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
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

Source Files

Filename	SHA256
Vesting.sol	4d82d6238e9fc669133c846a1a7bbdb044992507fdcccc3fb2572a13e7a207d8
USDTs.sol	f2f79d8a70271573f9d8f0b9cf1a776515c1f6ad4f775da20517a46fcde095c2
USDCs.sol	56d6ec2f478f0736af779b6226e5729a826e1d6a07999c176c4bd96796e26583
Staking.sol	2ab7b202e6d2aca914bd88b06fa929b28a3181bcc1caf65bb3bcad758a8c524a
Router.sol	bbfe7e108a2a63c3db4060c074447218819016b9f452381f15fec4b4e7055c0d
Pair.sol	2fee8cb9fac07a5b29b5107f78ec6f5a49b52e4502ac0c288b340eb9e2b04d61

LiquidityLock.sol	d1970dae8d2834f4288169f8f5d43a3403bb87b9725ff6de54b0a8c4e89195d8
ICO.sol	810575e04aeaf776e2fe91ad3ba1576532fc67c546deb4a78384279f0bd9b40b
Factory.sol	1d3189d4e554677aec96982ffbe14fd6adac13cc5ba91d516a31bd87b2c4ea2a
CRNT.sol	04a16ef0b33f0c0d28f061745fffc59560898c5d0e86b85a024348a4eec3c66b

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 the owner 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. Users can also exchange tokens utilizing the deployed pairs.

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. It does not implement mechanisms for minting or managing LP tokens, focusing solely on maintaining token reserves. It implements an exchanging mechanism with price derived from the held reserves.

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 "USDC Tokens" with the symbol "USDCs."

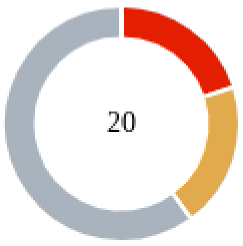
USDTs Contract Functionality

The USDTs contract is an ERC20 token named "Tether USDs" with the symbol "USDTs."

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	4
●	Medium	4
●	Minor / Informative	12

Severity		Unresolved	Acknowledged	Resolved	Other
●	Critical	4	0	0	0
●	Medium	4	0	0	0
●	Minor / Informative	12	0	0	0

Diagnostics

🔴 Critical 🟠 Medium ⚪ Minor / Informative

Severity	Code	Description	Status
🔴	DDI	Decimal Division Inconsistency	Unresolved
🔴	IPI	Incorrect Pair Implementation	Unresolved
🔴	FLV	Flash Loan Vulnerability	Unresolved
🔴	PPM	Potential Price Manipulation	Unresolved
🟠	IPA	Improper Penalty Application	Unresolved
🟠	LRI	Liquidity Rewards Inconsistency	Unresolved
🟠	PTAI	Potential Transfer Amount Inconsistency	Unresolved
🟠	UOA	Unverified Output Amount	Unresolved
⚪	CCS	Commented Code Segments	Unresolved
⚪	CCR	Contract Centralization Risk	Unresolved
⚪	MLLI	Misleading Liquidity Lock Implementation	Unresolved
⚪	MTD	Misleading Tax Declaration	Unresolved
⚪	MTN	Misleading Token Naming	Unresolved
⚪	MU	Modifiers Usage	Unresolved

●	RCS	Redundant Conditional Statements	Unresolved
●	L04	Conformance to Solidity Naming Conventions	Unresolved
●	L13	Divide before Multiply Operation	Unresolved
●	L16	Validate Variable Setters	Unresolved
●	L19	Stable Compiler Version	Unresolved
●	L22	Potential Locked Ether	Unresolved

DDI - Decimal Division Inconsistency

Criticality	Critical
Location	Pair.sol#L64
Status	Unresolved

Description

Division of decimal (fixed-point) numbers can result in rounding errors due to the way division is implemented in Solidity, potentially leading to issues with precise calculations. Specifically, the `Pair` contract relies on the provided liquidity reserves to calculate the price of one token relative to another based on the ratio of their supplies. If the tokens involved have different decimal places (common for stablecoins), the division may return zero values if the token with the lowest decimal is in the numerator. This type of inconsistency can lead to significant vulnerabilities, such as potential of price manipulation and loss of funds.

```
function getToken0ToToken1Price() public nonReentrant returns (uint256) {
    uint256 token0Balance = IERC20(token0).balanceOf(address(this));
    uint256 token1Balance = IERC20(token1).balanceOf(address(this));
    require(token1Balance > 0, "token1 balance is zero");
    require(token0Balance > 0, "Token0 balance is zero");
    return token0Balance / token1Balance;
}
```

Recommendation

The team is advised to take into consideration the rounding results that are produced from the solidity calculations. Normalizing the decimal precision to the higher number could help resolve the issue.

