer
balanceOf
approve
transferFrom
totalSupply
allowance
_amount
IUniswapV2Router01 (iface)
factory
WETH
IUniswapV2Router02 (iface)
swapExactTokensForTokensSupportingFeeOnTransferTokens
newbies
totalRewards
totalReward
timeDifference
totalForward

Summary

BFC Smart Contract implements a lottery mechanism. This audit investigates security issues, business logic concerns, and potential improvements.

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