Cyberscope

Audit Report

Lotostake

December 2024

Network BSC

Address 0x78aB2E30e8C7ee6ED6019E091001aaa5FD1C2fd4

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/0x78ab2e30e8c7ee6ed6019e091001aaa5fd1c2fd4

Audit Updates

Initial Audit	05 Aug 2024 https://github.com/cyberscope-io/audits/blob/main/3-lst/v1/audit.pdf
Corrected Phase 2	29 Nov 2024

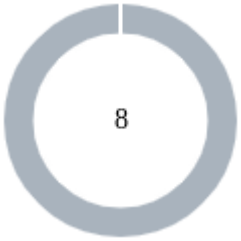
Source Files

Filename	SHA256
Lottery.sol	56a607da13ba0168c63447a3374102724e862522b9d86940c1c09cf472b5fced

Overview

This report provides an assessment of the Lottery.sol contract, deployed on the BSC network at address "0x78ab2e30e8c7ee6ed6019e091001aaa5fd1c2fd4". The contract facilitates the creation, management, and execution of lotteries, including ticket sales and winner selection through an integration with Chainlink’s VRF engine. Notable features include prize structuring, ticket ownership tracking, and lottery information querying are included in the contract. 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	8

Severity	Unresolved	Acknowledged	Resolved	Other
Critical	0	0	0	0
Medium	0	0	0	0
Minor / Informative	8	0	0	0

Diagnostics

Critical Medium Minor / Informative

Severity	Code	Description	Status
●	CCR	Contract Centralization Risk	Unresolved
●	MEM	Misleading Error Messages	Unresolved
●	MC	Missing Check	Unresolved
●	L04	Conformance to Solidity Naming Conventions	Unresolved
●	L16	Validate Variable Setters	Unresolved
●	L17	Usage of Solidity Assembly	Unresolved
●	L18	Multiple Pragma Directives	Unresolved
●	L19	Stable Compiler Version	Unresolved

CCR - Contract Centralization Risk

Criticality	Minor / Informative
Location	Lottery.sol#L705,718,741,769,807,884
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 setPaymentReceiver(address newPaymentReceiver) public onlyOwner {}
function setTokenAddress(address newPaymentToken) public onlyOwner {}
function setVRFConfig(
        bytes32 newKeyHash,
        uint256 newSubscriptionId,
        uint16 newRequestConfirmations,
        uint32 newCallbackGasLimit
    ) public onlyOwner {}
function createLottery(
        uint256 ticketPrice,
        uint256 startDate,
        uint256 endDate,
        Prize[] memory prizes,
        uint256 firstTicketNumber
    ) public lotteryDataAreValid(ticketPrice, startDate, endDate, prizes)
onlyOwner {}
function updateLottery(
        uint32 lotteryId,
        uint256 ticketPrice,
        uint256 startDate,
        uint256 endDate,
        Status status,
        Prize[] memory prizes
    ) public lotteryExists(lotteryId) lotteryDataAreValid(ticketPrice,
startDate, endDate, prizes) onlyOwner {}
function requestRandomWords(uint32 lotteryId, bool enableNativePayment)
public lotteryExists(lotteryId) 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.

MEM - Misleading Error Messages

Criticality	Minor / Informative
Location	Lottery.sol#L859
Status	Unresolved

Description

The contract is potentially using misleading error messages. These error messages do not accurately reflect the actual implementation, making it difficult to understand the source of the issue. In this instance, the owner has the privilege to change the paymentToken to something different than BUSD. The error message however would not reflect this change.

```
require(IERC20(paymentToken).transferFrom(msg.sender, paymentReceiver,
totalPrice), "BUSD transfer failed");
```

Recommendation

The team is advised to carefully review the error messages in order to reflect the actual source of the issues. To improve code interpretation, the team should use more specific and descriptive error messages.

