Cyberscope

Audit Report

Creationnetwork

December 2023

Files CRNT.sol,ICO.sol,ZapV2.sol,Whitelist.sol,Referral.sol,

Audited by © cyberscope

Table of Contents

Review

Contract Name	Creation Network
Symbol	CRNT
Decimals	18
Total Supply	369,000,000

Audit Updates

Initial Audit	12 Dec 2023

Source Files

Filename	SHA256
ZapV2.sol	40c797ecf1069ce86c872267a6f414b4f628592b24995e1e97389003c234dfa1
Whitelist.sol	9896d92ca616ac05ffdc04ed09fdaf41b5646def71ecf43c17c650a934a908e7
USDT.sol	88b57e6495b8dc3b0f3c6066e205e5deb6dafcf7ad9f7b5872a54113fad7707a
Router.sol	911e231768ec256f70f70434188b99c6a01ac9ee1b3b5480776e33d27879eb9e
Referral.sol	98bfd88d96a518e5d80d8c4e339a250362eb190e1c7ca5aa5aca1bb95e00a5d7
Pair.sol	24dbf3add96dc0660ef263364ba247c196e916a1a9255f8d0e227bef67368ead
Ownable.sol	33422e7771fefe5fbfe8934837515097119d82a50eda0e49b38e4d6a64a1c25d

IERC20.sol	9b3ddf6c1c9da600aa996789d63e71075ac93334bc79bb7684096bf6d460a1ae
ICO.sol	00fec9d5d1c4c68d6f6f58e0020ca6acd4b0a990c24f758c345f269428401845
Factory.sol	1071f5a1fd5a45bda51c9084194b49c9d75547b12c1ee6b87f55d9a1c1cdcba3
Context.sol	b2cfee351bcafd0f8f27c72d76c054df9b571b62cfac4781ed12c86354e2a56c
CRNT.sol	86f97fa40313a445d57b457a54a79bc1a7ef975b98852d01785546b2e54c0ad1
BUSD.sol	2472214feb8afe722b2b012b3e3df7d5a2ee7e30ec59d3deb50bbcbec55cf84c

Overview

This document provides the overview of the smart contract audit conducted for the "Creationnetwork” project. The project implements for a token sale with multiple stages, including seed sale, presale, and public sale. It leverages the ERC20 standard for token transactions.

Functionality

Token Sale Mechanics

The contract manages the sale of "CRNT" tokens with allocations for different sale stages and distinct pricing for each stage.

Spending Limits

Implements spending limits for participants, restricting the number of tokens purchasable based on the stage of the sale and ensuring compliance with the set buying limit.

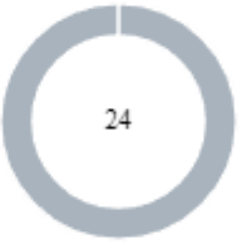
Dynamic Stage Progression

The contract automatically progresses through different stages of the sale based on time and allocation limits.

Referral System

Includes a referral mechanism, rewarding participants for referring new buyers.

Findings Breakdown



Critical	0
Medium	0
Minor / Informative	24

Severity	Unresolved	Acknowledged	Resolved	Other
Critical	0	0	0	0
Medium	0	0	0	0
Minor / Informative	24	0	0	0

Diagnostics

Critical Medium Minor / Informative

Severity	Code	Description	Status
	MT	Mints Tokens	Unresolved
	MTAS	Minimum Token Amount Stuck	Unresolved
	CCR	Contract Centralization Risks	Unresolved
	IDPE	Integer Division Precision Error	Unresolved
	DPI	Decimals Precision Inconsistency	Unresolved
	VO	Variable Optimization	Unresolved
	LVR	Local Variable Redundancy	Unresolved
	RCC	Redundant Conditional Checks	Unresolved
	CR	Code Repetition	Unresolved
	PTAI	Potential Transfer Amount Inconsistency	Unresolved
	AETA	Approve Excessive Token Amounts	Unresolved
	RSML	Redundant SafeMath Library	Unresolved
	IDI	Immutable Declaration Improvement	Unresolved
	L02	State Variables could be Declared Constant	Unresolved

