Cyberscope

Audit Report

creationnetwork.ai

December 2024

Files CRNT.sol, Factory.sol, ICO.sol, MultiSigWallet.sol, Pair.sol, Router.sol, Staking.sol, USDTs.sol, USDCs.sol, Vesting.sol

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

Audit Updates

Initial Audit	12 Dec 2023 https://github.com/cyberscope-io/audits/blob/main/crtn/v1/audit.pdf
Corrected Phase 2	21 Nov 2024 https://github.com/cyberscope-io/audits/blob/main/crtn/v2/audit.pdf
Corrected Phase 3	26 Nov 2024 https://github.com/cyberscope-io/audits/blob/main/crtn/v3/audit.pdf
Corrected Phase 4	06 Dec 2024

Source Files

Filename	SHA256
Vesting.sol	d0813dff96ff309507d51b029c717a1c159cb790f204aae60cde740ef0aeafb9
USDTs.sol	f5114dce34e08d13168f56b645e6438ef7bdcc9c290c1216c8516dc078753c0a
USDCs.sol	9b5b781cca1519778cd9ffb2541f3445da9ef9a1aeb96b4061a6ad57759e7aa2
Staking.sol	dda5cba321f15067c2b46d00a475d2d20b52ba187ee891e211526733cec49889
Router.sol	b5ef548d4b144725c990d393116329de14a2b0dc55fb139ada3cdb5c0490e592

Pair.sol	03d4003a62e6e4d0f403ee4d678bfbe090a1c1fd2e3e237a7d6b5b974a715e1d
MultiSigWallet.sol	2d99fd549cfe7860471f9d0987cf908f6089f6e44f5b2b488c78fb42e25886af
ICO.sol	3bfaa1b3490bfb393005cdaf93c1eedc0a3a81a54b5e08ba49a75ece07ef107a
Factory.sol	620ebe5b194164da09dea9698b9b7e747cf1e1402e5f789ca6999f943faf7c5e
CRNT.sol	352139cb21a8aae31bfab8623a2552f5fc19f3cc5ac6c765417feebee1a7910ea

Overview

CRNT Contract Functionality

The CRNT contract is an ERC20 token implementation with additional functionality for taxes, as well as burning mechanisms. It defines several pre-allocated addresses for different purposes such as team, marketing, reserve, creator, liquidity, seed sale, and airdrop. Each of these addresses receives a specific allocation of tokens during contract deployment. The contract also introduces configurable taxes that apply to transfers, with exemptions based on specific conditions, such as interactions with a Dex.

ICO Contract Functionality

The ICO contract facilitates the initial token distribution across predefined stages. Each stage is associated with a token allocation, price, and duration, allowing participants to purchase tokens during the respective stages. The contract manages token purchases and tracks allocations for each participant. It also supports a claim mechanism where users can periodically release a portion of their purchased tokens. Additionally, the owner can withdraw stablecoins used for token purchases and update configurations such as the beneficiary and claimable token.

MultiSigWallet Contract Functionality

The `MultiSigWallet` contract is designed to manage and secure authorized approvals by the project owners. The contract accepts new proposals, which, once signed by a minimum number of owners, are broadcasted to other contracts within the project. Such actions include setting the DEX information for the token and distributing revenues to stakers.

Staking Contract Functionality

The Staking contract allows users to stake the project's token and earn rewards over time. Stakers receive rewards based on the amount staked and the duration of their staking. The contract tracks balances, staking timestamps, and stakers' activity, enabling the distribution of rewards through a reward rate defined in the contract. It also supports early withdrawal,

with penalties applied if tokens are withdrawn before a defined lock period. The owner can distribute additional revenue to stakers as part of the staking rewards system.

Factory Contract Functionality

The Factory contract is responsible for creating and managing pairs of tokens. It allows users with the appropriate PAIR_CREATOR_ROLE to create new token pairs, ensuring that each pair is unique and no duplicate pairs exist. The contract maintains a mapping of all created pairs, indexed by the token addresses, and stores an array of all pairs for easy tracking.

