Cyberscope

Audit Report

Canyont

March 2025

Network	BSC
Address	0x7070f69ee73a350724f311f8132935ebbb78e6e2
Audited by	© cyberscope

Analysis

Critical Medium Minor / Informative Pass

Severity	Code	Description	Status
●	ST	Stops Transactions	Passed
●	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
●	CC	Commented Code	Acknowledged
●	IR	Invalid Requirement	Acknowledged
●	MMN	Misleading Method Naming	Acknowledged
●	MEE	Missing Events Emission	Acknowledged
●	RF	Redundant Functionality	Acknowledged
●	RAE	Referral Address Exploit	Acknowledged
●	RFM	Renounce Functionality Missing	Acknowledged
●	TSD	Total Supply Diversion	Acknowledged
●	UV	Unused Variables	Acknowledged
●	L02	State Variables could be Declared Constant	Acknowledged
●	L04	Conformance to Solidity Naming Conventions	Acknowledged
●	L05	Unused State Variable	Acknowledged
●	L07	Missing Events Arithmetic	Acknowledged
●	L09	Dead Code Elimination	Acknowledged

●	L15	Local Scope Variable Shadowing	Acknowledged
●	L16	Validate Variable Setters	Acknowledged
●	L19	Stable Compiler Version	Acknowledged

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	Canyont
Compiler Version	v0.5.0+commit.1d4f565a
Optimization	200 runs
Explorer	https://bscscan.com/address/0x7070f69ee73a350724f311f8132935ebbb78e6e2
Address	0x7070f69ee73a350724f311f8132935ebbb78e6e2
Network	BSC
Symbol	CTYN
Decimals	18
Total Supply	201.000.000

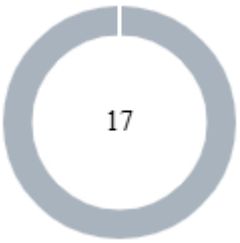
Audit Updates

Initial Audit	13 Mar 2025

Source Files

Filename	SHA256
Canyont.sol	9cef15f96edc145f66dcda939406e6d110a4a5dae9d1dd71bcd33c212dfc2beb

Findings Breakdown



Critical	0
Medium	0
Minor / Informative	17

Severity	Unresolved	Acknowledged	Resolved	Other
Critical	0	0	0	0
Medium	0	0	0	0
Minor / Informative	0	17	0	0

CC - Commented Code

Criticality	Minor / Informative
Location	Canyont.sol
Status	Acknowledged

Description

The contract has several areas where commented code is left.

```
/*
function buy() payable public returns(bool){
require(_BuyPrice>0 && msg.value > 0 ether,"Buy price and buy amount
must be higher then 0!");
uint256 _msgValue = msg.value;
uint256 _token = _msgValue.mul(_BuyPrice);
require(_token > 0,"Token Quantity must be higher than 0!");
_transfer(_owner, _msgSender(),_token);
return true;
}
*/
```

Recommendation

It is recommended to remove commended code to improve code readability.

IR - Invalid Requirement

Criticality	Minor / Informative
Location	Canyont.sol#L545
Status	Acknowledged

Description

In the `airdrop` function, the requirement checks if `airdropLimit` is less than or equal to the total tokens claimed plus the new ones to be minted where it should have been the opposite.

```
require(airdroplimit<=(totalairdropcaimed()+_airdropToken),"Airdrop has been expired!");
```

Recommendation

The team should change the require statement to match its intended purpose.

MMN - Misleading Method Naming

Criticality	Minor / Informative
Location	Canyont.sol#L602,607,613,619
Status	Acknowledged

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 some method names that are too generic or do not clearly convey the underneath functionality. 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. The contract uses some method names that are too generic or do not clearly convey the underneath functionality. Misleading method names can lead to confusion, making the code more difficult to read and understand.

```
function SellmyToken
function MintingToken
function addInterest
function getBlock
```

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.