●	L04	Conformance to Solidity Naming Conventions	Unresolved
●	L08	Tautology or Contradiction	Unresolved
●	L09	Dead Code Elimination	Unresolved
●	L11	Unnecessary Boolean equality	Unresolved
●	L14	Uninitialized Variables in Local Scope	Unresolved
●	L16	Validate Variable Setters	Unresolved
●	L17	Usage of Solidity Assembly	Unresolved
●	L18	Multiple Pragma Directives	Unresolved
●	L19	Stable Compiler Version	Unresolved
●	L20	Succeeded Transfer Check	Unresolved

MT - Mints Tokens

Criticality	Minor / Informative
Location	CRNT.sol#L529
Status	Unresolved

Description

The contract owner has the authority to mint tokens, up to a certain limit defined by `supplyLeft` variable. The owner may take advantage of it by calling the `mint` function. The team should carefully manage the balance between the burned and the minted tokens. If the `supplyLeft` decreases dramatically, then the owner will not be able to recover the burned tokens.

```
function mint(address account, uint256 amount) public onlyOwner
{
  require(supplyLeft - amount >= 0, "Limit Exceeding");

  require(account != address(0), "BEP20: mint to the zero
address");

  _beforeTokenTransfer(address(0), account, amount);

    _totalSupply = _totalSupply + amount;
  supplyLeft = supplyLeft - amount;
  unchecked {
      // Overflow not possible: balance + amount is at most
totalSupply + amount, which is checked above.
      _balances[account] += amount;
  }
  emit Transfer(address(0), account, amount);

  _afterTokenTransfer(address(0), account, 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.

MTAS - Minimum Token Amount Stuck

Criticality	Minor / Informative
Location	CRNT.sol#L338,398
Status	Unresolved

Description

In the `transfer` and the `transferFrom` function, a portion of the transferred amount (5%) is automatically sent to the `zapInToken` function for liquidity pool addition. However, the `zapInToken` function has a require statement that checks if the amount is greater than or equal to a predefined minimum amount. This creates a scenario where, if the 5% amount deducted in the transfer functions is below this MIN_AMT threshold, the `zapInToken` call will fail due to the require check. Consequently, the entire `transfer` transaction will revert, potentially leading to a situation where smaller transactions consistently fail.

```
function transfer(
        address to,
        uint256 amount
    ) public virtual override returns (bool) {
        uint256 tenPercentAmount = (amount * 100) / 1000; //
Calculate 10% tax amount
        uint256 fivePercent = tenPercentAmount / 2; //  5% tax
amount for liquidity and 5% tax amount for burn
        uint256 amountAfterTax = amount - tenPercentAmount;
        _transfer(_msgSender(), to, amountAfterTax);
        _transfer(_msgSender(), address(this), fivePercent);
        IBEP20(address(this)).approve(address(zapper),
fivePercent);
        zapper.zapInToken(address(this), fivePercent, poolAdd,
router, owner()); //add to pool
        _burn(_msgSender(), fivePercent);
        return true;
    }

function zapInToken(
        address _from,
        uint amount,
        address _to,
        address routerAddr,
        address _recipient
    ) external {
        require(amount >= MIN_AMT, "AMOUNT TOO SMALL");
        // From an ERC20 to an LP token, through specified
router, going through base asset if necessary
        IERC20(_from).safeTransferFrom(msg.sender,
address(this), amount);
        if (FEE_RATE != 0) {
            uint feeAmount = amount.mul(FEE_RATE).div(10000);
            IERC20(_from).safeTransfer(FEE_TO_ADDR, feeAmount);
            amount = amount.sub(feeAmount);
        }
        // we'll need this approval to add liquidity
        _approveTokenIfNeeded(_from, routerAddr);
        _swapTokenToLP(_from, amount, _to, _recipient,
routerAddr);
    }
```

Recommendation

It is recommended to reevaluate and adjust the `MIN_AMT` check within the `zapInToken` function. This could involve either conducting the check before any token transfer occurs or revising the function's logic to accommodate transactions involving

amounts below the `MIN_AMT` threshold, potentially by bypassing the liquidity addition process for these smaller amounts. Additionally, implementing a safeguard within the transfer function is advisable. This safeguard should ensure that the 5% amount being redirected to the `zapInToken` function meets or exceeds the `MIN_AMT` requirement.

CCR - Contract Centralization Risks

Criticality	Minor / Informative
Status	Unresolved

Description

In the `initialize` function, several addresses are initialized. These addresses, which include token contract addresses (such as CRNT, BUSD, USDT), along with whitelist and referral system addresses, are set directly by the contract owner. Given the lack of an inherent verification mechanism, there is a significant risk that these addresses could be set incorrectly or, worse, maliciously. These addresses are integral to the contract's key operations, including token purchases and referral mechanisms. If a malicious or incorrect address is input, the contract could inadvertently interact with malicious entities or execute unintended operations, leading to financial losses and a breakdown of the intended functionalities.

