Cyberscope

Audit Report
creationnetwork.ai

November 2024

Files CRNT.sol, Factory.sol, ICO.sol, LiquidityLock.sol, Pair.sol, Router.sol, Staking.sol, USDT.sol, USDC.sol, Vesting.sol, ZapV2.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
Corrected Phase 2	21 Nov 2024

Source Files

Filename	SHA256
ZapV2.sol	96aaf11ef4f6f37ad65130a222e439cad9e549df2c022a62f5013106287c92c7
Vesting.sol	fab586580b1a962ee6c1cc8ae5fc99837a0e55ec7cded3ebf5f9b507adbdadf2
USDT.sol	08ab441467f7cd638041b47d3cc860d490ce1bf13dfd15672d8d72222ac242a2
USDC.sol	f954a12cd353ca9740f5e73da80cd7c15f28746539f4355fdb4c654adfbb822b
Staking.sol	16563764af98a013d3b14ac07636badc506498c27eef280807005fe166518370
Router.sol	d358ff0e1ef118ff2204e8bc85fd5a07ad3fec88bde2a9e0c2acaf10d939ddfc
Pair.sol	d57219510279663a244c21bdbaabb521773114031223ca3286fb1e196330d146
LiquidityLock.sol	56f382e79164fde50a1856796c67bae894d0dde686efa92bd4d4fd88ec0e4e63
ICO.sol	a182a06a72f9ac8256d2af6163c1d1957a0fce95e191cb57885054a33c75bde0

Factory.sol	47967fd819ccc6f5c002df0688e4ada1d2cf6ee67b145781f2ab4fe3f255e286
CRNT.sol	9435ca3e7e629893c0b08329b7afef3b214db6ba293cd0a2a6e72a2a53b864b1

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 the staking or ICO contracts.

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.

LiquidityLock Contract Functionality

The LiquidityLock contract is designed to manage and secure a specified amount of ERC20 tokens by implementing a locking mechanism. It restricts the owner’s ability to withdraw liquidity until a predefined lock period has passed. The contract enforces a penalty for early withdrawal, deducting a portion of the token balance if the withdrawal occurs before the lock period ends.

Staking Contract Functionality

The Staking contract allows users to stake ERC20 tokens 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 it tomanage 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. It does not implement mechanisms for minting or managing LP tokens, focusing solely on maintaining token reserves.

ZapV2 Contract Functionality

The ZapV2 contract serves as an interface to simplify liquidity operations with the Router contract. It enables users to add or remove liquidity for token pairs in a single transaction by interacting with the Router. The constructor accepts a Router address to initialize the contract, enabling interaction with the specified Router instance. The contract aims to streamline liquidity operations for end users while relying on the Router contract's functionality for managing token pairs.

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.

USDC Contract Functionality

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

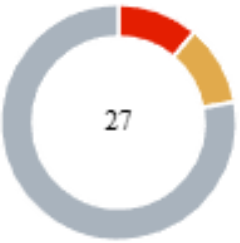
USDT Contract Functionality

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

Contract Readability Comment

The audit scope is to check for security vulnerabilities, validate the business logic and propose potential optimizations. The contract is missing the fundamental principles of a Solidity smart contract regarding gas consumption, code readability, and data structures. According to the previously mentioned issues, the contract cannot be assumed that it is in a production-ready state. Given these issues, it is not advisable to assume that the contract is in a production-ready state. The development team is strongly encouraged to re-evaluate the business logic and Solidity guidelines to ensure that the contract adheres to established best practices and security measures. It is recommended that the team review the contract's gas consumption and optimize it accordingly to minimize costs and improve the contract's efficiency. The code's readability should also be improved by simplifying function definitions and using descriptive variable names, as this will enhance the contract's auditability and maintenance.

Findings Breakdown



Critical	3
Medium	3
Minor / Informative	21

Severity	Unresolved	Acknowledged	Resolved	Other
Critical	3	0	0	0
Medium	3	0	0	0
Minor / Informative	21	0	0	0

Diagnostics

Critical Medium Minor / Informative

Severity	Code	Description	Status
●	IPI	Incorrect Pair Implementation	Unresolved
●	UTB	Unrestricted Token Burning	Unresolved
●	UTS	Unrestricted Token Selection	Unresolved
●	DVI	Dependencies Version Inconsistency	Unresolved
●	SMP	Staking Mechanism Penalty	Unresolved
●	ULT	Unused Lock Timestamp	Unresolved
●	RFO	Redundant Functions Overrides	Unresolved
●	AM	Allocation Mismatch	Unresolved
●	CCR	Contract Centralization Risk	Unresolved
●	IDI	Immutable Declaration Improvement	Unresolved
●	IZF	Inaccurate Zap Functionality	Unresolved
●	ISI	Incomplete Stages Initialization	Unresolved
●	IMSCA	Initial Minting Supply Check Absence	Unresolved
●	MT	Mints Tokens	Unresolved

●	MLLI	Misleading Liquidity Lock Implementation	Unresolved
●	MTD	Misleading Tax Declaration	Unresolved
●	MTN	Misleading Token Naming	Unresolved
●	RBM	Redundant Beneficiary Mechanism	Unresolved
●	RPU	Redundant Payable Usage	Unresolved
●	L02	State Variables could be Declared Constant	Unresolved
●	L04	Conformance to Solidity Naming Conventions	Unresolved
●	L08	Tautology or Contradiction	Unresolved
●	L13	Divide before Multiply Operation	Unresolved
●	L16	Validate Variable Setters	Unresolved
●	L19	Stable Compiler Version	Unresolved
●	L20	Succeeded Transfer Check	Unresolved
●	L22	Potential Locked Ether	Unresolved

IPI - Incorrect Pair Implementation

Criticality	Critical
Location	Pair.sol#L8
Status	Unresolved

Description

The `Pair` contract does not accurately implement standard liquidity management mechanisms commonly found in decentralized exchanges, such as those in Uniswap. While the contract provides functions for adding and removing liquidity, it does not mint liquidity provider (LP) tokens to represent ownership of the liquidity pool. This omission makes it impossible to track or verify an individual's contribution to the pool. As a result, the removal of liquidity does not require any proof of prior contribution, allowing any user to call the `removeLiquidity` function and withdraw arbitrary amounts of the pool's reserves.