Router Contract Functionality

The Router contract's purpose is to implement the interactions with the Factory and Pair contracts to facilitate the addition and removal of liquidity for token pairs. Users can add liquidity by providing specified amounts of two tokens, which are transferred to the corresponding Pair contract. Similarly, users can remove liquidity by specifying the token pair and desired amounts. The Router ensures that only existing pairs created by the Factory are used.

Pair Contract Functionality

The Pair contract's purpose is to manage token reserves for a specific pair of tokens. It allows users to add liquidity by transferring the tokens to the contract and updating the reserve balances accordingly. Users can also remove liquidity, provided there are sufficient reserves, with the specified amounts transferred back to the user. The contract emits events for minting, burning, and swapping tokens to track liquidity-related actions.

Vesting Contract Functionality

The Vesting contract is designed to manage the scheduled release of tokens over a specified period. It enables the owner to allocate a total token supply for vesting, define a monthly release amount, and set a start time for the vesting schedule. Tokens can be claimed periodically by the beneficiary after the specified 30-day interval, ensuring that the total released tokens do not exceed the allocated amount. The contract also allows the owner to update the beneficiary address to redirect the vesting proceeds. Key

functionalities include token release, beneficiary management, and periodic release enforcement.

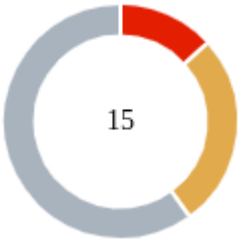
USDCs Contract Functionality

The USDCs contract implements an ERC20 token named "Custom USD Token" with the symbol "CUSD." The contract includes minting functionality, allowing the owner to mint tokens to specified addresses.

USDTs Contract Functionality

The USDTs contract is an ERC20 token named "Custom Tether Token" with the symbol "CTT." It enables the owner to mint tokens to specified addresses, similar to the USDC contract.

Findings Breakdown



Critical	2
Medium	4
Minor / Informative	9

Severity	Unresolved	Acknowledged	Resolved	Other
Critical	2	0	0	0
Medium	4	0	0	0
Minor / Informative	9	0	0	0

Diagnostics

Critical Medium Minor / Informative

Severity	Code	Description	Status
Critical	ISA	Incorrect Swap Amount	Unresolved
Critical	FLV	Flash Loan Vulnerability	Unresolved
Medium	IBAC	Incorrect Buy Amount Calculation	Unresolved
Medium	MPF	Missing Payable Functionality	Unresolved
Medium	PTAI	Potential Transfer Amount Inconsistency	Unresolved
Medium	URM	Uninitialized Reward Mechanism	Unresolved
Minor / Informative	CCR	Contract Centralization Risk	Unresolved
Minor / Informative	IDI	Immutable Declaration Improvement	Unresolved
Minor / Informative	INPC	Incorrect Nested Percentage Calculations	Unresolved
Minor / Informative	PSU	Potential Subtraction Underflow	Unresolved
Minor / Informative	RCS	Redundant Conditional Statements	Unresolved
Minor / Informative	L02	State Variables could be Declared Constant	Unresolved
Minor / Informative	L04	Conformance to Solidity Naming Conventions	Unresolved
Minor / Informative	L06	Missing Events Access Control	Unresolved

●	L13	Divide before Multiply Operation	Unresolved

ISA - Incorrect Swap Amount

Criticality	Critical
Location	Pair.sol#L157,209
Status	Unresolved

Description

The `Pair` contract implements the `swap` function to facilitate token exchanges using the contract product formula. The calculation for the output amount relies on normalized reserves instead of actual reserves. Additionally, the fee is not properly incorporated into the formula, as the `FEE_DENOMINATOR` is introduced to the numerator. These deviations from the standard formula result in significant discrepancies from the expected output amount.

```
uint256 amountInWithFee = (actualAmountIn * FEE_NUMERATOR) /
FEE_DENOMINATOR;
        amountOut = (amountInWithFee * normalizedReserveOut) /
(normalizedReserveIn + amountInWithFee);
```

```
function getOutputAmount(address tokenIn, uint256 amountIn)
external view returns (uint256 outputAmount) {
        bool isToken0In = tokenIn == token0;

        uint256 reserveIn = isToken0In ? reserve0 : reserve1;
        uint256 reserveOut = isToken0In ? reserve1 : reserve0;

        uint256 amountInWithFee = (amountIn * FEE_NUMERATOR) /
FEE_DENOMINATOR;
        outputAmount = (amountInWithFee * reserveOut) / (reserveIn
+ amountInWithFee);
}
```

Recommendation

Proper implementation of the contract product equation is crucial for the secure and consistent operation of the swap function. Inconsistencies could lead to a loss of assets in the pool. The team is advised to thoroughly test their implementation across different test cases to ensure the output amount is as expected, including the correct decimal precision for the output token.