```
function initialize(
    address _whitelistAddress,
    address _referralAddress,
    address _crntAddress,
    address _busdAddress,
    address _usdtAddress
) public onlyOwner {
    require(initialized == false, "Already started");
    currentStage = 1;
    initialized = true;
    startTime = block.timestamp;
    presaleStartTime = startTime + 15 days;
    publicSaleStartTime = startTime + 30 days;
    busdAddress = _busdAddress;
    usdtAddress = _usdtAddress;
    crntAddress = _crntAddress;
    whitelistAddress = _whitelistAddress;
    referralAddress = _referralAddress;
}
```

Recommendation

To mitigate these risks, it is imperative that the contract incorporates a robust mechanism for verifying the legitimacy of all critical external addresses before they are set. This can be achieved by using pre-verified and well-known addresses, establishing registries of approved addresses, or implementing other forms of validation checks. Additionally, the contract should enhance its access control measures. This could include implementing multi-signature requirements or other stringent checks to ensure that only authorized and validated changes are made.

IDPE - Integer Division Precision Error

Criticality	Minor / Informative
Location	CRNT.sol#L338,398
Status	Unresolved

Description

`transfer` and `transferFrom` functions exhibit a precision limitation issue in their tax calculation mechanism. Specifically, the functions computes a 10% tax on the transaction amount, which is then divided by two to allocate equal parts for liquidity addition and token burning. However, arithmetic operations are inherently handled using integer division, which lacks decimal precision. This approach can lead to imprecise outcomes, particularly evident when the 10% tax calculation results in an odd number. For example, a 10% tax amounting to 9 tokens, when halved, should yield 4.5 tokens for each purpose. Yet, due to integer division, this is truncated to 4 tokens, resulting in an unintended discrepancy. This precision issue can cumulatively distort the intended distribution of funds between liquidity and burning, deviating from the contract's economic design and potentially leading to long-term imbalances in token dynamics.

```
uint256 tenPercentAmount = (amount * 100) / 1000; // Calculate
10% tax amount
uint256 fivePercent = tenPercentAmount / 2; //  5% tax amount
for liquidity and 5% tax amount for burn
```

Recommendation

To rectify this precision issue, a reevaluation of the tax calculation methodology in the `transfer` functions is recommended.

DPI - Decimals Precision Inconsistency

Criticality	Minor / Informative
Location	ICO.sol#L120
Status	Unresolved

Description

The decimals field of a contract's ERC20 token can be used to specify the number of decimal places that the token uses. For example, if decimals are set to `8`, it means that the smallest unit of the token is `0.00000001`, and if decimals are set to `18`, it means that the smallest unit of the token is `0.000000000000000001`.

However, there is an inconsistency in the way that the decimals field is handled in some ERC20 contracts. The ERC20 specification does not specify how the decimals field should be implemented, and as a result, some contracts use different precision numbers.

This inconsistency can cause problems when interacting with these contracts, as it is not always clear how the decimals field should be interpreted. For example, if a contract expects the decimals field to be 18 digits, but the contract being interacted with uses 8 digits, the result of the interaction may not be what was expected.

```
function calculateReceivingAmount(
    uint256 dollarAmount,
    uint256 stage
) private view returns (uint256 buyerReceives) {
    if (stage == 1) {
        buyerReceives = ((dollarAmount * 10 ** 18) /
seedSalePrice);
        if ((buyerReceives + seedsaleMinted) >
seedSaleAllocation) {
            revert Sale_Limit_Exceeding();
        }
        return buyerReceives;
    } else if (stage == 2) {
        buyerReceives = ((dollarAmount * 10 ** 18) /
preSalePrice);
        if ((buyerReceives + presaleMinted) >
preSaleAllocation) {
            revert Sale_Limit_Exceeding();
        }
        return buyerReceives;
    } else if (stage == 3) {
        buyerReceives = ((dollarAmount * 10 ** 18) /
publicSalePrice);

