Cyberscope

Audit Report

Lotostake

August 2024

Network	BSC
Address	0xbD7677918024E9a727B5C82af12fE14E8842CB51
Audited by	© cyberscope

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

Explorer	https://bscscan.com/address/0xbd7677918024e9a727b5c82af12fe14e8842cb51

Audit Updates

Initial Audit	05 Aug 2024

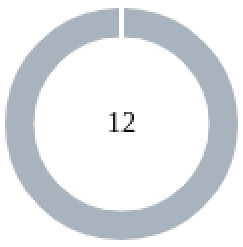
Source Files

Filename	SHA256
lottery.sol	b21803259ccd0a39b31aef94262126a4d39f36ec223b6d45822b6b156fad8e5c
Ownable.sol	eae0ce4dfc7e904c7f2b709f780b839aa51edc523a70a219004ac64ba56ea223
IERC20.sol	a200535e4e8488753f8aa68e1399e896e2b4d9b0fb9bbcdd146c816b3373837a
Context.sol	a9befa7165a2e3f7c5f8fb3c1e52c875c7e91a6da37e48dccfbd46fb9a8f2d9d

Overview

This report provides an assessment of the Lottery.sol contract, deployed on the BSC network at address "0xbD7677918024E9a727B5C82af12fE14E8842CB51". The contract facilitates the creation, management, and execution of lottery events, including ticket sales and winner selection through a pseudo-randomized process. Notable features include prize structuring, ticket ownership tracking, and lottery information querying. The audit evaluated the contract's security, verified its business logic, and examined performance to ensure secure, risk-free, and efficient operations.

Findings Breakdown



●	Critical	0
●	Medium	0
●	Minor / Informative	12

	Severity	Unresolved	Acknowledged	Resolved	Other
●	Critical	0	0	0	0
●	Medium	0	0	0	0
●	Minor / Informative	12	0	0	0

Diagnostics

● Critical ● Medium ● Minor / Informative

Severity	Code	Description	Status
●	CO	Code Optimization	Unresolved
●	CCR	Contract Centralization Risk	Unresolved
●	MVN	Misleading Variables Naming	Unresolved
●	MC	Missing Check	Unresolved
●	MEE	Missing Events Emission	Unresolved
●	MU	Modifiers Usage	Unresolved
●	RV	Randomization Vulnerability	Unresolved
●	SVI	Status Variable Inconsistencies	Unresolved
●	L04	Conformance to Solidity Naming Conventions	Unresolved
●	L08	Tautology or Contradiction	Unresolved
●	L16	Validate Variable Setters	Unresolved
●	L19	Stable Compiler Version	Unresolved

CO - Code Optimization

Criticality	Minor / Informative
Location	lottery.sol#L187
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.

Specifically, the `findPlayer()` function contains several opportunities for optimization, particularly in reducing unnecessary iterations and redundant operations, which can lead to increased gas consumption.

The function currently iterates from `ticketNumber` down to `0`, but since `ticketNumber` is required to be greater than or equal to `100000`, any iterations within the range `[0, 99999]` are redundant. This can cause the function to consume more gas than necessary, even when handling a relatively small number of tickets.

Additionally, the code includes a range search that checks if the `ticketNumber` falls within a specific range for the identified address. However, since `startTicketNumber` maps the beginning of a ticket range directly to its owner, this search is unnecessary. The wallet identified by `lottery.startTicketNumber[start]` will always be the owner of the ticket, making the range search redundant.

Finally, the loop in the function is designed to stop when `start == 0`, therefore the segment that breaks the loop when `start == 0` is unnecessary. Additionally, Solidity automatically returns `address(0)` for any uninitialized variable, thus explicitly setting the returned value of this case as `address(0)` is redundant.

```
function findPlayer(uint256 lotteryId, uint256 ticketNumber) public
view returns (address){
    require(lotteryId >= 0, "Lottery is required");
    require(ticketNumber >= 100000, "ticketNumber is required");
    ...
    for (uint256 start = ticketNumber; start >= 0; start--) {
        ...
        if (start == 0)
            break;
        }
        return address(0);
}
```

```
Range[] memory ranges = lottery.players[wallet];

for (uint256 i = 0; i < ranges.length; i++) {
if (ticketNumber >= ranges[i].start && ticketNumber <= ranges[i].end) {
...
}
```

```
...
if (start == 0)
    break;
}
    return address(0);
```

Recommendation

The team is advised to take these segments into consideration 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.