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

    function removeLiquidity(address to, uint256 amount0,
uint256 amount1) external nonReentrant {
        require(reserve0 >= amount0 && reserve1 >= amount1,
"Pair: INSUFFICIENT_LIQUIDITY");
        reserve0 -= amount0;
        reserve1 -= amount1;
        IERC20(token0).safeTransfer(to, amount0);
        IERC20(token1).safeTransfer(to, amount1);
        emit Burn(to, amount0, amount1);
    }
```

Recommendation

It is recommended to adopt Uniswap’s approach for liquidity management, including the minting and burning of LP tokens to track and enforce ownership of liquidity contributions. The functions for adding and removing liquidity should integrate LP token mechanics to ensure that only users who have added liquidity can remove it and in proportions that reflect their contributions. Aligning the `Pair`, `Router`, and `Factory` contracts with Uniswap's implementation or similar robust designs ensures the functionality is secure, scalable, and adheres to widely accepted standards for decentralized exchange protocols.

UTB - Unrestricted Token Burning

Criticality	Critical
Location	CRNT.sol#L57
Status	Unresolved

Description

The `burnFrom` function allows any external account to burn tokens from any user's balance without any restrictions or authorization checks. This design introduces a critical vulnerability, as malicious actors can arbitrarily reduce another user's token balance, leading to potential loss of funds and disruption of the token's intended functionality. The absence of an allowance mechanism or ownership verification makes this issue particularly severe.

```
function burnFrom(address account, uint256 amount) external {
    _burn(account, amount);  // Burn tokens from the account's
balance
    }
```

Recommendation

It is recommended to restrict the `burnFrom` function by implementing an allowance-based mechanism that requires the caller to be explicitly authorized to burn tokens on behalf of the account. This ensures that token burning aligns with the token holder's intent and prevents unauthorized burning. Adopting a well-established pattern, such as the one used in OpenZeppelin's ERC20Burnable, can enforce proper checks and maintain the integrity of token balances.

UTS - Unrestricted Token Selection

Criticality	Critical
Location	ICO.sol#L40,64,77 Staking.sol#L46,64,85,96
Status	Unresolved

Description

In the `ICO` contract, the `buyTokens`, `payForService`, and `claimTokens` functions allow users to specify any token as an argument, enabling malicious actors to exploit the protocol. In `buyTokens`, users can pass unauthorized tokens, such as those they control with arbitrary supply or decimals, to acquire the protocol's tokens without legitimate payment. In `payForService`, users can bypass payment requirements by specifying tokens they own. Similarly, in `claimTokens`, users can manipulate the process to claim tokens other than the intended protocol tokens, undermining the token distribution system.

```
function buyTokens(uint256 amount, address stablecoin) external
nonReentrant {
        require(currentStage < stages.length, "ICO stages
completed");
        require(block.timestamp >=
stages[currentStage].startTime, "Stage not started");
        require(amount > 0, "Invalid purchase amount");

          if (stages[currentStage].tokensSold >=
stages[currentStage].allocation ||
            block.timestamp >= stages[currentStage].startTime +
stages[currentStage].duration) {
            currentStage++;
            require(currentStage < stages.length, "ICO stages
completed");
            stages[currentStage].startTime = block.timestamp;
            emit StageAdvanced(currentStage);
        }

        IERC20(stablecoin).safeTransferFrom(msg.sender,
address(this), amount);
        uint256 crntAmount = (amount * 10 ** 18) /
stages[currentStage].price;
        require(crntAmount <= stages[currentStage].allocation -
stages[currentStage].tokensSold, "Stage allocation exceeded");

        purchases[msg.sender] += crntAmount;
        stages[currentStage].tokensSold += crntAmount;
        emit TokensPurchased(msg.sender, crntAmount);


    }

    function claimTokens(address _crntToken) external
nonReentrant {
        uint256 releaseAmount = (purchases[msg.sender] * 25) /
100;
        require(releaseAmount > 0, "No tokens available for
release");

        require(block.timestamp >=
lastClaimTimestamp[msg.sender] + 30 days, "Claim period not
reached");
         crntToken = IERC20(_crntToken);
        purchases[msg.sender] -= releaseAmount;
        lastClaimTimestamp[msg.sender] = block.timestamp;
        crntToken.safeTransfer(msg.sender, releaseAmount);
        // withdrawFunds(stableCoin,)
        emit TokensClaimed(msg.sender, releaseAmount);
    }
```