        if ((buyerReceives + publicSaleMinted) >
publicSaleAllocation) {
            revert Sale_Limit_Exceeding();
        }
        return buyerReceives;
    }
}
```

Recommendation

To avoid these issues, it is important to carefully review the implementation of the decimals field of the underlying tokens. The team is advised to normalize each decimal to one single source of truth. A recommended way is to scale all the decimals to the greatest token's decimal. Hence, the contract will not lose precision in the calculations.

All the decimals could be normalized to 18 since it represents the ERC20 token with the greatest digits.

VO - Variable Optimization

Criticality	Minor / Informative
Location	ICO.sol#L61Whitelist.sol#L19
Status	Unresolved

Description

Storage variable `currentStage` is declared as `uint256`, however is designed to store small integer values (0,1,2 or 3). The use of uint256, a 256-bit unsigned integer, is not the most efficient choice for such limited-range values. This choice leads to suboptimal use of storage space and, consequently, higher gas costs for operations involving this variable.

Additionally, the declaration of iterator `i` as uint256 in the `addToWhitelist` function is not necessary, since `numAddressesWhitelisted` is declared as `uint8`.

```
uint256 public currentStage = 0;

for (uint256 i; i < _addresses.length; ++i) {
```

Recommendation

Optimize both variables by changing their type from `uint256` to `uint8`. The `uint8` type, an 8-bit unsigned integer, is more suitable for the range of values required and will consume less storage space.

LVR - Local Variable Redundancy

Criticality	Minor / Informative
Location	ICO.sol#L225,236,248
Status	Unresolved

Description

Within the `buyTokens` function, there is an unnecessary creation of local variables that simply duplicate existing state variables. Specifically, the local variable `stage` is assigned the value of the state variable `currentStage` and then used in subsequent function calls. This redundancy adds unnecessary complexity and can potentially lead to confusion in code maintenance.

```
uint256 stage = currentStage;
stage1n2Buying(buyingWith, dollarAmount, referralCode, stage);

uint256 stage = currentStage;
stage1n2Buying(buyingWith, dollarAmount, referralCode, stage);

uint256 stage = currentStage;
stage3Buying(buyingWith, dollarAmount, referralCode, stage);
```

Recommendation

It is recommended to directly use the `currentStage` variable in the function calls instead of creating a local stage variable. This change will simplify the function logic and improve the clarity of the code.

RCC - Redundant Conditional Checks

Criticality	Minor / Informative
Location	ICO.sol#L227,240
Status	Unresolved

Description

In Ico.sol, multiple instances of redundant conditional checks have been identified. These redundancies occur in various else if blocks throughout the contract, where common conditions are repetitively checked in conjunction with mutually exclusive conditions.

```
else if (
    ((block.timestamp >= presaleStartTime) &&
        (block.timestamp < publicSaleStartTime) &&
        (presaleMinted < preSaleAllocation)) ||
    ((seedsaleMinted == seedSaleAllocation) &&
        (block.timestamp < publicSaleStartTime) &&
        (presaleMinted < preSaleAllocation))
)

else if (
    (((block.timestamp >= publicSaleStartTime) &&
        block.timestamp <= (startTime + 45 days)) &&
        publicSaleMinted < publicSaleAllocation) ||
    (presaleMinted == preSaleAllocation &&
        block.timestamp <= (startTime + 45 days) &&
        publicSaleMinted < publicSaleAllocation)
)
```

Recommendation

It is recommended to refactor these conditional statements to optimize for efficiency and readability. This refactoring will result in cleaner, more maintainable code and reduce the cost of gas during execution.

CR - Code Repetition

Criticality	Minor / Informative
Location	ZapV2.sol#L1669
Status	Unresolved

Description

There are repetitive code segments. Potential issues can arise when using code segments in Solidity. Some of them can lead to issues like gas efficiency, complexity, readability, security, and maintainability of the source code. It is generally a good idea to try to minimize code repetition where possible.