IPI - Incorrect Pair Implementation

Criticality	Critical
Location	Pair.sol#L37
Status	Unresolved

Description

The `Pair` contract does not accurately implement standard liquidity management mechanisms commonly found in decentralized exchanges like Uniswap. Although 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, which is crucial for ensuring price stability. Specifically, using variables such as `amount0ForUser` and `amount1ForUser` to trace and withdraw deposited amounts may not accurately reflect the actual reserves in the pool and cannot be relied upon for withdrawing liquidity as the reserves may not suffice after exchange operations.

```
function addLiquidity(address to, uint256 amount0, uint256 amount1)
external nonReentrant {
        require(amount0 > 0 && amount1 > 0, "Pair:
INVALID_AMOUNTS");
        reserve0 += amount0;
        reserve1 += amount1;
        amount0ForUser[to] += amount0;
        amount1ForUser[to] += amount1;
        if (addedTime[to] == 0) {
            addedTime[to] = block.timestamp;
        }
        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;

        amount0ForUser[to] -= amount0;
        amount1ForUser[to] -= 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.

FLV - Flash Loan Vulnerability

Criticality	Critical
Location	Pair.sol#L110
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(address to) public nonReentrant {
    // uint256 totalShares = reserve0 + reserve1;
    uint256 totalShares = amount0ForUser[to] + amount1ForUser[to];
    require(totalShares > 0, "Pair: No liquidity added for the user");

    uint256 blockTime = block.timestamp - addedTime[to];
    require(blockTime >= 30 days,"reward claim period not reached");

    uint256 reward = (totalShares * blockTime * REWARD_RATE) /
            (100 * SECONDS_IN_YEAR);
    require(reward>0,"Pair: No reward token available");
    addedTime[to] = block.timestamp;

    _mint(to, reward);
   }
```

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.

PPM - Potential Price Manipulation

Criticality	Critical
Location	Pair.sol#L37
Status	Unresolved

Description

The Pair contract allows users to deposit and withdraw funds at arbitrary ratios, which can directly impact token prices and increase the potential for price manipulation. Altering the pool's reserves without maintaining a constant ratio can introduce severe vulnerabilities and attack vectors.

```
function addLiquidity(address to, uint256 amount0, uint256 amount1) external
nonReentrant {
        require(amount0 > 0 && amount1 > 0, "Pair: INVALID_AMOUNTS");
        reserve0 += amount0;
        reserve1 += amount1;
        amount0ForUser[to] += amount0;
        amount1ForUser[to] += amount1;
        if (addedTime[to] == 0) {
            addedTime[to] = block.timestamp;
        }
        IERC20(token0).safeTransferFrom(to, address(this), amount0);
        IERC20(token1).safeTransferFrom(to, address(this), amount1);
        emit Mint(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 can add and remove liquidity at predentermined rations ensuring price stability.

IPA - Improper Penalty Application

Criticality	Medium
Location	Staking.sol#L71
Status	Unresolved

Description

The contract implements a penalty for stakers who choose to withdraw before the end of a lockup period. In this implementation, the penalty is stored as a state variable. This means that once a value is stored in the penaltyAmount variable, the respective amount will be subtracted from a user's balance, even if the user is not eligible for a penalty.

```
function withdraw(uint256 amount) external {
    ...
    if (block.timestamp < stakedfromTS[msg.sender] + LOCK_PERIOD) {
    penaltyAmount = (amount * 10) / 100; // 10% of amount penalty
    balances[msg.sender] -= penaltyAmount;
    ICRNT(crntAddress).burnFrom(penaltyAmount);
    }

    balances[msg.sender] -= (amount - penaltyAmount);
    ...
}
```

Recommendation

The team is advised to revise the implementation of the penalty mechanism to ensure that only addresses eligible for penalties are affected.

LRI - Liquidity Rewards Inconsistency

Criticality	Medium
Location	Pair.sol#L37
Status	Unresolved

Description

The Pair contract is designed to reward users who provide liquidity. It tracks the time of the initial deposit but fails to update this timestamp for any additional deposits. This creates a loophole where a user can make a small initial deposit, wait for a significant amount of time, and then make a larger deposit for a short duration. Because the rewards are calculated based on the time elapsed since the first deposit, the user can claim substantial rewards for the short-term larger deposit, despite the initial deposit being minimal. This results in reward tokens being issued that do not accurately correspond to the actual liquidity provided, potentially skewing the reward distribution.