FLV - Flash Loan Vulnerability

Criticality	Critical
Location	Staking.sol#L98
Status	Unresolved

Description

The `claimRewards` function is susceptible to flash-loan attacks. Specifically, the function mints reward tokens proportionally to the provided liquidity. A flash loan allows a user to borrow a large amount of tokens from a liquidity pool, perform operations, and return them within the same transaction. In this scenario, a user could borrow a large amount of tokens, add them to the liquidity pool, claim a disproportionately large amount of rewards, and then return the borrowed tokens. This operation could result in significant inflation of the reward token due to excessive minting.

```
function claimRewards() public nonReentrant {
    require(balances[msg.sender] > 0, "Insufficient balance");
    uint256 secondsStaked = block.timestamp -
stakedFromTS[msg.sender];
    uint256 reward = (balances[msg.sender] * REWARD_RATE *
secondsStaked) /
            (100 * SECONDS_IN_YEAR);
    require(reward > 0, "No rewards to claim");

    _mint(msg.sender, reward);
    stakedFromTS[msg.sender] = block.timestamp;

    emit RewardsClaimed(msg.sender, reward);
}
```

```
function claimReward() external nonReentrant {
    updateReward(msg.sender); // Update rewards before claiming

    uint256 owed = (balanceOf(msg.sender) * accRewardPerLiquidity)
/ 1e18 - rewardDebt[msg.sender];
    require(owed > 0, "Pair: NO_REWARD_AVAILABLE");

    rewardDebt[msg.sender] += owed;

    // Send rewards to the user
    IERC20(token0).safeTransfer(msg.sender, owed); // Assuming
rewards are in token0
    emit RewardClaimed(msg.sender, owed);
}
```

Recommendation

The team is advised to revise the implementation of the reward distribution mechanism to ensure secure operations. Implementing standard liquidity pool token mechanisms and ensuring proper tracking of deposit duration will help mitigate this issue.

IBAC - Incorrect Buy Amount Calculation

Criticality	Medium
Location	ICO.sol#L67
Status	Unresolved

Description

The ICO contract implements the `buyTokens` function to facilitate the sale of tokens in exchange for whitelisted assets. The amount to be bought is calculated by dividing the incoming assets by a fixed price. The contract then requires that this amount is less than the current allocation.

```
uint256 crntAmount = (amount * 10 ** 18) /
stages[currentStage].price;
```

In this calculation, both the numerator and denominator include the same decimal precision of 1e18. Therefore, the amount of bought tokens shares the same decimal precision as the incoming assets. However, tokens, particularly stablecoins, often have a different decimal precision than 1e18. If this is the case, the bought tokens will share the same decimal precision as the stablecoin, even though the allocation is defined with a precision of 1e18. This discrepancy could prevent the sale from being completed successfully.

Recommendation

To ensure successful token sales, the team is advised to adjust the buyTokens function to account for varying decimal precisions of incoming assets. Implementing this adjustment will prevent discrepancies and ensure that token sales can be completed as intended.