MC - Missing Check

Criticality	Minor / Informative
Location	Lottery.sol#L682,718,918
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 modifier `lotteryDataAreValid` does not ensure that the `startDate` is past the current timestamp. In the case a startDate in the past is assigned, inconsistencies will arise when the status of the lottery is not yet active. Similarly, the `fulfillRandomWords` function does not verify that the number of provided `randomWords` matches the number of expected prizes.

```
modifier lotteryDataAreValid(uint256 ticketPrice, uint256 startDate, uint256
endDate, Prize[] memory prizes) {
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");
_;}
```

Additionally, the `setTokenAddress` function does not ensure consistency in decimal precision. Specifically, if the token address is updated to a token with a different decimal precision than the previous token, the payment amount expected for an existing lottery could significantly vary from what is intended.

```
function setTokenAddress(address newPaymentToken) public onlyOwner {
require(newPaymentToken != address(0), "Payment Receiver address required");
uint32 size;
assembly {
size := extcodesize(newPaymentToken)
}
require(size > 0, "Address is not a contract");
require(IERC20(newPaymentToken).totalSupply() > 0, "Not a valid ERC20
token");
paymentToken = newPaymentToken;
emit PaymentTokenChanged(paymentToken);}
```

Recommendation

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

L04 - Conformance to Solidity Naming Conventions

Criticality	Minor / Informative
Location	Lottery.sol#L469,511
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.

```
IVRFCoordinatorV2Plus public s_vrfCoordinator
address _vrfCoordinator
```

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.

L16 - Validate Variable Setters

Criticality	Minor / Informative
Location	Lottery.sol#L657
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.

```
paymentToken = newPaymentToken
```

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	Lottery.sol#L722
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 {
            size := extcodesize(newPaymentToken)
        }
```

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.

L18 - Multiple Pragma Directives

Criticality	Minor / Informative
Location	Lottery.sol#L7,87,114,215,253,269,282,352,364,531
Status	Unresolved

Description

If the contract includes multiple conflicting pragma directives, it may produce unexpected errors. To avoid this, it's important to include the correct pragma directive at the top of the contract and to ensure that it is the only pragma directive included in the contract.

```
pragma solidity ^0.8.20;
pragma solidity ^0.8.4;
pragma solidity ^0.8.0;
pragma solidity 0.8.26;
```

Recommendation

It is important to include only one pragma directive at the top of the contract and to ensure that it accurately reflects the version of Solidity that the contract is written in.

By including all required compiler options and flags in a single pragma directive, the potential conflicts could be avoided and ensure that the contract can be compiled correctly.

L19 - Stable Compiler Version

Criticality	Minor / Informative
Location	Lottery.sol#L7,215,364
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 ^0.8.0;
pragma solidity ^0.8.4;
```

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

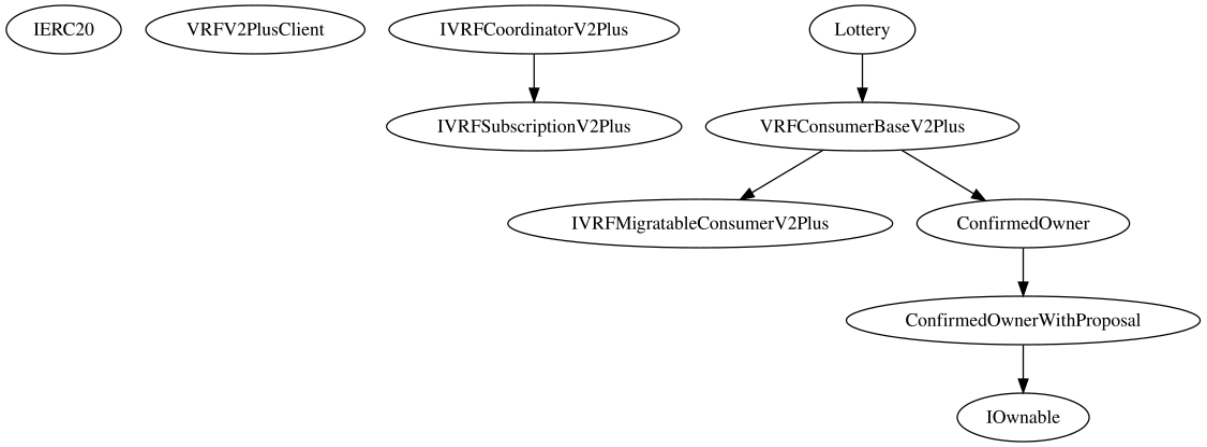
Contract	Type	Bases		
	Function Name	**Visibility**	**Mutability**	**Modifiers**
IERC20	Interface			
	totalSupply	External		-
	balanceOf	External		-
	transfer	External	✓	-
	allowance	External		-
	approve	External	✓	-
	transferFrom	External	✓	-
VRFV2PlusClient	Library			
	_argsToBytes	Internal		
IVRFSubscriptionV2Plus	Interface			
	addConsumer	External	✓	-
	removeConsumer	External	✓	-
	cancelSubscription	External	✓	-
	acceptSubscriptionOwnerTransfer	External	✓	-
	requestSubscriptionOwnerTransfer	External	✓	-
	createSubscription	External	✓	-