```
function addLiquidity(address to, uint256 amount0, uint256 amount1) external
nonReentrant {
    require(amount0 > 0 && amount1 > 0, "Pair: INVALID_AMOUNTS");
    reserve0 += amount0;
    reserve1 += amount1;
    amount0ForUser[to] += amount0;
    amount1ForUser[to] += amount1;
    if (addedTime[to] == 0) {
        addedTime[to] = block.timestamp;
    }
    IERC20(token0).safeTransferFrom(to, address(this), amount0);
    IERC20(token1).safeTransferFrom(to, address(this), amount1);
    emit Mint(to, amount0, amount1);
}
```

```
function claimRewards(address to) public nonReentrant {
    // uint256 totalShares = reserve0 + reserve1;
    uint256 totalShares = amount0ForUser[to] + amount1ForUser[to];
    require(totalShares > 0, "Pair: No liquidity added for the user");

    uint256 blockTime = block.timestamp - addedTime[to];
    require(blockTime >= 30 days,"reward claim period not reached");

    uint256 reward = (totalShares * blockTime * REWARD_RATE) /
            (100 * SECONDS_IN_YEAR);
    require(reward>0,"Pair: No reward token available");
    addedTime[to] = block.timestamp;

    _mint(to, reward);
}
```

Recommendation

To prevent this issue, it is advised that rewards be automatically claimed each time a new deposit is made or withdrawn, and the timestamp should be reset at that instance.

PTAI - Potential Transfer Amount Inconsistency

Criticality	Medium
Location	Pair.sol#L64
Status	Unresolved

Description

The `transfer()` and `transferFrom()` functions are used to transfer a specified amount of tokens to an address. 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 `exchange()` function expects the received amount to be equal to `fromAmount`. However, this assumption is incorrect if the token implements transfer fees. As a result, a significant inconsistency arises during the swap process.

```
function exchange(
        address fromToken,
        uint256 fromAmount,
        address toToken
    ) public {
    ...
     if (fromToken == token0) {
            toAmount = amountInWithFee / (exchangeRate);
            reserve0 += amountInWithFee;
            reserve1 -= toAmount;
        } else {
            toAmount = amountInWithFee * (exchangeRate);
            reserve1 += amountInWithFee;
            reserve0 -= toAmount;
        }
    ...
}
```

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`

UOA - Unverified Output Amount

Criticality	Medium
Location	Pair.sol#L73
Status	Unresolved

Description

The contract implements the exchange function to swap an input token for an output token. However, it lacks checks to ensure that the output amount meets a minimum threshold set by the user. This omission could allow exchanges to be front-run, exposing users to potential loss of funds.

```
function exchange(
    address fromToken,
    uint256 fromAmount,
    address toToken
) public {
...
IERC20(toToken).safeTransfer(msg.sender, toAmount);
}
```

Recommendation

The team is advised to implement a conditional check within the `exchange` function to ensure that the output amount meets or exceeds a minimum threshold specified by the user. This will help protect users from front-running attacks and potential losses.

CCR - Contract Centralization Risk

Criticality	Minor / Informative
Location	CRNT.sol#L72 Vesting.sol#L35,42 Staking.sol#L106,139 LiquidityLock.sol#L22 ICO.sol#L87,95
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 {
    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.

CCS - Commented Code Segments

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

Description

The contract contains commented code segments that are never executed. These segments increase the code size and hinder readability. Removing these segments will improve code clarity and maintainability.

```
// 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

The team is advised to remove all commented and unused code segments to enhance readability and reduce the overall code size, thereby improving maintainability.