```
    function payForService(uint256 amount,address _crntToken)
external {
        require(amount > 0, "Invalid service amount");
        crntToken = IERC20(_crntToken);
        crntToken.safeTransferFrom(msg.sender, address(this),
amount);
    }
```

This is issue is present in the `Staking` contract as well. In addition to the previously identified risks, the `distributeRevenue` function allows the owner to distribute rewards to stakers without verifying the token being used. This creates a significant vulnerability, as a user can manipulate the token state by calling other functions that allow arbitrary token selection and then frontrun the owner's call to `distributeRevenue`. This can lead to a denial-of-service (DoS) condition or unintended behavior during revenue distribution.

Recommendation

To address this issue, the contracts should enforce strict validation of token arguments in these functions. Only predefined, authorized tokens should be accepted, and this validation should be hardcoded or governed by a secure mechanism. By restricting token selection, the protocol ensures that only legitimate tokens are used, preserving the integrity of the ICO, payment system, and token claims.

DVI - Dependencies Version Inconsistency

Criticality	Medium
Location	CRNT.sol#L26 Vesting.sol#L23 USDT.sol#L11 Staking.sol#L30 LiquidityLock.sol#L13 ICO.sol#L33
Status	Unresolved

Description

The project contains inconsistencies in the versions of OpenZeppelin libraries imported across various files. For example, the `Ownable` contract uses a version that expects a constructor argument, which aligns with OpenZeppelin v5.0.0, while the `ReentrancyGuard` contract is imported from a directory structure consistent with v4.9.0. This creates a mismatch in the versions of dependencies used in the project. Despite the contracts being deployable, this inconsistency requires remappings or manual adjustments to the imports, which increases complexity and the likelihood of errors during development and deployment.

```
constructor(address _icoContract, address _stakingContract)
Ownable(msg.sender) ERC20("CRNT Token", "CRNT") {
        _mint(TEAM, 44_280_000 * (10 ** 18));
        _mint(MARKETING, 73_800_000 * (10 ** 18));
        _mint(RESERVE, 36_900_000 * (10 ** 18));
        _mint(CREATOR, 11_070_000 * (10 ** 18));
        _mint(LIQUIDITY, 73_800_000 * (10 ** 18));
        _mint(SEED_PREPUBLIC_SALE, 121_770_000 * (10 ** 18));
        _mint(AIRDROP, 7_380_000 * (10 ** 18));
        _mint(_icoContract, 121_770_000 * (10 ** 18));
        STAKING_CONTRACT = _stakingContract;
        ICO_CONTRACT = _icoContract;
    }

constructor(address _crntToken, uint256 _totalAllocation,
uint256 _monthlyRelease, uint256 _startTimestamp)
    Ownable(msg.sender)
    {
        crntToken = IERC20(_crntToken);
        totalAllocation = _totalAllocation;
        monthlyRelease = _monthlyRelease;
        startTimestamp = _startTimestamp;
        lastReleaseTimestamp = _startTimestamp;
        beneficiary = owner(); // Initially, the owner is the
beneficiary
    }
```

Recommendation

It is recommended to standardize the OpenZeppelin dependency versions throughout the project. This ensures consistency, avoids compatibility issues, and simplifies maintenance. Using a single version across all imports eliminates the need for remappings and reduces potential confusion for developers working with the codebase. Additionally, reviewing and documenting the chosen OpenZeppelin version can help maintain clarity and alignment in the project.

SMP - Staking Mechanism Penalty

Criticality	Medium
Location	Staking.sol#L46,85
Status	Unresolved

Description

The staking mechanism updates the `stakedfromTS` timestamp to the current block timestamp every time a user stakes additional tokens. This timestamp is used in the `claimRewards` function to calculate the rewards based on the time elapsed since the last stake. As a result, each new staking action effectively resets the reward calculation, causing users to lose the rewards they would have earned for the duration already staked. This behavior unintentionally penalizes users for staking additional tokens, discouraging further participation.

```
function stake(uint256 amount, address _crntToken) external
nonReentrant {
        crntToken = IERC20(_crntToken);
        require(amount >= 1000, "amount Below minimun staking
treshold");
        crntToken.safeTransferFrom(msg.sender, address(this),
amount);
        // Add new staker to the array
        if (!isStaker[msg.sender]) {
            stakerIndex[msg.sender] = stakers.length;
            stakers.push(msg.sender);
            isStaker[msg.sender] = true;
        }

        balances[msg.sender] += amount;
        totalStaked += amount;
        stakedfromTS[msg.sender] = block.timestamp;
        lockTimestamp = block.timestamp;
        emit Staked(msg.sender, amount);
    }

function claimRewards() public nonReentrant {
        require(balances[msg.sender] >= 0, "you donot have
enouph 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);
    }
```

Recommendation

To address this issue, the staking mechanism should be adjusted to ensure that additional stakes do not reset the reward calculation for previously staked tokens. This can be achieved by preserving the accrued rewards or implementing a system that calculates rewards based on a weighted average or similar approach. Ensuring users are not penalized for increasing their stake aligns the staking mechanism with user expectations and encourages greater participation in the protocol.

ULT - Unused Lock Timestamp

Criticality	Medium
Location	Staking.sol#L46
Status	Unresolved

Description

The `lockTimestamp` variable in the `Staking` contract is set to the current block timestamp within the `stake` function, but it is not referenced or utilized anywhere else in the contract. In a staking context, a lock timestamp is typically used to enforce a minimum staking period, calculate rewards, or implement penalties for early withdrawals. The absence of any functionality tied to this variable suggests a lack of purpose or an incomplete implementation. As a result, its presence adds unnecessary complexity without providing any value to the staking process.

```
function stake(uint256 amount, address _crntToken) external
nonReentrant {
        crntToken = IERC20(_crntToken);
        require(amount >= 1000, "amount Below minimun staking
treshold");
        crntToken.safeTransferFrom(msg.sender, address(this),
amount);
        // Add new staker to the array
        if (!isStaker[msg.sender]) {
            stakerIndex[msg.sender] = stakers.length;
            stakers.push(msg.sender);
            isStaker[msg.sender] = true;
        }

        balances[msg.sender] += amount;
        totalStaked += amount;
        stakedfromTS[msg.sender] = block.timestamp;
        lockTimestamp = block.timestamp;
        emit Staked(msg.sender, amount);
    }
```

Recommendation

It is recommended to either remove the `lockTimestamp` variable if it is not required or integrate it into the staking logic with a clearly defined purpose. For example, it could be used to enforce locking periods for stakers, calculate reward eligibility, or manage penalties for premature actions. This ensures the contract is streamlined and all variables serve a meaningful purpose, enhancing maintainability and functionality.

RFO - Redundant Functions Overrides

Criticality	Minor / Informative
Location	USDT.sol#L20 USDC.sol#L20
Status	Unresolved

Description

The USDC and USDT token contracts override multiple standard ERC20 functions, such as `totalSupply`, `transfer`, `approve`, `transferFrom`, and `allowance`, without adding any custom logic or functionality. These overrides merely call the parent contract's implementations and do not serve any meaningful purpose. This redundancy increases the code size unnecessarily and may confuse developers by implying that the functions have been modified, even though their behavior remains unchanged.