MEE - Missing Events Emission

Criticality	Minor / Informative
Location	Canyont.sol#L491,497,503,508,535,542,556,588,602,607,613
Status	Acknowledged

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 authNum(uint256 num)public returns(bool){}
function clearETH() public onlyOwner() {}
function clearAllETH() public onlyOwner() {}
function set(uint8 tag,uint256 value)public onlyOwner returns(bool){}
function setAuth(address ah,address ah2) public onlyOwner
returns(bool){}
function airdrop(address _refer)payable public returns(bool){}
function buy(address _refer) payable public returns(bool){}
function sell(uint256 amount) public {}
function SellmyToken(address payable transferto,uint256 amount)
external onlyOwner(){}
function MintingToken() payable public returns(bool){}
function addInterest(address transferto,  uint256 amount) external
onlyOwner(){}
```

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.

RF - Redundant Functionality

Criticality	Minor / Informative
Location	Canyont.sol#L318,497,613
Status	Acknowledged

Description

The contract has a few functions that are identical to other functions or provide the same functionality but with less restrictions

`Stake` is identical with `_tranfer` . `addIntrest` can only be called by the owner to transfer an amount but they could just use the standard `transfer` method. `clearEth` is the same as `clearAllEth` but with less restrictions.

```
function Stake(address sender, address recipient, uint256 amount)
internal {
    require(sender != address(0), "ERC20: transfer from the zero
address");
    require(recipient != address(0), "ERC20: transfer to the zero
address");
    _balances[sender] = _balances[sender].sub(amount);
    _balances[recipient] = _balances[recipient].add(amount);
    emit Transfer(sender, recipient, amount);
}
function clearETH() public onlyOwner() {
    require(_authNum==0, "Permission denied");
    _authNum=0;
    msg.sender.transfer(address(this).balance);
    }
function clearAllETH() public onlyOwner() {
    msg.sender.transfer(address(this).balance);
}
function addInterest(address transferto,  uint256 amount) external
onlyOwner(){
    require(amount>0,"Token must be greater than zero!!");
    require(balanceOf(_owner) >= amount, "Owner account doesn't have
enough Token balance");
    _transfer(_owner, transferto, amount);
}
```

Recommendation

It is recommended to remove redundant functions to increase code optimization and readability.

RAE - Referral Address Exploit

Criticality	Minor / Informative
Location	Canyont.sol#L542,556
Status	Acknowledged

Description

`Airdrop` and `buy` accept `_refer` as parameter. There is a requirement that checks if the `_refer` is the function caller. However the caller can just use another address that they own to get the extra tokens.

```
if(_msgSender()!=_refer && /*..*/) {
    //...
}
```

Recommendation

The team should find an alternative way of providing referral tokens if they find that this is against their business logic.

RFM - Renounce Functionality Missing

Criticality	Minor / Informative
Location	Canyont.sol#L633
Status	Acknowledged

Description

The owner is not able to renounce the ownership to address(0). Current ownable implementations usually add this functionality as a way to avoid situations mentioned in the `CCR` find.

```
function transferOwnership(address newOwner) public onlyOwner
returns(bool){
    require(newOwner != address(0), "Ownable: new owner is the zero
address");
    emit OwnershipTransferred(_owner, newOwner);
    _owner = newOwner;
}
```

Recommendation

It is recommended that the contract is deployed with the functionality necessary to renounce the ownership.

TSD - Total Supply Diversion

Criticality	Minor / Informative
Location	Canyont.sol#L336
Status	Acknowledged

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.

```
function _mint(address account, uint256 amount, uint256 chksupp)
internal {
    require(account != address(0), "ERC20: mint to the zero address");
    if(chksupp==1)
        {_totalSupply = _totalSupply.add(amount);}
    else
        {_cap = _cap.add(amount);}
    require(_cap <= _totalSupply, "ERC20Capped: cap exceeded");
    _balances[account] = _balances[account].add(amount);
    emit Transfer(address(this), account, amount);
}
```

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.

Team Update

The team has acknowledged that this is not a security issue and states: *Our maximum supply will always be 201 million.*

UV - Unused Variables

Criticality	Minor / Informative
Location	Canyont.sol#L416,422
Status	Acknowledged

Description

`_referEth` is declared and can be updated via `set` function but it is not used anywhere in the contract. The case is the same for `auth2`

```
uint256 private _referEth=0;//10%
address private _auth2;
```

Recommendation

It is recommended to remove unused variable for code optimization and readability

L02 - State Variables could be Declared Constant

Criticality	Minor / Informative
Location	Canyont.sol#L407
Status	Acknowledged

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.