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.

```solidity
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.20;
import
"https://github.com/OpenZeppelin/openzeppelin-contracts/blob/v4.9.0
/contracts/access/Ownable.sol";
import
"https://github.com/OpenZeppelin/openzeppelin-contracts/blob/v4.9.0
/contracts/token/ERC20/ERC20.sol";
import
"https://github.com/OpenZeppelin/openzeppelin-contracts/blob/v4.9.0
/contracts/security/ReentrancyGuard.sol";
import
"https://github.com/OpenZeppelin/openzeppelin-contracts/blob/v4.9.0
/contracts/token/ERC20/utils/SafeERC20.sol";

contract LiquidityLock is Ownable, ReentrancyGuard {
    using SafeERC20 for IERC20;
    IERC20 public immutable liquidityToken;
    uint256 public immutable lockTimestamp;
    uint256 public constant LOCK_PERIOD = 21 days;

    constructor(address _liquidityToken) Ownable() {
        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.safeTransfer(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#L90
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 = 0;

    // Whitelisted addresses should not incur taxes
    if (
        sender != STAKING_CONTRACT &&
        recipient != STAKING_CONTRACT &&
        sender != ICO_CONTRACT &&
        recipient != ICO_CONTRACT &&
        sender != owner() &&
        recipient != owner()
    ) {
        if (recipient == address(this)) {
            // Treat transfers to the contract address as "sells"
            taxAmount = (amount * sellTax) / 100;
        } else if (sender == address(this)) {
            // Treat transfers from the contract address as "buys"
            taxAmount = (amount * buyTax) / 100;
        }
    }
...
}
```

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	USDTs.sol#L11 USDCs.sol#L11
Status	Unresolved

Description

The contracts create tokens named "USDCs" and "USDTs," which are the names that resemble 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()  ERC20("USDC Tokens", "USDCs")   {
        _mint(msg.sender, 1_000_000 * (10 ** decimals())); //
Initial mint of 1 million BUSD
    }

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

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

MU - Modifiers Usage

Criticality	Minor / Informative
Location	CRNT.sol#L66,67 Pair.sol#L32 Router.sol#L20,28,36,43 Staking.sol#L47,73 Vesting.sol#L36
Status	Unresolved

Description

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

```
require(crntAddress != address(0),"crntAddress not set");
```

Recommendation

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

RCS - Redundant Conditional Statements

Criticality	Minor / Informative
Location	Pair.sol#L78
Status	Unresolved

Description

The contract contains redundant conditional statements that can be simplified to improve code efficiency and performance. Redundant conditional statements may lead to larger code size, increased memory usage, and slower execution times. Such redundancies are common when simple comparisons or checks are performed within conditional statements, leading to redundant operations.

```
require(fromToken == token0 || fromToken == token1,"Invalid fromToken.");
require(toToken == token0 || toToken == token1, "Invalid toToken.");
require(fromToken != toToken, "Tokens must be different");
```

Recommendation

It is recommended to refactor conditional statements that return results by eliminating unnecessary code structures. This practice minimizes the number of operations required, reduces the code footprint, and optimizes memory and gas usage. Simplifying such statements makes the code more readable and improves its overall performance.

L04 - Conformance to Solidity Naming Conventions

Criticality	Minor / Informative
Location	Staking.sol#L139 ICO.sol#L92 CRNT.sol#L27,28,72
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 immutable STAKING_CONTRACT
address immutable 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.

L13 - Divide before Multiply Operation

Criticality	Minor / Informative
Location	Staking.sol#L110,117 Pair.sol#L90,100
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 = (fromAmount * fee) / 1000
toAmount = amountInWithFee * (exchangeRate)
```

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	Staking.sol#L140 ICO.sol#L93
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.

```
crntAddress = _crnt
```

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	Vesting.sol#L2 Staking.sol#L2 Pair.sol#L2 LiquidityLock.sol#L2 ICO.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.

L22 - Potential Locked Ether

Criticality	Minor / Informative
Location	Router.sol#L47
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");
    }
```

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

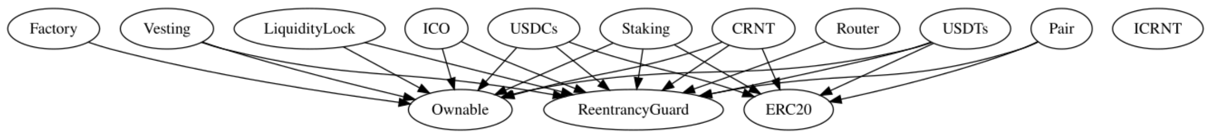
Contract	Type	Bases		
	Function Name	**Visibility**	**Mutability**	**Modifiers**
Vesting	Implementation	Ownable, ReentrancyGuard		
		Public	✓	Ownable
	updateBeneficiary	External	✓	onlyOwner
	releaseTokens	External	✓	nonReentrant
USDTs	Implementation	ERC20, Ownable, ReentrancyGuard		
		Public	✓	Ownable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