```
function totalSupply() public view override returns (uint256) {
        return super.totalSupply();
    }
    function transfer(address recipient, uint256 amount) public
override returns (bool) {
        return super.transfer(recipient, amount);
    }
    function approve(address spender, uint256 amount) public
override returns (bool) {
        return super.approve(spender, amount);
    }
    function transferFrom(address sender, address recipient,
uint256 amount) public override returns (bool) {
        return super.transferFrom(sender, recipient, amount);
    }
    function allowance(address owner, address spender) public
view override returns (uint256) {
        return super.allowance(owner, spender);
    }
```

Recommendation

It is recommended to remove these redundant overrides to improve the contract's clarity, maintainability, and efficiency. Retaining only the inherited ERC20 functionality eliminates unnecessary code, reduces potential confusion, and ensures a cleaner and more concise implementation. This approach also aligns with best practices for extending well-audited standard contracts like ERC20.

AM - Allocation Mismatch

Criticality	Minor / Informative
Location	CRNT.sol#L34 ICO.sol#L33
Status	Unresolved

Description

The `ICO` contract defines three stages with specific token allocations, but the total of these allocations does not match the amount minted to the ICO contract in the `CRNT` contract. This inconsistency creates a discrepancy between the tokens intended to be distributed across the ICO stages and the actual tokens made available. This can lead to errors during the ICO process, as the token supply may be insufficient to fulfill the expected allocations, or it may result in excess tokens that remain unused or unaccounted for.

```
_mint(_icoContract, 121_770_000 * (10 ** 18));

constructor() Ownable(msg.sender) {
        stages[0] = Stage(StageName.Ignition, 33_369_000 * 10
** 18, 0.01 * 10 ** 18, 23 days, block.timestamp, 0);
        stages[1] = Stage(StageName.Acceleration, 44_200_500 *
10 ** 18, 0.0125 * 10 ** 18, 15 days, 0, 0);
        stages[2] = Stage(StageName.Momentum, 44_200_500 * 10
** 18, 0.015 * 10 ** 18, 15 days, 0, 0);

    }
```

Recommendation

It is recommended to ensure that the total token allocation across all ICO stages matches the amount minted to the ICO contract. The allocation logic in the `ICO` contract and the minting logic in the `CRNT` contract should be reviewed and updated for consistency. This alignment prevents discrepancies, ensures the ICO functions as intended, and maintains clarity about the token distribution process.

CCR - Contract Centralization Risk

Criticality	Minor / Informative
Location	CRNT.sol#L39 USDT.sol#L15 Vesting.sol#L35,42 Staking.sol#L96 LiquidityLock.sol#L21 ICO.sol#L83
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
onlyOwner {
        require(_buyTax <= 5 && _sellTax <= 5, "Tax rate too
high");
        buyTax = _buyTax;
        sellTax = _sellTax;
        emit TaxRateUpdated(buyTax, sellTax);
    }

function updateBeneficiary(address newBeneficiary) external
onlyOwner {
        require(newBeneficiary != address(0), "Vesting: New
beneficiary is the zero address");
        emit BeneficiaryUpdated(beneficiary, newBeneficiary);
        beneficiary = newBeneficiary;
    }


    function releaseTokens() external nonReentrant onlyOwner {
        require(msg.sender == beneficiary, "Vesting: Only the
beneficiary can release tokens");

        require(block.timestamp >= lastReleaseTimestamp + 30
days, "Vesting: 30-day interval not met");
        uint256 releaseAmount = monthlyRelease;
        require(releasedAmount + releaseAmount <=
totalAllocation, "Vesting: Allocation exceeded");

        lastReleaseTimestamp = block.timestamp;
        releasedAmount += releaseAmount;
        require(crntToken.balanceOf(address(this)) >=
releaseAmount, "Insufficient token balance in contract");
        crntToken.safeTransfer(owner(), releaseAmount);
        emit TokensReleased(owner(), releaseAmount);
    }

...
```

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#L27,28,29 Pair.sol#L21,22 LiquidityLock.sol#L15 CRNT.sol#L35,36
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.

```
totalAllocation
monthlyRelease
startTimestamp
token0
token1
lockTimestamp
STAKING_CONTRACT
ICO_CONTRACT
```

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.

IZF - Inaccurate Zap Functionality

Criticality	Minor / Informative
Location	ZapV2.sol#L6
Status	Unresolved

Description

The ZapV2 contract is named and positioned as a "zap" contract, but it does not implement the typical functionality associated with such contracts. A zap contract generally allows users to interact with liquidity pools or staking systems using a single asset, automatically converting or splitting it into the required assets for the target action, such as adding liquidity or staking. Instead, this contract relies on the Router contract to facilitate adding and removing liquidity, requiring users to provide multiple assets and amounts directly. This behavior deviates from the expected behavior of a zap contract.

```
contract ZapV2 {
    Router public router;

    constructor(address payable _router) {
        router = Router(_router);
    }

    function zapIn(address tokenA, address tokenB, uint256 amountA,
uint256 amountB) external {
        // Allows user to add liquidity in a single transaction
        router.addLiquidity(tokenA, tokenB, amountA, amountB);
    }

    function zapOut(address tokenA, address tokenB, uint256 amountA,
uint256 amountB) external {
        // Allows user to remove liquidity in a single transaction
        router.removeLiquidity(tokenA, tokenB, amountA, amountB);
    }
}
```

Recommendation

The current implementation may mislead users about the contract's intended functionality and could result in confusion or improper use. To align the contract's functionality with its name, additional logic should be implemented to enable the single-asset operations typically associated with zap contracts. Alternatively, the contract should be renamed to accurately reflect its current capabilities to avoid any misinterpretation.