MPF - Missing Payable Functionality

Criticality	Medium
Location	MultiSigWallet.sol#L54,69
Status	Unresolved

Description

The `MultiSigWallet` contract implements the `approveTransaction` and `executeTransaction` functions, which trigger a low-level external call based on the authorized parameters. During this call, a value in native currency equal to `transaction.value` is also transferred. However, the calling function is not payable, and the contract includes no mechanism to receive native tokens. Therefore, transactions with non-zero value fields will fail.

```
function approveTransaction(uint256 txId) external onlyOwner {
require(txId < transactions.length, "Invalid transaction ID");
require(!approvals[txId][msg.sender], "Already approved");
require(!transactions[txId].executed, "Already executed");

approvals[txId][msg.sender] = true;
transactions[txId].approvals++;
emit TransactionApproved(txId, msg.sender);

if (transactions[txId].approvals >= required) {
executeTransaction(txId);
}
}
```

```
function executeTransaction(uint256 txId) internal {
Transaction storage transaction = transactions[txId];
require(transaction.approvals >= required, "Not enough approvals");
require(!transaction.executed, "Already executed");

transaction.executed = true;
(bool success, ) = transaction.to.call{value:
transaction.value}(transaction.data);
require(success, "Transaction execution failed");

emit TransactionExecuted(txId);
}
```

Recommendation

Ensuring the respective methods are payable or that the contract can receive payments in the native currency will improve code consistency and enhance flexibility and functionality.

PTAI - Potential Transfer Amount Inconsistency

Criticality	Medium
Location	Pair.sol#L53
Status	Unresolved

Description

The `addLiquidity()` function is used to transfer a specified amount of tokens to the contract. The fee or tax is an amount that is charged to the sender of an ERC20 token when tokens are transferred to another address. According to the specification, the transferred amount could potentially be less than the expected amount. This may produce inconsistency between the expected and the actual behavior.

The following example depicts the diversion between the expected and actual amount.

Tax	Amount	Expected	Actual
No Tax	100	100	100
10% Tax	100	100	90

In this case, the `addLiquidity()` function expects the received amounts to be equal to `amount0` and `amount1` and adds them to the reserves. However, this assumption is incorrect if the token implements transfer fees. As a result, a significant inconsistency arises during the provision of liquidity.

```
function addLiquidity(
address to,
uint256 amount0,
uint256 amount1
) external nonReentrant returns (uint256 liquidity) {
...
IERC20(token0).safeTransferFrom(to, address(this), amount0);
IERC20(token1).safeTransferFrom(to, address(this), amount1);
...
reserve0 += amount0;
reserve1 += amount1;
...
}
```

Recommendation

The team is advised to take into consideration the actual amount that has been transferred instead of the expected.

It is important to note that an ERC20 transfer tax is not a standard feature of the ERC20 specification, and it is not universally implemented by all ERC20 contracts. Therefore, the contract could produce the actual amount by calculating the difference between the transfer call.

`Actual Transferred Amount = Balance After Transfer - Balance Before Transfer`

URM - Uninitialized Reward Mechanism

Criticality	Medium
Location	Pair.sol#L28,121
Status	Unresolved

Description

The `Pair` contract implements functionalities to support a reward distribution mechanism. In these functionalities, rewards are calculated based on a fixed rate per unit of liquidity provided and the provision period. However, the `rewardRate` is initialized with a zero value and is not set at any point in the contract. As a result, functionalities related to reward distribution are ineffective. Specifically, the `updateReward` function always returns a zero value for `accRewardPerLiquidity`, rendering the `claimReward` function ineffective. Similarly, the `rewardDebt` mapping always stores a zero value.

```
uint256 public rewardRate; // Rate of reward per liquidity unit
```

```
function updateReward(address user) internal {
        uint256 totalLiquidity = totalSupply();
        if (totalLiquidity > 0) {
            uint256 duration = block.timestamp -
liquidityAddedTime[user];
            accRewardPerLiquidity += (rewardRate * duration) /
totalLiquidity;
        }
    }
```

Recommendation

Ensuring that the rewardRate is properly initialized will activate all reward distribution mechanisms.