		Public	✓	Ownable ERC20
	stake	External	✓	nonReentrant
	withdraw	External	✓	-
	claimRewards	Public	✓	nonReentrant
	distributeRevenue	External	✓	onlyOwner
	_removeStaker	Internal	✓	
	setCrntAddress	External	✓	onlyOwner
Router	Implementation	ReentrancyGuard		
		Public	✓	-
	addLiquidity	External	✓	nonReentrant
	removeLiquidity	External	✓	nonReentrant
	exchange	External	✓	nonReentrant
	claimRewards	External	✓	nonReentrant
		External	Payable	-
		External	Payable	-
Pair	Implementation	ERC20, ReentrancyGuard		
		Public	✓	ERC20
	addLiquidity	External	✓	nonReentrant
	removeLiquidity	External	✓	nonReentrant
	getToken0ToToken1Price	Public	✓	nonReentrant

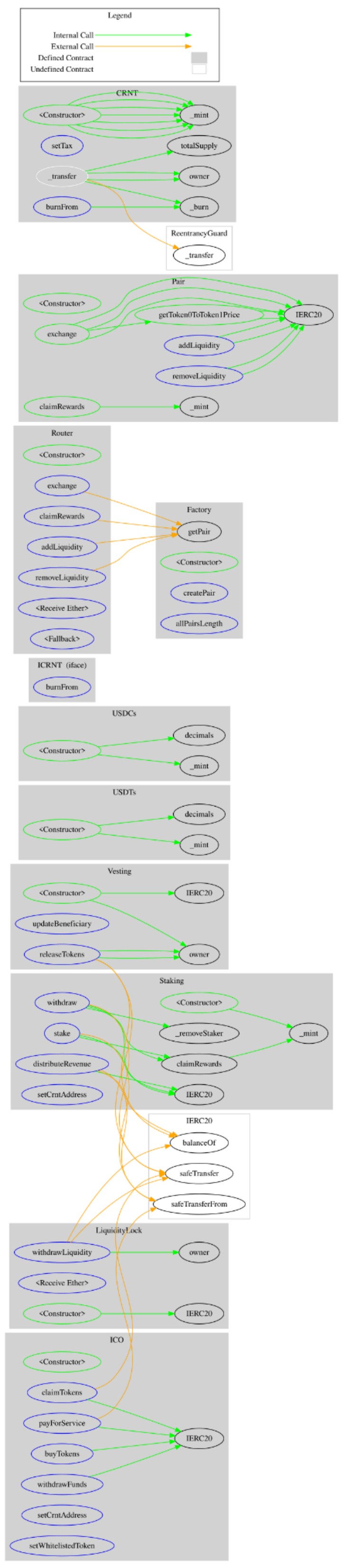
	exchange	Public	✓	-
	claimRewards	Public	✓	nonReentrant
LiquidityLock	Implementation	Ownable, ReentrancyGuard		
		Public	✓	Ownable
		External	Payable	-
	withdrawLiquidity	External	✓	onlyOwner nonReentrant
ICO	Implementation	Ownable, ReentrancyGuard		
		Public	✓	Ownable
	buyTokens	External	✓	nonReentrant
	claimTokens	External	✓	nonReentrant
	payForService	External	✓	-
	withdrawFunds	External	✓	onlyOwner
	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	✓	onlyOwner
	_transfer	Internal	✓	
	burnFrom	External	✓	-

Inheritance Graph



Flow Graph



Legend
Internal Call
External Call
Defined Contract
Undefined Contract
CRNT
<Constructor>
_mint
setTax
totalSupply
_transfer
owner
burnFrom
_burn
ReentrancyGuard
_transfer
Pair
<Constructor>
getToken0ToToken1Price
IERC20
exchange
addLiquidity
removeLiquidity
claimRewards
_mint
Router
<Constructor>
exchange
claimRewards
addLiquidity
removeLiquidity
<Receive Ether>
<Fallback>
Factory
getPair
<Constructor>
createPair
allPairsLength
ICRNT (iface)
burnFrom
USDCs
decimals
<Constructor>
_mint
USDTs
decimals
<Constructor>
_mint
Vesting
<Constructor>
IERC20
updateBeneficiary
releaseTokens
owner
Staking
withdraw
<Constructor>
stake
_removeStaker
_mint
distributeRevenue
claimRewards
setCrntAddress
IERC20
IERC20
balanceOf
safeTransfer
safeTransferFrom
LiquidityLock
withdrawLiquidity
owner
<Receive Ether>
<Constructor>
IERC20
ICO
<Constructor>
claimTokens
payForService
IERC20
buyTokens
withdrawFunds
setCrntAddress
setWhitelistedToken

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