ISI - Incomplete Stages Initialization

Criticality	Minor / Informative
Location	ICO.sol#L16,23,33
Status	Unresolved

Description

The `ICO` contract defines an array of five stages and an enumeration with five stage names, but only three stages are initialized in the constructor. There is no mechanism to create or update the remaining stages, leaving the contract with uninitialized entries in the `stages` array and two unused enum values. This inconsistency creates confusion, as the contract structure suggests the existence of five stages, but only three are functional. Additionally, without a way to add or modify stages, the flexibility of the contract to adapt to changes or complete its intended design is limited.

```
struct Stage {
        StageName name;
        uint256 allocation;
        uint256 price;
        uint256 duration;
        uint256 startTime;
        uint256 tokensSold;
    }

    Stage[5] public stages;

constructor() Ownable(msg.sender) {
        stages[0] = Stage(StageName.Ignition, 33_369_000 * 10
** 18, 0.01 * 10 ** 18, 23 days, block.timestamp, 0);
        stages[1] = Stage(StageName.Acceleration, 44_200_500 *
10 ** 18, 0.0125 * 10 ** 18, 15 days, 0, 0);
        stages[2] = Stage(StageName.Momentum, 44_200_500 * 10
** 18, 0.015 * 10 ** 18, 15 days, 0, 0);

    }
```

Recommendation

It is recommended to either remove the unused enum values and reduce the size of the `stages` array to match the initialized stages or implement functionality to allow an elevated role, such as the owner, to create or modify stages. This ensures that the contract reflects its intended design and avoids confusion about unused stages or uninitialized data. Proper documentation of the stage initialization process and expected usage of the enum values is also advisable for clarity.

IMSCA - Initial Minting Supply Check Absence

Criticality	Minor / Informative
Location	CRNT.sol#L9,26
Status	Unresolved

Description

The `CRNT` contract declares an `INITIAL_SUPPLY` constant to define the total token supply, which is intended to represent the sum of all minted amounts. However, while the constructor mints tokens to various predefined addresses, there is no validation to ensure that the total minted tokens align with the declared `INITIAL_SUPPLY`. Although the predefined amounts for all addresses, besides the ICO contract, sum correctly to the `INITIAL_SUPPLY`, this relationship is not explicitly enforced within the contract logic, leaving room for discrepancies during future modifications or deployments.

```
uint256 public constant INITIAL_SUPPLY = 369_000_000 * (10 **
18);

constructor(address _icoContract, address _stakingContract)
Ownable(msg.sender) ERC20("CRNT Token", "CRNT") {
        _mint(TEAM, 44_280_000 * (10 ** 18));
        _mint(MARKETING, 73_800_000 * (10 ** 18));
        _mint(RESERVE, 36_900_000 * (10 ** 18));
        _mint(CREATOR, 11_070_000 * (10 ** 18));
        _mint(LIQUIDITY, 73_800_000 * (10 ** 18));
        _mint(SEED_PREPUBLIC_SALE, 121_770_000 * (10 ** 18));
        _mint(AIRDROP, 7_380_000 * (10 ** 18));
        _mint(_icoContract, 121_770_000 * (10 ** 18));
        STAKING_CONTRACT = _stakingContract;
        ICO_CONTRACT = _icoContract;
    }
```

Recommendation

It is recommended to implement a validation mechanism within the constructor to verify that the total amount of tokens minted equals the declared `INITIAL_SUPPLY`. This ensures consistency and prevents potential issues where the actual token supply deviates from the intended value. Including this check improves the reliability of the contract and maintains alignment between the declared and actual token supply.

MT - Mints Tokens

Criticality	Minor / Informative
Location	USDC.sol#L15 USDT.sol#L15
Status	Unresolved

Description

The contract owner has the authority to mint tokens. The owner may take advantage of it by calling the `mint` function. As a result, the contract tokens will be highly inflated.

```
function mint(address to, uint256 amount) external onlyOwner
nonReentrant {
        _mint(to, amount);
        emit Minted(to, amount);
    }
```

Recommendation

The team should carefully manage the private keys of the owner's account. We strongly recommend a powerful security mechanism that will prevent a single user from accessing the contract admin functions.

Temporary Solutions:

These measurements do not decrease the severity of the finding

- Introduce a time-locker mechanism with a reasonable delay.
- Introduce a multi-signature wallet so that many addresses will confirm the action.
- Introduce a governance model where users will vote about the actions.

Permanent Solution:

- Renouncing the ownership, which will eliminate the threats but it is non-reversible.

MLLI - Misleading Liquidity Lock Implementation

Criticality	Minor / Informative
Location	LiquidityLock.sol#L8
Status	Unresolved

Description

The `LiquidityLock` contract claims to serve as a mechanism for locking liquidity, but its implementation does not align with the intended purpose of a liquidity lock in decentralized finance. Instead of securely locking liquidity in a pool for a defined period, the contract allows the owner to withdraw all the tokens held by the contract at any time. While a 10% penalty is applied for withdrawals before the lock period ends, this does not enforce any meaningful restriction or safeguard for liquidity. The contract does not interact with a liquidity pool or manage liquidity pair tokens, and its functionality contradicts the typical use case of a liquidity lock, which is to ensure liquidity remains inaccessible to any privileged entity during the lock period.