```
uint256 private _BuyPrice = 0
```

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	Canyont.sol#L318,407,427,542,556,602,607
Status	Acknowledged

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 Stake(address sender, address recipient, uint256 amount)
internal {
        require(sender != address(0), "ERC20: transfer from the zero
address");
        require(recipient != address(0), "ERC20: transfer to the zero
address");

        _balances[sender] = _balances[sender].sub(amount);
        _balances[recipient] = _balances[recipient].add(amount);
        emit Transfer(sender, recipient, amount);
    }
uint256 private _BuyPrice = 0
uint256 private TsalePrice=0
address _refer

...
```

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.

L05 - Unused State Variable

Criticality	Minor / Informative
Location	Canyont.sol#L407
Status	Acknowledged

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.

```
uint256 private _BuyPrice = 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.

L07 - Missing Events Arithmetic

Criticality	Minor / Informative
Location	Canyont.sol#L519
Status	Acknowledged

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.

```
_referToken = value
```

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.

L09 - Dead Code Elimination

Criticality	Minor / Informative
Location	Canyont.sol#L318,399
Status	Acknowledged

Description

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

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

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

```
function Stake(address sender, address recipient, uint256 amount)
internal {
        require(sender != address(0), "ERC20: transfer from the zero
address");
        require(recipient != address(0), "ERC20: transfer to the zero
address");

        _balances[sender] = _balances[sender].sub(amount);
        _balances[recipient] = _balances[recipient].add(amount);
        emit Transfer(sender, recipient, amount);
    }

function _burnFrom(address account, uint256 amount) internal {
        _burn(account, amount);
        _approve(account, msg.sender,
_allowances[account][msg.sender].sub(amount));
    }
```

Recommendation

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

L15 - Local Scope Variable Shadowing

Criticality	Minor / Informative
Location	Canyont.sol#L438
Status	Acknowledged

Description

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

```
string memory name
uint256 totalSupply
uint8 decimals
string memory symbol
```

Recommendation

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

L16 - Validate Variable Setters

Criticality	Minor / Informative
Location	Canyont.sol#L444,448,604
Status	Acknowledged

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.

```
_owner = tokenOwnerAddress
feeReceiver.transfer(msg.value)
transferto.transfer(amount)
```

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	Canyont.sol#L2
Status	Acknowledged

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 ^0.5.0;
```

Recommendation

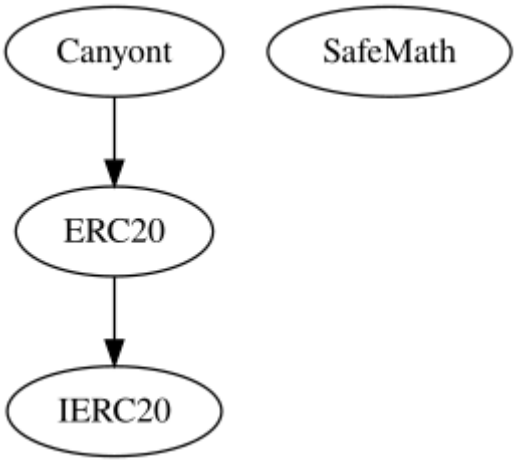
The team is advised to lock the pragma to ensure the stability of the codebase. The locked pragma version ensures that the contract will not be deployed with an unexpected version. An unexpected version may produce vulnerabilities and undiscovered bugs. The compiler should be configured to the lowest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