```
function transfer(
    address to,
    uint256 amount
) public virtual override returns (bool) {
    uint256 tenPercentAmount = (amount * 100) / 1000; //
Calculate 10% tax amount
    uint256 fivePercent = tenPercentAmount / 2; //  5% tax
amount for liquidity and 5% tax amount for burn
    uint256 amountAfterTax = amount - tenPercentAmount;
    _transfer(_msgSender(), to, amountAfterTax);
    _transfer(_msgSender(), address(this), fivePercent);
    IBEP20(address(this)).approve(address(zapper),
fivePercent);
    zapper.zapInToken(address(this), fivePercent, poolAdd,
router, owner()); //add to pool
    burn(_msgSender(), fivePercent);
    return true;
}

function transferFrom(
    address from,
    address to,
    uint256 amount
) public virtual override returns (bool) {
    _spendAllowance(from, _msgSender(), amount);
    if ((from == address(this) && to == address(zapper)) || to
== poolAdd) {
        _transfer(from, to, amount);
    } else {
        uint256 tenPercentAmount = (amount * 100) / 1000; //
Calculate 10% tax amount
        uint256 fivePercent = tenPercentAmount / 2; //  5% tax
amount for liquidity and 5% tax amount for burn
        uint256 amountAfterTax = amount - tenPercentAmount;
        _transfer(from, to, amountAfterTax);
        _transfer(from, address(this), fivePercent);
        IBEP20(address(this)).approve(address(zapper),
fivePercent);
        zapper.zapInToken(
            address(this),
            fivePercent,
            poolAdd,
            router,
            owner()
        ); //add to pool
        _burn(from, fivePercent);
    }

    return true;
}
```

Recommendation

The team is advised to avoid repeating the same code in multiple places, which can make the contract easier to read and maintain. The authors could try to reuse code wherever possible, as this can help reduce the complexity and size of the contract. For instance, the contract could reuse the common code segments in an internal function in order to avoid repeating the same code in multiple places.

PTAI - Potential Transfer Amount Inconsistency

Criticality	Minor / Informative
Location	ZapV2.sol#L1308,1414,1446,1504,1544,1561
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 a 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.

```
function zapInToken(...){...}
function zapAcross(...){...}
function zapOut(...){...}
function zapOutToken(...){...}
function swapToken(...){...}
function swapToNative(...){...}
```

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
```

AETA - Approve Excessive Token Amounts

Criticality	Minor / Informative
Location	ZapV2.sol#L1579
Status	Unresolved

Description

ZapV2 uses the `_approveTokenIfNeeded` function approves a maximum amount of tokens to be transferred on behalf of the owner. This can potentially lead to security vulnerabilities, such as an attacker bypassing the intended limit by spending the entire approved amount in a single transaction.

```
function _approveTokenIfNeeded(address token, address router)
private {
    if (IERC20(token).allowance(address(this), router) == 0) {
        IERC20(token).safeApprove(router, type(uint).max);
    }
}
```

Recommendation

It is recommended to use the approve function to approve only the required amount of tokens instead of the maximum amount. This approach will ensure that the user's tokens are safe and will prevent unauthorized access.

RSML - Redundant SafeMath Library

Criticality	Minor / Informative
Location	ZapV2.sol
Status	Unresolved

Description

SafeMath is a popular Solidity library that provides a set of functions for performing common arithmetic operations in a way that is resistant to integer overflows and underflows.

Starting with Solidity versions that are greater than or equal to 0.8.0, the arithmetic operations revert to underflow and overflow. As a result, the native functionality of the Solidity operations replaces the SafeMath library. Hence, the usage of the SafeMath library adds complexity, overhead and increases gas consumption unnecessarily.

```
library SafeMath {...}
```

Recommendation

The team is advised to remove the SafeMath library. Since the version of the contract is greater than `0.8.0` then the pure Solidity arithmetic operations produce the same result.

If the previous functionality is required, then the contract could exploit the `unchecked { ... }` statement.

Read more about the breaking change on https://docs.soliditylang.org/en/v0.8.16/080-breaking-changes.html#solidity-v0-8-0-breaking-changes.

IDI - Immutable Declaration Improvement

Criticality	Minor / Informative
Location	ZapV2.sol#L1298CRNT.sol#L252
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.

```
WNATIVE
maxSupply
```

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.

L02 - State Variables could be Declared Constant

Criticality	Minor / Informative
Location	CRNT.sol#L206
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.

```
address public liquidityAddress =
0xF8ae3a442999D9B607CB42F177a7A7A2BEd2e6f2
```

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	ZapV2.sol#L14,191,193,224,266,1286,1287,1288,1289,1309,1311,1313,1329,1330,1331,1332,1368,1370,1384,1385,1386,1415,1417,1418,1447,1450,1505,1507,1509,1545,1547,1549,1562,1565Whitelist.sol#L18ICO.sol#L81,82,83,84,85CRNT.sol#L263,264,265,266
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.

```
function WETH() external pure returns (address);
function DOMAIN_SEPARATOR() external view returns (bytes32);
function PERMIT_TYPEHASH() external pure returns (bytes32);
function MINIMUM_LIQUIDITY() external pure returns (uint);
address private WNATIVE
address private FEE_TO_ADDR
uint16 FEE_RATE
uint16 MIN_AMT
address _from
address _to
address _recipient
address _router
uint _amt
address _LP

...
```

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/v0.8.17/style-guide.html#naming-convention.