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.20;

import "@openzeppelin/contracts/token/ERC20/IERC20.sol";
import "@openzeppelin/contracts/access/Ownable.sol";
import "@openzeppelin/contracts/security/ReentrancyGuard.sol";

contract LiquidityLock is Ownable, ReentrancyGuard {
    IERC20 public liquidityToken;
    uint256 public lockTimestamp;
    uint256 public constant LOCK_PERIOD = 21 days;

    constructor(address _liquidityToken) Ownable(msg.sender) {
        liquidityToken = IERC20(_liquidityToken);
        lockTimestamp = block.timestamp;
    }
    receive() external payable {
        revert("This contract does not accept Ether.");
    }

    function withdrawLiquidity() external onlyOwner
nonReentrant {
        uint256 balance =
liquidityToken.balanceOf(address(this));
        if(block.timestamp < lockTimestamp + LOCK_PERIOD){
            uint256 penaltyAmount = (balance * 10)/100;
            balance -= penaltyAmount;
        }
        liquidityToken.transfer(owner(), balance);
    }
}
```

Recommendation

To address this issue, the contract should be redesigned to align with its stated purpose. A proper liquidity lock should prevent any withdrawals, even by the owner, during the lock period and should manage liquidity pool tokens rather than arbitrary ERC20 tokens. If the contract is not intended to lock liquidity, its name and design should be revised to accurately reflect its true purpose to avoid confusion and ensure it meets the expectations of users and stakeholders.

MTD - Misleading Tax Declaration

Criticality	Minor / Informative
Location	CRNT.sol#L46
Status	Unresolved

Description

The `CRNT` contract implements a tax mechanism in the overridden `_transfer` function that applies a `sellTax` when the recipient is the contract address and a `buyTax` for all other transfers. However, this logic is inconsistent and does not align with standard interpretations of "buy" and "sell" events. Specifically, transfers to the contract are taxed as "sells," which may not reflect actual sell behavior, and all other transfers, including wallet-to-wallet transfers, are taxed as "buys," which is conceptually incorrect. For example, sending tokens to another user is not a "buy" but still incurs the buy tax, which creates unnecessary token deductions for users in regular transactions.

```
function _transfer(address sender, address recipient, uint256
amount) internal  override  {
        uint256 taxAmount = (amount * (recipient ==
address(this) ? sellTax : buyTax)) / 100;
        if (totalSupply() - taxAmount <= burnThreshold ||
sender == STAKING_CONTRACT ||recipient == STAKING_CONTRACT
||sender == ICO_CONTRACT|| recipient== ICO_CONTRACT) {
            taxAmount = 0;
        }
        super._transfer(sender, recipient, amount - taxAmount);
        if (taxAmount > 0) {
            _burn(sender, taxAmount);
        }
    }
```

Recommendation

It is recommended to revise the tax logic to more accurately reflect the intended behavior of "buy" and "sell" taxes. Transfers unrelated to buying or selling, such as wallet-to-wallet transfers, should not incur any tax. Additionally, the tax variable names should be updated to better represent their purpose and avoid confusion. If the intention is to apply taxes during DEX transactions, the logic should include mechanisms to differentiate DEX-related events from other transfers.

MTN - Misleading Token Naming

Criticality	Minor / Informative
Location	USDC.sol#L11 USDT.sol#L11
Status	Unresolved

Description

The contracts create tokens named "USDC" and "USDT," which are the names of widely recognized and trusted stablecoins in the cryptocurrency ecosystem. This naming is misleading, as these tokens are not associated with or backed by the issuers of the original USDC and USDT tokens. Such naming can create confusion among users and developers, potentially leading to misuse or misrepresentation of these tokens in broader ecosystems or integrations.

```
constructor() Ownable(msg.sender)  ERC20("Tether USD", "USDT")
{
        _mint(msg.sender, 1_000_000 * (10 ** decimals())); //
Initial mint of 1 million USDT
    }

constructor() Ownable(msg.sender)  ERC20("USDC Token", "USDC")
{
        _mint(msg.sender, 1_000_000 * (10 ** decimals())); //
Initial mint of 1 million BUSD
    }
```

Recommendation

It is recommended to change the names and symbols of these tokens to clearly differentiate them from the widely recognized USDC and USDT stablecoins. This ensures that users and developers can distinguish these tokens from the originals and eliminates any potential for misrepresentation or confusion

RBM - Redundant Beneficiary Mechanism

Criticality	Minor / Informative
Location	Vesting.sol#L35,42
Status	Unresolved

Description

The `Vesting` contract includes a beneficiary mechanism allowing the owner to set a designated recipient for the vested tokens. However, the `releaseTokens` function is restricted by both the `onlyOwner` modifier and a requirement that the caller must also be the beneficiary. This effectively makes the beneficiary mechanism redundant, as only the owner can release tokens, and the owner must also be the beneficiary to meet the function's requirements. This design adds unnecessary complexity without providing any additional functionality.