CCR - Contract Centralization Risk

Criticality	Minor / Informative
Location	CRNT.sol#L81,93,103 Vesting.sol#L47 Staking.sol#L111 ICO.sol#L99,108,112 MultiSigWallet.sol#L23,43,54
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 setTax(uint256 _buyTax, uint256 _sellTax) external {
    require(msg.sender == multiSigWallet, "CRNT: Only
MultiSigWallet can set taxes");
    ...
}

function isOwner(address account) public view returns (bool) {
    for (uint256 i = 0; i < owners.length; i++) {
    if (owners[i] == account) return true;
    }
    return false;
}


function distributeRevenue(uint256 revenueAmount) external {
    require(msg.sender == multiSigWallet, "Only MultiSigWallet can
distribute revenue");
    ...
}
...
```

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.

IDI - Immutable Declaration Improvement

Criticality	Minor / Informative
Location	Vesting.sol#L42 Staking.sol#L46 MultiSigWallet.sol#L33 CRNT.sol#L50
Status	Unresolved

Description

The contract declares state variables that their value is initialized once in the constructor and are not modified afterwards. The `immutable` is a special declaration for this kind of state variables that saves gas when it is defined.

```
multiSigWallet
required
```

Recommendation

By declaring a variable as immutable, the Solidity compiler is able to make certain optimizations. This can reduce the amount of storage and computation required by the contract, and make it more gas-efficient.

INPC - Incorrect Nested Percentage Calculations

Criticality	Minor / Informative
Location	ICO.sol#L79
Status	Unresolved

Description

The contract is currently designed to perform a series of percentage-based calculations wherein it initially calculates a certain amount by applying a percentage to the original value. However, subsequent calculations erroneously use this newly derived amount as their base rather than the original value. This recursive calculation approach leads to each output amount being a percentage of the previous amount rather than a consistent percentage of the original amount. This results in significantly smaller outputs than expected and may not align with the intended financial logic of the contract.

```
function claimTokens(uint8 claimedStage ) external nonReentrant
crntAddressSet{
uint256 releaseAmount = ( stagePurchases[msg.sender][claimedStage]
* 25) / 100;
...
stagePurchases[msg.sender][claimedStage] -= releaseAmount;
...
}
```

Recommendation

Refactoring the contract code to ensure that all percentage calculations consistently use the original amount as their base is recommended. This can be achieved by storing the original amount in a separate variable and referencing this variable for each percentage calculation throughout the contract. This approach will ensure that all percentages are correctly calculated from the same initial value, preserving the integrity and expected functionality of the financial calculations.

PSU - Potential Subtraction Underflow

Criticality	Minor / Informative
Location	Pair.sol#L225,229
Status	Unresolved

Description

The contract subtracts two values, the second value may be greater than the first value if the contract owner misuses the configuration. As a result, the subtraction may underflow and cause the execution to revert.

```
function normalize(uint256 value, uint8 decimals) internal pure
returns (uint256) {
return value * 10**(18 - decimals);
}

function denormalize(uint256 value, uint8 decimals) internal pure
returns (uint256) {
return value / 10**(18 - decimals);
}
```

Recommendation

The team is advised to properly handle the code to avoid underflow subtractions and ensure the reliability and safety of the contract. The contract should ensure that the first value is always greater than the second value. It should add a sanity check in the setters of the variable or not allow executing the corresponding section if the condition is violated.

RCS - Redundant Conditional Statements

Criticality	Minor / Informative
Location	MultiSigWallet.sol#L69
Status	Unresolved

Description

The contract contains redundant conditional statements that can be simplified to improve code efficiency and performance. Conditional statements that are always satisfied are unnecessary and lead to larger code size, increased memory usage, and slower execution times.

The contract implements the internal `executeTransaction` function, which is only called by the `approveTransaction` function if `transactions[txId].approvals >= required`. Therefore, the former redundantly checks that the necessary number of approvals is provided and that the transaction is not executed.

```
function executeTransaction(uint256 txId) internal {
    Transaction storage transaction = transactions[txId];
    require(transaction.approvals >= required, "Not enough
approvals");
    require(!transaction.executed, "Already executed");

    transaction.executed = true;
    (bool success, ) = transaction.to.call{value:
transaction.value}(transaction.data);
    require(success, "Transaction execution failed");

    emit TransactionExecuted(txId);
}
```

Recommendation

It is recommended to refactor redundant conditional statements by eliminating unnecessary code structures and directly returning the outcome of the expression. This practice minimizes the number of operations required, reduces the code footprint, and optimizes memory and gas usage.

L02 - State Variables could be Declared Constant

Criticality	Minor / Informative
Location	Pair.sol#L28
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.