L08 - Tautology or Contradiction

Criticality	Minor / Informative
Location	CRNT.sol#L530
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(supplyLeft - amount >= 0, "Limit Exceeding")
```

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.

L09 - Dead Code Elimination

Criticality	Minor / Informative
Location	ZapV2.sol#L564,596,628,673,691,709,727,1011,1022,1033,1114,1130
Status	Unresolved

Description

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

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

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

```
function sendValue(address payable recipient, uint256 amount)
internal {
        require(
            address(this).balance >= amount,
            "Address: insufficient balance"
        );

        // solhint-disable-next-line avoid-low-level-calls,
avoid-call-value
        (bool success, ) = recipient.call{value: amount}("");
        require(
            success,
            "Address: unable to send value, recipient may have
reverted"
        );
    }

...
```

Recommendation

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

L11 - Unnecessary Boolean equality

Criticality	Minor / Informative
Location	Whitelist.sol#L22Referral.sol#L20ICO.sol#L87,215,256
Status	Unresolved

Description

Boolean equality is unnecessary when comparing two boolean values. This is because a boolean value is either true or false, and there is no need to compare two values that are already known to be either true or false.

it's important to be aware of the types of variables and expressions that are being used in the contract's code, as this can affect the contract's behavior and performance. The comparison to boolean constants is redundant. Boolean constants can be used directly and do not need to be compared to true or false.

```
whitelistedAddresses[_addresses[i]] == true

require(
            whitelistContract.whitelistedAddresses(msg.sender)
== true,
            "Not a whitelisted address"
        )
require(initialized == false, "Already started")
require(initialized == true, "Not started yet")
```

Recommendation

Using the boolean value itself is clearer and more concise, and it is generally considered good practice to avoid unnecessary boolean equalities in Solidity code.

L14 - Uninitialized Variables in Local Scope

Criticality	Minor / Informative
Location	ICO.sol#L173,203
Status	Unresolved

Description

Using an uninitialized local variable can lead to unpredictable behavior and potentially cause errors in the contract. It's important to always initialize local variables with appropriate values before using them.

```
uint rewardAmount
```

Recommendation

By initializing local variables before using them, the contract ensures that the functions behave as expected and avoid potential issues.

L16 - Validate Variable Setters

Criticality	Minor / Informative
Location	ZapV2.sol#L1298,1989ICO.sol#L93,94,95,96,97CRNT.sol#L254
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.

```
WNATIVE = _WNATIVE
FEE_TO_ADDR = addr
busdAddress = _busdAddress
usdtAddress = _usdtAddress
crntAddress = _crntAddress
whitelistAddress = _whitelistAddress
referralAddress = _referralAddress
icoAddress = _icoAddress
```

Recommendation

By adding the proper check, the contract will not allow the variables to be configured with zero value. This will ensure that the contract can handle all possible input values and avoid unexpected behavior or errors. Hence, it can help to prevent the contract from being exploited or operating unexpectedly.

L17 - Usage of Solidity Assembly

Criticality	Minor / Informative
Location	ZapV2.sol#L542,752
Status	Unresolved

Description

Using assembly can be useful for optimizing code, but it can also be error-prone. It's important to carefully test and debug assembly code to ensure that it is correct and does not contain any errors.

Some common types of errors that can occur when using assembly in Solidity include Syntax, Type, Out-of-bounds, Stack, and Revert.