CCR - Contract Centralization Risk

Criticality	Minor / Informative
Location	lottery.sol#L63,L67,L95,L154
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. In this scenario, the owner of the smart contract possesses extensive rights that may affect the lottery's process and configuration. Specifically, the owner can influence this through the `setPaymentReceiver()` , `setBusdTokenAddress()` , `updateLottery()` and `pickWinners()` functions.

```
function setPaymentReceiver(address _paymentReceiver) public onlyOwner
{
         paymentReceiver = _paymentReceiver;
}
```

```
function setBusdTokenAddress(address _busdTokenAddress) public
onlyOwner {
         busdTokenAddress = _busdTokenAddress;
}
```

```
function updateLottery(uint256 lotteryId, uint256 ticketPrice, uint256
startDate, uint256 endDate, uint256 status, Prize[] memory prizes)
public onlyOwner {
        ...
    }
```

```
function pickWinners(uint256 lotteryId) public onlyOwner {
         ...
}
```

In particular, `updateLottery()` can be used to alter the specifics of the lottery once it has been initialized and due to pseudo-random design `pickWinners()` can be called by the owner at a block where `block.prevrandao` favours an expected outcome.

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.

MVN - Misleading Variables Naming

Criticality	Minor / Informative
Location	lottery.sol#L33
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 mapping `players` within the `LotteryDetails` structure is used to store the ranges assigned for each player. However, the naming of the mapping implies a different association.

```
mapping(address => Range[]) players;
```

Similarly, the `setBusdTokenAddress()` function may set the state variable to an address that is not the Busd token address. The `Prize` structure potentially exhibits the same issue, as it only contains an ID and a string without any reference to a monetary reward. It is worth noting that the contract does not include a reward distribution mechanism.

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.

MC - Missing Check

Criticality	Minor / Informative
Location	lottery.sol#L130,L154,L181,L207
Status	Unresolved

Description

The contract is processing variables that have not been properly sanitized and checked that they form the proper shape. These variables may produce vulnerability issues. Specifically, the functions `buyTickets()`, `pickWinners()`, `findPlayer()` and `getLotteryInfo()` accept a `lotteryId` as input and proceed their execution without verifying whether this is a valid id. Contrary, `updateLottery()` performs this check to confirm the id exists. Nevertheless, `updateLottery()` does not check whether the lottery's end date is past the start date, the Prizes are non-trivial and the ticket price is non-zero. These are checks implemented in the `createLottery()` function.

```
function createLottery(uint256 ticketPrice, uint256 startDate, uint256
endDate, Prize[] memory prizes) public onlyOwner {
        require(ticketPrice > 0, "Ticket price must be greater than
zero");
        require(startDate < endDate, "Start date must be before end
date");
        require(prizes.length > 0, "There must be at least one prize");
        ...
}
```

```
function updateLottery(uint256 lotteryId, uint256 ticketPrice, uint256
startDate, uint256 endDate, uint256 status, Prize[] memory prizes)
public onlyOwner {
        require(lotteryId < nextLotteryId, "Lottery does not exist");

        LotteryDetails storage lottery = lotteries[lotteryId];
        require(lottery.status != 2, "Closed lotteries cannot be
updated");
        ...
}
```

Recommendation

The team is advised to properly check the variables according to the required specifications.

MEE - Missing Events Emission

Criticality	Minor / Informative
Location	lottery.sol#L63,L67
Status	Unresolved

Description

The contract performs actions and state mutations from external methods that do not result in the emission of events. Emitting events for significant actions is important as it allows external parties, such as wallets or dApps, to track and monitor the activity on the contract. Without these events, it may be difficult for external parties to accurately determine the current state of the contract.

```
function setPaymentReceiver(address _paymentReceiver) public onlyOwner
{
    paymentReceiver = _paymentReceiver;
}
```

```
function setBusdTokenAddress(address _busdTokenAddress) public
onlyOwner {
    busdTokenAddress = _busdTokenAddress;
}
```

Recommendation

It is recommended to include events in the code that are triggered each time a significant action is taking place within the contract. These events should include relevant details such as the user's address and the nature of the action taken. By doing so, the contract will be more transparent and easily auditable by external parties. It will also help prevent potential issues or disputes that may arise in the future.