```
uint256 public rewardRate
```

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	Staking.sol#L146 ICO.sol#L108 CRNT.sol#L21,22,27,28,81,144,150
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 _crnt
address public immutable STAKING_CONTRACT
address public immutable ICO_CONTRACT
address public DEX_CONTRACT
address public LIQUIDITY_POOL
uint256 _sellTax
uint256 _buyTax
address _dexContract
address _liquidityPool
```

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.

L06 - Missing Events Access Control

Criticality	Minor / Informative
Location	Staking.sol#L148 ICO.sol#L110
Status	Unresolved

Description

Events are a way to record and log information about changes or actions that occur within a contract. They are often used to notify external parties or clients about events that have occurred within the contract, such as the transfer of tokens or the completion of a task. There are functions that have no event emitted, so it is difficult to track off-chain changes.

```
crntAddress = _crnt
```

Recommendation

To avoid this issue, 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.

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.

L13 - Divide before Multiply Operation

Criticality	Minor / Informative
Location	Staking.sol#L116,123 Pair.sol#L69,70,184,185,215,216
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 revenuePerToken = revenueAmount / totalStaked
uint256 stakerRevenue = balances[staker] * revenuePerToken

uint256 amountInWithFee = (actualAmountIn * FEE_NUMERATOR) /
FEE_DENOMINATOR
amountOut = (amountInWithFee * normalizedReserveOut) /
(normalizedReserveIn + amountInWithFee)
```

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.

Functions Analysis

Contract	Type	Bases		
	Function Name	**Visibility**	**Mutability**	**Modifiers**
Vesting	Implementation	Ownable, ReentrancyGuard		
		Public	✓	-
	updateBeneficiary	External	✓	-
	releaseTokens	External	✓	nonReentrant
USDTs	Implementation	ERC20, Ownable, ReentrancyGuard		
		Public	✓	Ownable ERC20
USDCs	Implementation	ERC20, Ownable, ReentrancyGuard		
		Public	✓	Ownable ERC20
ICRNT	Interface			
	burnFrom	External	✓	-
Staking	Implementation	ERC20, Ownable,		

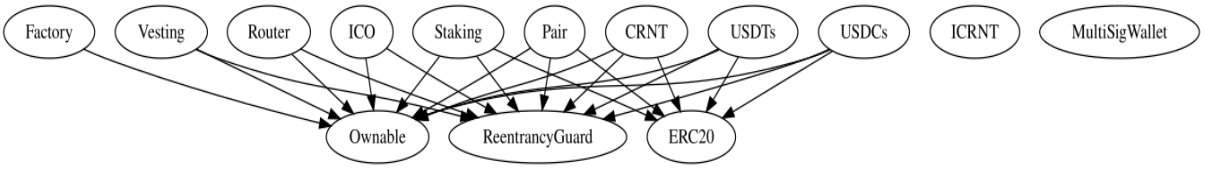
		ReentrancyGuard		
		Public	✓	Ownable ERC20
	stake	External	✓	nonReentrant crntAddressSet
	withdraw	External	✓	crntAddressSet
	claimRewards	Public	✓	nonReentrant
	distributeRevenue	External	✓	-
	_removeStaker	Internal	✓	
	setCrntAddress	External	✓	onlyOwner
Router	Implementation	ReentrancyGuard, Ownable		
		Public	✓	Ownable
	addLiquidity	External	✓	nonReentrant
	removeLiquidity	External	✓	nonReentrant
	swap	External	✓	nonReentrant
	getOutputAmount	External		-
	withdrawEther	External	✓	onlyOwner