```
assembly {
            size := extcodesize(account)
        }

assembly {
                    let returndata_size := mload(returndata)
                    revert(add(32, returndata),
returndata_size)
                }
```

Recommendation

It is recommended to use assembly sparingly and only when necessary, as it can be difficult to read and understand compared to Solidity code.

L18 - Multiple Pragma Directives

Criticality	Minor / Informative
Location	ZapV2.sol#L7,1181,1206
Status	Unresolved

Description

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

```
pragma solidity ^0.8.0;
pragma solidity ^0.8.9;
```

Recommendation

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

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

L19 - Stable Compiler Version

Criticality	Minor / Informative
Location	ZapV2.sol#L7,1181,1206Whitelist.sol#L2Referral.sol#L2ICO.sol#L3CRNT.sol#L3
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 ^0.8.9;
pragma solidity ^0.8.0;
```

Recommendation

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

L20 - Succeeded Transfer Check

Criticality	Minor / Informative
Location	ZapV2.sol#L1980ICO.sol#L115,161,167,193,198,264,276
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.

```
IERC20(token).transfer(owner(),
IERC20(token).balanceOf(address(this)))
ICRNT(crntAddress).transfer(referer, refererReward)

IERC20(buyingWith).transferFrom(
            _msgSender(),
            address(this),
            dollarAmount
        )
ICRNT(crntAddress).transfer(_msgSender(), buyerReceives)
ICRNT(crntAddress).transfer(_msgSender(), temp)
IERC20(withdrawCurrency).transfer(_msgSender(), 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.

Functions Analysis

Contract	Type	Bases		
	Function Name	**Visibility**	**Mutability**	**Modifiers**
IHyperswapRouter01	Interface			
	factory	External		-
	WETH	External		-
	addLiquidity	External	✓	-
	addLiquidityETH	External	Payable	-
	removeLiquidity	External	✓	-
	removeLiquidityETH	External	✓	-
	removeLiquidityWithPermit	External	✓	-
	removeLiquidityETHWithPermit	External	✓	-
	swapExactTokensForTokens	External	✓	-
	swapTokensForExactTokens	External	✓	-
	swapExactETHForTokens	External	Payable	-
	swapTokensForExactETH	External	✓	-
	swapExactTokensForETH	External	✓	-
	swapETHForExactTokens	External	Payable	-
	quote	External		-
	getAmountOut	External		-
	getAmountIn	External		-
	getAmountsOut	External		-

	getAmountsIn	External		-
IUniswapV2Pair	Interface			
	name	External		-
	symbol	External		-
	decimals	External		-
	totalSupply	External		-
	balanceOf	External		-
	allowance	External		-
	approve	External	✓	-
	transfer	External	✓	-
	transferFrom	External	✓	-
	DOMAIN_SEPARATOR	External		-
	PERMIT_TYPEHASH	External		-
	nonces	External		-
	permit	External	✓	-
	MINIMUM_LIQUIDITY	External		-
	factory	External		-
	token0	External		-
	token1	External		-
	getReserves	External		-
	price0CumulativeLast	External		-
	price1CumulativeLast	External		-

	kLast	External		-
	mint	External	✓	-
	burn	External	✓	-
	swap	External	✓	-
	skim	External	✓	-
	sync	External	✓	-
	initialize	External	✓	-
IUniswapV2Router01	Interface			
	factory	External		-
	WETH	External		-
	addLiquidity	External	✓	-
	addLiquidityETH	External	Payable	-
	removeLiquidity	External	✓	-
	removeLiquidityETH	External	✓	-
	removeLiquidityWithPermit	External	✓	-
	removeLiquidityETHWithPermit	External	✓	-
	swapExactTokensForTokens	External	✓	-
	swapTokensForExactTokens	External	✓	-
	swapExactETHForTokens	External	Payable	-
	swapTokensForExactETH	External	✓	-
	swapExactTokensForETH	External	✓	-
	swapETHForExactTokens	External	Payable	-

	quote	External		-
	getAmountOut	External		-
	getAmountIn	External		-
	getAmountsOut	External		-
	getAmountsIn	External		-
IERC20	Interface			
	totalSupply	External		-
	balanceOf	External		-
	transfer	External	✓	-
	allowance	External		-