	getSubscription	External		-
	pendingRequestExists	External		-
	getActiveSubscriptionIds	External		-
	fundSubscriptionWithNative	External	Payable	-
IVRFCoordinatorV2Plus	Interface	IVRFSubscriptionV2Plus		
	requestRandomWords	External	✓	-
IVRFMigratableConsumerV2Plus	Interface			
	setCoordinator	External	✓	-
IOwnable	Interface			
	owner	External	✓	-
	transferOwnership	External	✓	-
	acceptOwnership	External	✓	-
ConfirmedOwnerWithProposal	Implementation	IOwnable		
		Public	✓	-
	transferOwnership	Public	✓	onlyOwner
	acceptOwnership	External	✓	-
	owner	Public		-
	_transferOwnership	Private	✓	

	_validateOwnership	Internal		
ConfirmedOwner	Implementation	ConfirmedOwnerWithProposal		
		Public	✓	ConfirmedOwnerWithProposal
VRFConsumerBaseV2Plus	Implementation	IVRFMigratableConsumerV2Plus, ConfirmedOwner		
		Public	✓	ConfirmedOwner
	fulfillRandomWords	Internal	✓	
	rawFulfillRandomWords	External	✓	-
	setCoordinator	External	✓	onlyOwnerOrCoordinator
Lottery	Implementation	VRFConsumerBaseV2Plus		
		Public	✓	VRFConsumerBaseV2Plus
	setPaymentReceiver	Public	✓	onlyOwner
	setTokenAddress	Public	✓	onlyOwner
	setVRFConfig	Public	✓	onlyOwner
	createLottery	Public	✓	lotteryDataAreValid onlyOwner
	updateLottery	Public	✓	lotteryExists lotteryDataAreValid onlyOwner

	buyTickets	Public	✓	lotteryExists lotteryOpen
	requestRandomWords	Public	✓	lotteryExists onlyOwner
	fulfillRandomWords	Internal	✓	
	getTicketOwners	Public		lotteryExists
	getLotteryInfo	Public		lotteryExists

Inheritance Graph



Flow Graph


Legend
Internal Call
External Call
Defined Contract
Undefined Contract
ConfirmedOwner
<Constructor>
ConfirmedOwnerWithProposal
onlyOwner
_validateOwnership
transferOwnership
acceptOwnership
owner
_transferOwnership
<Constructor>
VRFConsumerBaseV2Plus
<Constructor>
ZeroAddress
setCoordinator
OnlyCoordinatorCanFulfill
rawFulfillRandomWords
fulfillRandomWords
onlyOwnerOrCoordinator
owner
OnlyOwnerOrCoordinator
IOwnable (iface)
owner
transferOwnership
acceptOwnership
IVRFMigratableConsumerV2Plus (iface)
setCoordinator
IVRFSubscriptionV2Plus (iface)
addConsumer
removeConsumer
cancelSubscription
acceptSubscriptionOwnerTransfer
requestSubscriptionOwnerTransfer
createSubscription
getSubscription
pendingRequestExists
getActiveSubscriptionIds
fundSubscriptionWithNative
Lottery
<Constructor>
lotteryExists
lotteryDataAreValid
lotteryOpen
setPaymentReceiver
setTokenAddress
setVRFConfig
createLottery
updateLottery
buyTickets
requestRandomWords
fulfillRandomWords
getTicketOwners
getLotteryInfo
VRFV2PlusClient (lib)
_argsToBytes
RandomWordsRequest
ExtraArgsV1
IVRFCoordinatorV2Plus (iface)
requestRandomWords
IERC20 (iface)
totalSupply
balanceOf
transfer
allowance
approve
transferFrom

Summary

The contract implements a lottery system where the owner can create lotteries and users can buy tickets. Chainlink’s VRF is used to determine the winners. The owner can modify active lotteries and initiate winner selection. The contract compiled without issues, with only minor performance and consistency improvements noted. This audit reviews security, business logic, and performance.

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