MU - Modifiers Usage

Criticality	Minor / Informative
Location	lottery.sol#L182,L217
Status	Unresolved

Description

The contract is using repetitive statements on some methods to validate some preconditions. In Solidity, the form of preconditions is usually represented by the modifiers. Modifiers allow you to define a piece of code that can be reused across multiple functions within a contract. This can be particularly useful when you have several functions that require the same checks to be performed before executing the logic within the function.

```
require(lotteryId >= 0, "Lottery is required");
```

Recommendation

The team is advised to use modifiers since it is a useful tool for reducing code duplication and improving the readability of smart contracts. By using modifiers to perform these checks, it reduces the amount of code that is needed to write, which can make the smart contract more efficient and easier to maintain.

RV - Randomization Vulnerability

Criticality	Minor
Location	lottery.sol#L177
Status	Unresolved

Description

The contract is using an on-chain technique in order to determine random numbers. The blockchain runtime environment is fully deterministic, as a result, the pseudo-random numbers could be predicted.

```
function random(uint256 lotteryId, uint256 nonce) internal view returns
(uint256) {
return uint256(keccak256(abi.encodePacked(block.prevrandao,
block.timestamp,
       lotteryId, nonce)));
}
```

Furthermore, the current implementation of pseudo-randomness in the selection of winners, permits the same ticket to win multiple prizes of the same lottery. Unless this is part of the intended business logic the team should consider revising the implementation of the `pickWinners()` function.

```
function pickWinners(uint256 lotteryId) public onlyOwner {
  ...
       for (uint256 i = 0; i < lottery.prizes.length; i++) {
            uint256 winningTicketIndex = random(lotteryId, i) %
(lottery.nextTicketNumber - 100000);
            uint256 winningTicket = 100000 + winningTicketIndex;

            lottery.prizeWinners.push(PrizeWinner({
                prizeId: lottery.prizes[i].id,
                winningTicket: winningTicket
            }));
       }
  …
}
```

Recommendation

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

SVI - Status Variable Inconsistencies

Criticality	Minor / Informative
Location	lottery.sol#L26
Status	Unresolved

Description

The contract utilizes a `status` variable to track the state of a lottery. The `status` variable is declared as a unit256 variable and it is using numeric values to indicate the state of the lottery: 0 for "planned," 1 for "ongoing," and 2 for "closed." While this approach is effective, it can be less readable and more error-prone. Solidity supports enumerables, which are useful to model choice and keep track of state. Using an enum for `status` would enhance readability and maintainability by providing named constants for each state.

```
uint256 status; // 0: planned, 1: ongoing, 2: closed
```

Furthermore, the use of `status` in the contract exhibits several inconsistencies. Specifically, in the `createLottery()` function, `status` is always set as 'ongoing' for a new lottery, nevertheless it is possible to set the start date at a future time. In this case, setting `status` as 'ongoing' instead of 'planned' does not reflect reality.

In addition, calling `getLotteryInfo()` with a lotteryId that has not been set, returns a value for `status` equal to 0, which according to the comments indicates a planned lottery.

```
function createLottery(uint256 ticketPrice, uint256 startDate, uint256
endDate, Prize[] memory prizes) public onlyOwner {
        ...
        newLottery.status = 1;
        ...
    }
```

```
function getLotteryInfo(uint256 lotteryId) public view returns (
        uint256 ticketPrice,
        uint256 nextTicketNumber,
        uint256 playerCount,
        uint256 status,
        uint256 startDate,
        uint256 endDate,
        PrizeWinner[] memory prizeWinners,
        Prize[] memory prizes
    ) {
        require(lotteryId >= 0, "Lottery is required");

        LotteryDetails storage lottery = lotteries[lotteryId];

        return (
            lottery.ticketPrice,
            lottery.nextTicketNumber,
            lottery.playerCount,
            lottery.status,
            lottery.startDate,
            lottery.endDate,
            lottery.prizeWinners,
            lottery.prizes
        );
    }
```

Recommendation

The team should consider defining an enum named `status` with values "planned", "ongoing", and "closed" to make the code more self-explanatory and reduce the risk of misinterpreting or misusing numeric values. Additionally, the team should address inconsistencies by ensuring that `status` accurately reflects the lottery's state, especially regarding future start dates and uninitialized lotteries.