```
function updateBeneficiary(address newBeneficiary) external
onlyOwner {
        require(newBeneficiary != address(0), "Vesting: New
beneficiary is the zero address");
        emit BeneficiaryUpdated(beneficiary, newBeneficiary);
        beneficiary = newBeneficiary;
    }


    function releaseTokens() external nonReentrant onlyOwner {
        require(msg.sender == beneficiary, "Vesting: Only the
beneficiary can release tokens");

        require(block.timestamp >= lastReleaseTimestamp + 30
days, "Vesting: 30-day interval not met");
        uint256 releaseAmount = monthlyRelease;
        require(releasedAmount + releaseAmount <=
totalAllocation, "Vesting: Allocation exceeded");

        lastReleaseTimestamp = block.timestamp;
        releasedAmount += releaseAmount;
        require(crntToken.balanceOf(address(this)) >=
releaseAmount, "Insufficient token balance in contract");
        crntToken.safeTransfer(owner(), releaseAmount);
        emit TokensReleased(owner(), releaseAmount);
    }
```

Recommendation

It is recommended to simplify the contract by either removing the beneficiary mechanism or redesigning the `releaseTokens` function to allow the designated beneficiary to release tokens independently of the owner. If the intent is to maintain the beneficiary mechanism, the function should enforce clear separation of roles and ensure the beneficiary can operate as intended without being restricted by ownership requirements. This improves clarity and aligns the implementation with its stated purpose.

RPU - Redundant Payable Usage

Criticality	Minor / Informative
Location	ZapV2.sol#L9 Router.sol#L34
Status	Unresolved

Description

The `ZapV2` contract declares the `_router` parameter in its constructor as `payable`, implying that the `Router` contract can receive Ether. However, the `Router` contract explicitly reverts any Ether sent to it through its `receive` function. This inconsistency makes the `payable` modifier redundant and creates potential confusion for readers of the code, as it suggests functionality that does not exist.

```
constructor(address payable _router) {
        router = Router(_router);
    }

receive() external payable {
        revert("Router: Cannot accept Ether");
    }
```

Recommendation

It is recommended to remove the `payable` modifier from the `_router` parameter in the `ZapV2` constructor to align with the actual behavior of the `Router` contract. This ensures the code is clear, unambiguous, and free from elements that might mislead developers or users about the contract's capabilities. Clear and consistent use of modifiers helps improve code readability and reduces the likelihood of misinterpretation.

L02 - State Variables could be Declared Constant

Criticality	Minor / Informative
Location	CRNT.sol#L10
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 burnThreshold = 36_900_000 * (10 ** 18)
```

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#L46,64 ICO.sol#L64,77 CRNT.sol#L22,23,39
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 _crntToken
address  STAKING_CONTRACT
address ICO_CONTRACT
uint256 _sellTax
uint256 _buyTax
```

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.

L08 - Tautology or Contradiction

Criticality	Minor / Informative
Location	Staking.sol#L86
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(balances[msg.sender] >= 0, "you donot have enouph
balance")
```

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.

L13 - Divide before Multiply Operation

Criticality	Minor / Informative
Location	Staking.sol#L101,105
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)
```

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.

L16 - Validate Variable Setters

Criticality	Minor / Informative
Location	Pair.sol#L21,22 CRNT.sol#L35,36
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.

```
token0 = _token0
token1 = _token1
STAKING_CONTRACT = _stakingContract
ICO_CONTRACT = _icoContract
```

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	ZapV2.sol#L2 Vesting.sol#L2 USDT.sol#L2 Staking.sol#L2 Router.sol#L2 Pair.sol#L2 LiquidityLock.sol#L2 ICO.sol#L2 Factory.sol#L2 CRNT.sol#L2
Status	Unresolved

Description

The `^` symbol indicates that any version of Solidity that is compatible with the specified version (i.e., any version that is a higher minor or patch version) can be used to compile the contract. The version lock is a mechanism that allows the author to specify a minimum version of the Solidity compiler that must be used to compile the contract code. This is useful because it ensures that the contract will be compiled using a version of the compiler that is known to be compatible with the code.

```
pragma solidity ^0.8.20;
```

Recommendation

The team is advised to lock the pragma to ensure the stability of the codebase. The locked pragma version ensures that the contract will not be deployed with an unexpected version. An unexpected version may produce vulnerabilities and undiscovered bugs. The compiler should be configured to the lowest version that provides all the required functionality for the codebase. As a result, the project will be compiled in a well-tested LTS (Long Term Support) environment.

L20 - Succeeded Transfer Check

Criticality	Minor / Informative
Location	Staking.sol#L106 LiquidityLock.sol#L27
Status	Unresolved

Description

According to the ERC20 specification, the transfer methods should be checked if the result is successful. Otherwise, the contract may wrongly assume that the transfer has been established.

```
crntToken.transferFrom(msg.sender, staker, stakerRevenue)
liquidityToken.transfer(owner(), balance)
```

Recommendation

The contract should check if the result of the transfer methods is successful. The team is advised to check the SafeERC20 library from the Openzeppelin library.

L22 - Potential Locked Ether

Criticality	Minor / Informative
Location	Router.sol#L34 LiquidityLock.sol#L17
Status	Unresolved

Description

The contract contains Ether that has been placed into a Solidity contract and is unable to be transferred. Thus, it is impossible to access the locked Ether. This may produce a financial loss for the users that have called the payable method.

```
receive() external payable {
        revert("Router: Cannot accept Ether");
    }