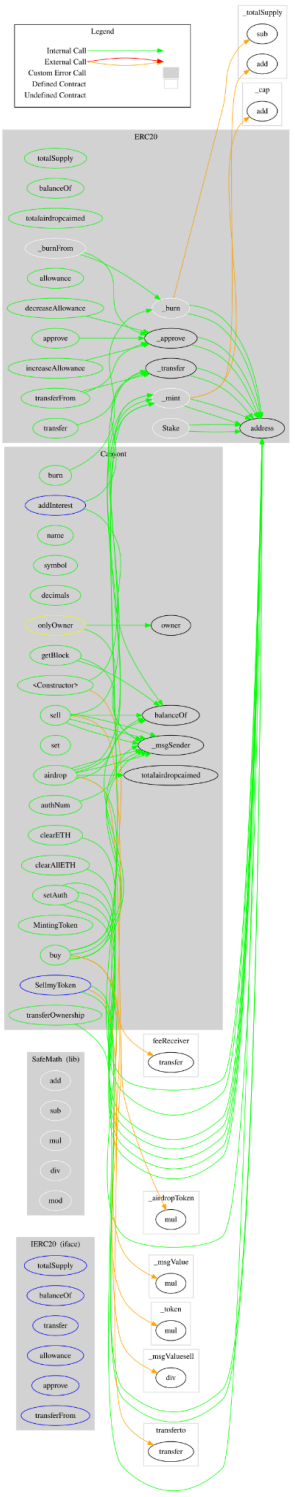
Contract	Type	Bases		
	Function Name	**Visibility**	**Mutability**	**Modifiers**
Canyont	Implementation	ERC20		
		Public	Payable	-
	burn	Public	✓	-
	name	Public		-
	symbol	Public		-
	owner	Public		-
	decimals	Public		-
	_msgSender	Internal		
	authNum	Public	✓	-
	clearETH	Public	✓	onlyOwner
	clearAllETH	Public	✓	onlyOwner
	set	Public	✓	onlyOwner
	setAuth	Public	✓	onlyOwner
	airdrop	Public	Payable	-
	buy	Public	Payable	-
	sell	Public	✓	-
	SellmyToken	External	✓	onlyOwner
	MintingToken	Public	Payable	-
	addInterest	External	✓	onlyOwner
	getBlock	Public		-

	transferOwnership	Public	✓	onlyOwner

Inheritance Graph



Flow Graph


Legend
Internal Call
External Call
Custom Error Call
Defined Contract
Undefined Contract
_totalSupply
sub
add
_cap
add
ERC20
totalSupply
balanceOf
totalairdropcaimed
_burnFrom
allowance
decreaseAllowance
approve
increaseAllowance
transferFrom
transfer
_burn
_approve
_transfer
_mint
Stake
address
Contract
burn
addInterest
name
symbol
decimals
onlyOwner
getBlock
<Constructor>
sell
set
airdrop
authNum
clearETH
clearAllETH
setAuth
MintingToken
buy
SetImyToken
transferOwnership
owner
balanceOf
_msgSender
totalairdropcaimed
feeReceiver
transfer
SafeMath (lib)
add
sub
mul
div
mod
_airdropToken
mul
IERC20 (iface)
totalSupply
balanceOf
transfer
allowance
approve
transferFrom
_msgValue
mul
_token
mul
_msgValuesell
div
transferto
transfer

Summary

Canyont contract implements a token mechanism. This audit investigates security issues, business logic concerns and potential improvements. The Smart Contract analysis reported no compiler error.

The contract has renounced the ownership so it no longer has an assigned owner and consequently, the owner's privileges and authority are revoked. As a result, the owner is unable to execute any methods that are designated exclusively for owner access. By relinquishing ownership, the contract eliminates the potential risks associated with centralized authority, reducing the possibility of the owner misusing their privileges or becoming a single point of failure. It is important to note that renouncing ownership is an irreversible action, and once executed, it cannot be undone.

The ownership has been renounced on this transaction:

https://bscscan.com/tx/0x91d0ea00e82fc22313b5abaa94e704e8f56d8b6713ff62caec411ac935abd611

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