L04 - Conformance to Solidity Naming Conventions

Criticality	Minor / Informative
Location	lottery.sol#L63,67
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 _paymentReceiver
address _busdTokenAddress
```

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.

L08 - Tautology or Contradiction

Criticality	Minor / Informative
Location	lottery.sol#L182,187,217
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.

```
require(lotteryId >= 0, "Lottery is required")
start >= 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.

L16 - Validate Variable Setters

Criticality	Minor / Informative
Location	lottery.sol#L51,64,68
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.

```
busdTokenAddress = _busdTokenAddress
paymentReceiver = _paymentReceiver
```

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	lottery.sol#L3
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 ^0.8.26;
```

Recommendation

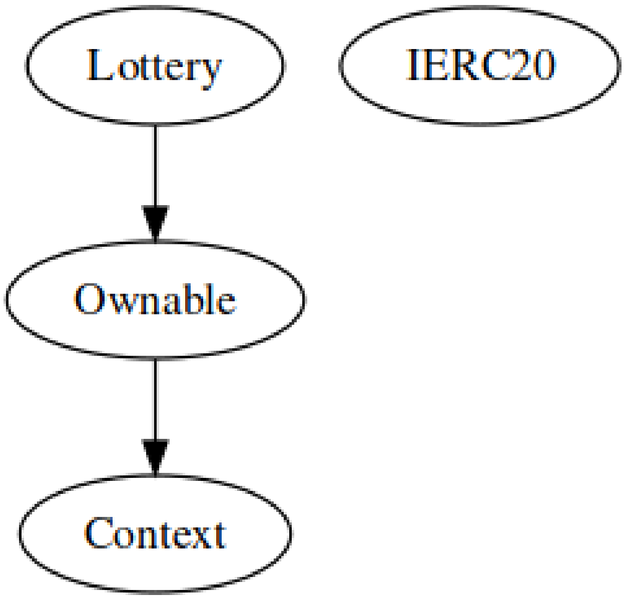
The team is advised to lock the pragma to ensure the stability of the codebase. The locked pragma version ensures that the contract will not be deployed with an unexpected version. An unexpected version may produce vulnerabilities and undiscovered bugs. The compiler should be configured to the lowest 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.

Functions Analysis

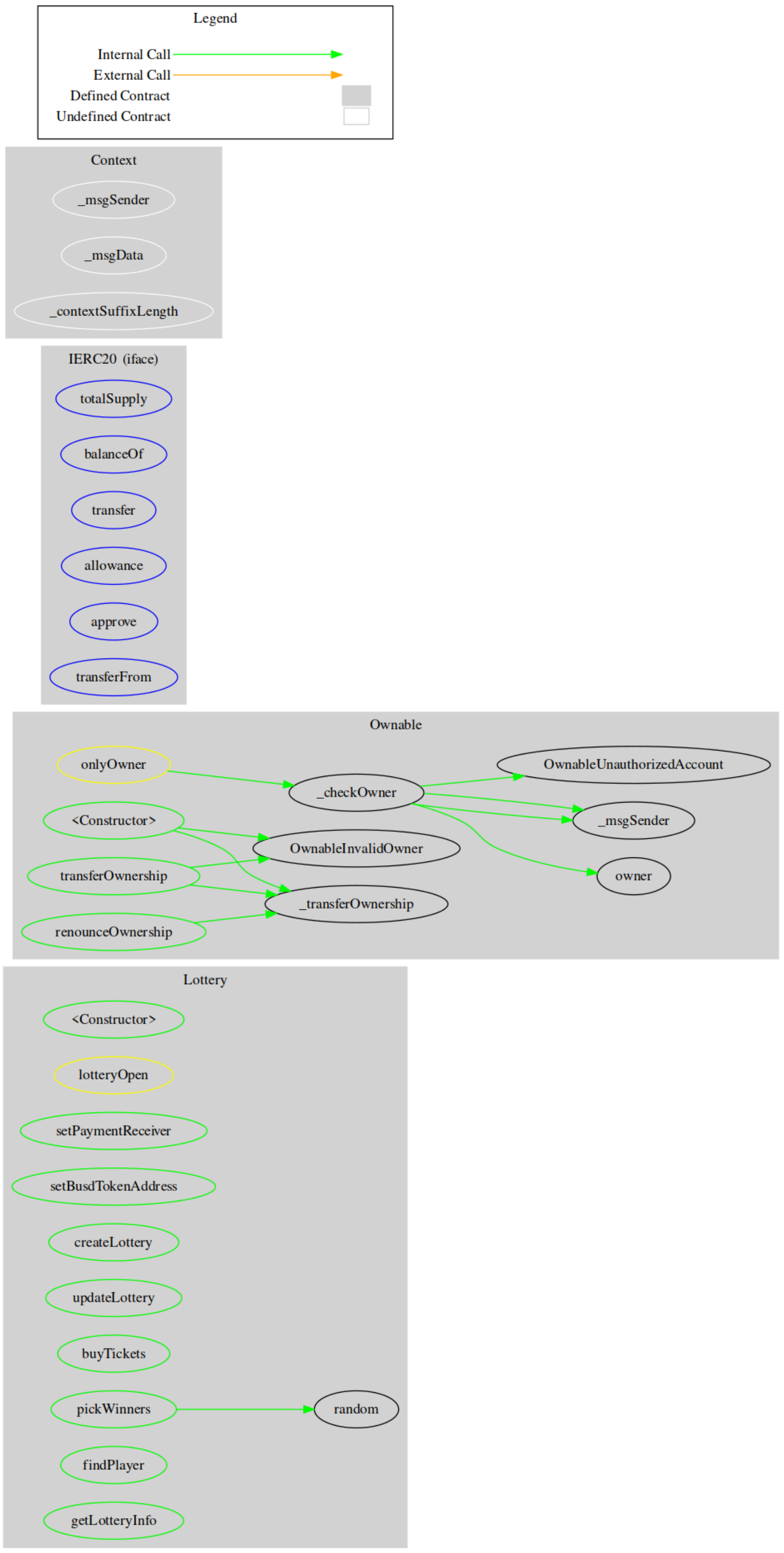
Contract	Type	Bases		
	Function Name	**Visibility**	**Mutability**	**Modifiers**
Lottery	Implementation	Ownable		
		Public	✓	Ownable
	setPaymentReceiver	Public	✓	onlyOwner
	setBusdTokenAddress	Public	✓	onlyOwner
	createLottery	Public	✓	onlyOwner
	updateLottery	Public	✓	onlyOwner
	buyTickets	Public	✓	lotteryOpen
	pickWinners	Public	✓	onlyOwner
	random	Internal		
	findPlayer	Public		-
	getLotteryInfo	Public		-
Ownable	Implementation	Context		