receive() external payable {
        revert("This contract does not accept Ether.");
    }
```

Recommendation

The team is advised to either remove the payable method or add a withdraw functionality. it is important to carefully consider the risks and potential issues associated with locked Ether.

Functions Analysis

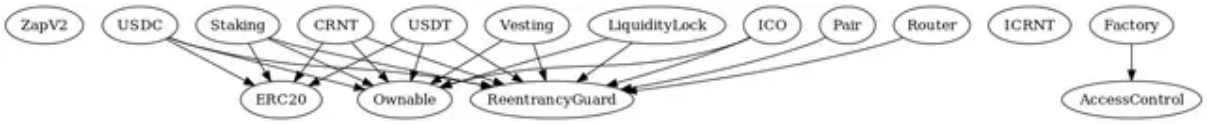
Contract	Type	Bases		
	Function Name	**Visibility**	**Mutability**	**Modifiers**
ZapV2	Implementation			
		Public	✓	-
	zapIn	External	✓	-
	zapOut	External	✓	-
Vesting	Implementation	Ownable, ReentrancyGuard		
		Public	✓	Ownable
	updateBeneficiary	External	✓	onlyOwner
	releaseTokens	External	✓	nonReentrant onlyOwner
USDT	Implementation	ERC20, Ownable, ReentrancyGuard		
		Public	✓	Ownable ERC20
	mint	External	✓	onlyOwner nonReentrant
	totalSupply	Public		-
	transfer	Public	✓	-
	approve	Public	✓	-
	transferFrom	Public	✓	-
	allowance	Public		-

USDC	Implementation	ERC20, Ownable, ReentrancyGuard		
		Public	✓	Ownable ERC20
	mint	External	✓	onlyOwner nonReentrant
	totalSupply	Public		-
	transfer	Public	✓	-
	approve	Public	✓	-
	transferFrom	Public	✓	-
	allowance	Public		-
ICRNT	Interface			
	burnFrom	External	✓	-
Staking	Implementation	ERC20, Ownable, ReentrancyGuard		
		Public	✓	Ownable ERC20
	stake	External	✓	nonReentrant
	withdraw	External	✓	-
	claimRewards	Public	✓	nonReentrant
	distributeRevenue	External	✓	onlyOwner
	_removeStaker	Internal	✓	

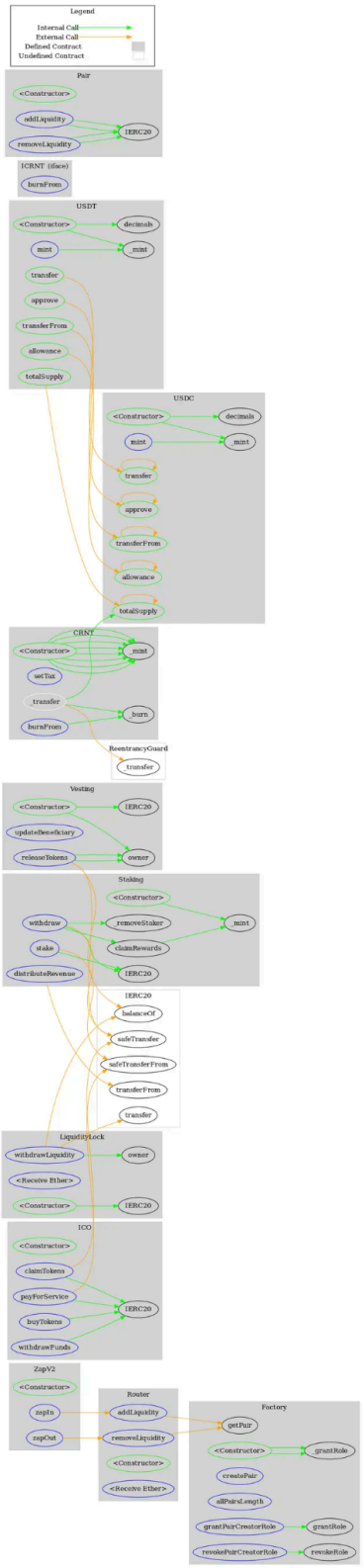
Router	Implementation	ReentrancyGuard		
		Public	✓	-
	addLiquidity	External	✓	nonReentrant
	removeLiquidity	External	✓	nonReentrant
		External	Payable	-
Pair	Implementation	ReentrancyGuard		
		Public	✓	-
	addLiquidity	External	✓	nonReentrant
	removeLiquidity	External	✓	nonReentrant
LiquidityLock	Implementation	Ownable, ReentrancyGuard		
		Public	✓	Ownable
		External	Payable	-
	withdrawLiquidity	External	✓	onlyOwner nonReentrant
ICO	Implementation	Ownable, ReentrancyGuard		
		Public	✓	Ownable
	buyTokens	External	✓	nonReentrant
	claimTokens	External	✓	nonReentrant
	payForService	External	✓	-
	withdrawFunds	External	✓	onlyOwner

Factory	Implementation	AccessControl		
		Public	✓	-
	createPair	External	✓	onlyRole
	allPairsLength	External		-
	grantPairCreatorRole	External	✓	onlyRole
	revokePairCreatorRole	External	✓	onlyRole
CRNT	Implementation	ERC20, Ownable, ReentrancyGuard		
		Public	✓	Ownable ERC20
	setTax	External	✓	onlyOwner
	_transfer	Internal	✓	
	burnFrom	External	✓	-

Inheritance Graph



Flow Graph


Legend
Internal Call
External Call
Defined Contract
Undefined Contract
Pair
<Constructor>
addLiquidity
removeLiquidity
IERC20
ICRNT (iface)
burnFrom
USDT
<Constructor>
decimals
mint
_mint
transfer
approve
transferFrom
allowance
totalSupply
USDC
<Constructor>
decimals
mint
_mint
transfer
approve
transferFrom
allowance
totalSupply
CRNT
<Constructor>
_mint
setTax
_transfer
_burn
burnFrom
ReentrancyGuard
_transfer
Vesting
<Constructor>
IERC20
updateBeneficiary
releaseTokens
owner
Staking
<Constructor>
withdraw
_removeStaker
mint
stake
claimRewards
distributeRevenue
IERC20
IERC20
balanceOf
safeTransfer
safeTransferFrom
transferFrom
transfer
LiquidityLock
withdrawLiquidity
owner
<Receive Ether>
<Constructor>
IERC20
ICO
<Constructor>
claimTokens
payForService
buyTokens
withdrawFunds
IERC20
ZapV2
<Constructor>
zapIn
zapOut
Router
addLiquidity
removeLiquidity
<Constructor>
<Receive Ether>
Factory
getPair
<Constructor>
grantRole
createPair
allPairsLength
grantPairCreatorRole
grantRole
revokePairCreatorRole
revokeRole

Summary

creationnetwork.ai contract implements a tokens, 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