		External	Payable	-
		External	Payable	-
Pair	Implementation	ERC20, ReentrancyGuard, Ownable		

		Public	✓	ERC20
	addLiquidity	External	✓	nonReentrant
	claimReward	External	✓	nonReentrant
	updateReward	Internal	✓	
	removeLiquidity	External	✓	nonReentrant
	swap	External	✓	nonReentrant
	getOutputAmount	External		-
	_update	Private	✓	
	normalize	Internal		
	denormalize	Internal		
	sqrt	Internal		
	min	Internal		
MultiSigWallet	Implementation			
		Public	✓	-
	isOwner	Public		-
	submitTransaction	External	✓	onlyOwner
	approveTransaction	External	✓	onlyOwner
	executeTransaction	Internal	✓	
ICO	Implementation	Ownable, ReentrancyGuard		
		Public	✓	Ownable

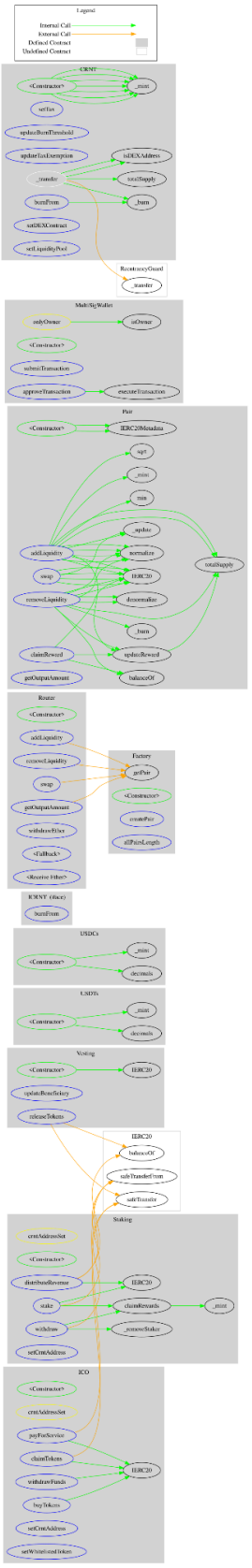
	buyTokens	External	✓	nonReentrant
	claimTokens	External	✓	nonReentrant crntAddressSet
	payForService	External	✓	crntAddressSet
	withdrawFunds	External	✓	onlyOwner
	setCrntAddress	External	✓	onlyOwner
	setWhitelistedToken	External	✓	onlyOwner
Factory	Implementation	Ownable		
		Public	✓	Ownable
	createPair	External	✓	onlyOwner
	allPairsLength	External		-
CRNT	Implementation	ERC20, Ownable, ReentrancyGuard		
		Public	✓	Ownable ERC20
	setTax	External	✓	-
	updateBurnThreshold	External	✓	-
	updateTaxExemption	External	✓	-
	_transfer	Internal	✓	
	burnFrom	External	✓	-
	setDEXContract	External	✓	-
	setLiquidityPool	External	✓	-

	isDEXAddress	Internal		

Inheritance Graph



Flow Graph


Legend
Internal Call
External Call
Defined Contract
Undefined Contract
CRNT
<Constructor>
_mint
setTax
updateBurnThreshold
updateTaxExemption
isDEXAddress
_transfer
totalSupply
burnFrom
_burn
setDEXContract
setLiquidityPool
ReentrancyGuard
_transfer
MultiSigWallet
onlyOwner
isOwner
<Constructor>
submitTransaction
approveTransaction
executeTransaction
Pair
<Constructor>
IERC20Metadata
sqrt
_min
min
update
addLiquidity
normalize
totalSupply
swap
IERC20
removeLiquidity
denormalize
_burn
claimReward
updateReward
getOutputAmount
balanceOf
Router
<Constructor>
addLiquidity
removeLiquidity
swap
getOutputAmount
withdrawFloor
<fallback>
<Receive Ether>
Factory
getPair
<Constructor>
createPair
allPairsLength
ICRNT (Iface)
burnFrom
USDCs
<Constructor>
_mint
decimals
USDTs
<Constructor>
_mint
decimals
Vesting
<Constructor>
IERC20
updateBeneficiary
releaseTokens
IERC20
balanceOf
safeTransferFrom
safeTransfer
Staking
crntAddressSet
<Constructor>
distributeReverse
IERC20
stake
claimRewards
_mint
withdraw
_removeStaker
setCrntAddress
ICO
<Constructor>
crntAddressSet
payForService
claimTokens
IERC20
withdrawFunds
buyTokens
setCrntAddress
setWhitelistedToken

Summary

creationnetwork.ai contract implements a token, ICO, staking, vesting and utility 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

cyberscope.io