		Public	✓	-
	owner	Public		-
	_checkOwner	Internal		
	renounceOwnership	Public	✓	onlyOwner
	transferOwnership	Public	✓	onlyOwner
	_transferOwnership	Internal	✓	

IERC20	Interface			
	totalSupply	External		-
	balanceOf	External		-
	transfer	External	✓	-
	allowance	External		-
	approve	External	✓	-
	transferFrom	External	✓	-
Context	Implementation			
	_msgSender	Internal		
	_msgData	Internal		
	_contextSuffixLength	Internal		

Inheritance Graph



Flow Graph


Legend
Internal Call
External Call
Defined Contract
Undefined Contract
Context
_msgSender
_msgData
_contextSuffixLength
IERC20 (iface)
totalSupply
balanceOf
transfer
allowance
approve
transferFrom
Ownable
onlyOwner
_checkOwner
OwnableUnauthorizedAccount
<Constructor>
_msgSender
OwnableInvalidOwner
transferOwnership
owner
_transferOwnership
renounceOwnership
Lottery
<Constructor>
lotteryOpen
setPaymentReceiver
setBusdTokenAddress
createLottery
updateLottery
buyTickets
pickWinners
random
findPlayer
getLotteryInfo

Summary

The contract implements a lottery mechanism where the owner can create new lotteries, and users can buy tickets for each lottery. Upon concluding a lottery, a pseudo-random method identifies the winning ticket, and the winning address is permanently recorded on the blockchain.

The contract compiled without errors or critical issues, although minor/informative issues were identified to enhance performance and decentralisation. The contract owner has access to several administrative functions that can modify or impact an active lottery. The team is advised to explore solutions that enhance decentralisation, particularly regarding ownership on specific functionalities.

This audit examines security issues, business logic concerns, and potential performance improvements.

Disclaimer

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