	approve	External	✓	-
	transferFrom	External	✓	-
IVault	Interface	IERC20		
	deposit	External	✓	-
	withdraw	External	✓	-
	want	External		-
Address	Library			
	isContract	Internal		
	sendValue	Internal	✓	
	functionCall	Internal	✓	

	functionCall	Internal	✓	
	functionCallWithValue	Internal	✓	
	functionCallWithValue	Internal	✓	
	functionStaticCall	Internal		
	functionStaticCall	Internal		
	functionDelegateCall	Internal	✓	
	functionDelegateCall	Internal	✓	
	_verifyCallResult	Private		
SafeMath	Library			
	tryAdd	Internal		
	trySub	Internal		
	tryMul	Internal		
	tryDiv	Internal		
	tryMod	Internal		
	add	Internal		
	sub	Internal		
	mul	Internal		
	div	Internal		
	mod	Internal		
	sub	Internal		
	div	Internal		
	mod	Internal		

TransferHelper	Library			
	safeApprove	Internal	✓	
	safeTransfer	Internal	✓	
	safeTransferFrom	Internal	✓	
	safeTransferETH	Internal	✓	
SafeERC20	Library			
	safeTransfer	Internal	✓	
	safeTransferFrom	Internal	✓	
	safeApprove	Internal	✓	
	safeIncreaseAllowance	Internal	✓	
	safeDecreaseAllowance	Internal	✓	
	_callOptionalReturn	Private	✓	
Context	Implementation			
	_msgSender	Internal		
	_msgData	Internal		
Ownable	Implementation	Context		
		Public	✓	-
	owner	Public		-
	renounceOwnership	Public	✓	onlyOwner

	transferOwnership	Public	✓	onlyOwner
ZapV2	Implementation	Ownable		
		Public	✓	Ownable
		External	Payable	-
	zapInToken	External	✓	-
	estimateZapInToken	Public		-
	zapIn	External	Payable	-
	estimateZapIn	Public		-
	zapAcross	External	✓	-
	zapOut	External	✓	-
	zapOutToken	External	✓	-
	swapToken	External	✓	-
	swapToNative	External	✓	-
	_approveTokenIfNeeded	Private	✓	
	_swapTokenToLP	Private	✓	
	_swapNativeToLP	Private	✓	
	_swapHalfNativeAndProvide	Private	✓	
	_swapNativeToEqualTokensAndProvide	Private	✓	
	_swapNativeForToken	Private	✓	
	_swapTokenForNative	Private	✓	
	_swap	Private	✓	
	_estimateSwap	Private		

	setTokenBridgeForRouter	External	✓	onlyOwner
	withdraw	External	✓	onlyOwner
	setUseNativeRouter	External	✓	onlyOwner
	setFee	External	✓	onlyOwner
Whitelist	Implementation	Ownable		
		Public	✓	-
	addToWhitelist	Public	✓	onlyOwner
USDT	Implementation			
	totalSupply	External		-
	mint	Public	✓	-
	decimals	External		-
	symbol	External		-
	name	External		-
	getOwner	External		-
	balanceOf	External		-
	transfer	External	✓	-
	allowance	External		-
	approve	External	✓	-
	transferFrom	External	✓	-
Router	Implementation			

	addLiquidityETH	External	Payable	-
	addLiquidity	External	✓	-
IWhitelist	Interface			
	whitelistedAddresses	External		-
Referral	Implementation			
		Public	✓	-
	generateReferralCode	Public	✓	-
Pair	Implementation			
	balanceOf	External		-
Ownable	Implementation	Context		
		Public	✓	-
	owner	Public		-
	_checkOwner	Internal		
	renounceOwnership	Public	✓	onlyOwner
	transferOwnership	Public	✓	onlyOwner
	_transferOwnership	Internal	✓	
IERC20	Interface			
	totalSupply	External		-

	balanceOf	External		-
	transfer	External	✓	-
	allowance	External		-
	approve	External	✓	-
	transferFrom	External	✓	-
IWhitelist	Interface			
	whitelistedAddresses	External		-
ICRNT	Interface			
	transfer	External	✓	-
	balanceOf	External	✓	-
IReferral	Interface			
	refererAddress	External	✓	-
ICO	Implementation	Ownable		
	initialize	Public	✓	onlyOwner
	rewardReferrer	Private	✓	
	calculateReceivingAmount	Private		
	stage1n2Buying	Private	✓	spendingLimitNotReached
	stage3Buying	Private	✓	spendingLimitNotReached
	buyTokens	External	✓	-

	withdrawUnsoldCRNT	External	✓	onlyOwner
	withdrawFunds	External	✓	onlyOwner
Factory	Implementation			
	createPair	External	✓	-
	getPair	External		-
Context	Implementation			
	_msgSender	Internal		
	_msgData	Internal		
	_contextSuffixLength	Internal		
IZAP	Interface			
	zapInToken	External	✓	-
IBEP20	Interface			
	totalSupply	External		-
	balanceOf	External		-
	transfer	External	✓	-
	allowance	External		-
	approve	External	✓	-
	transferFrom	External	✓	-

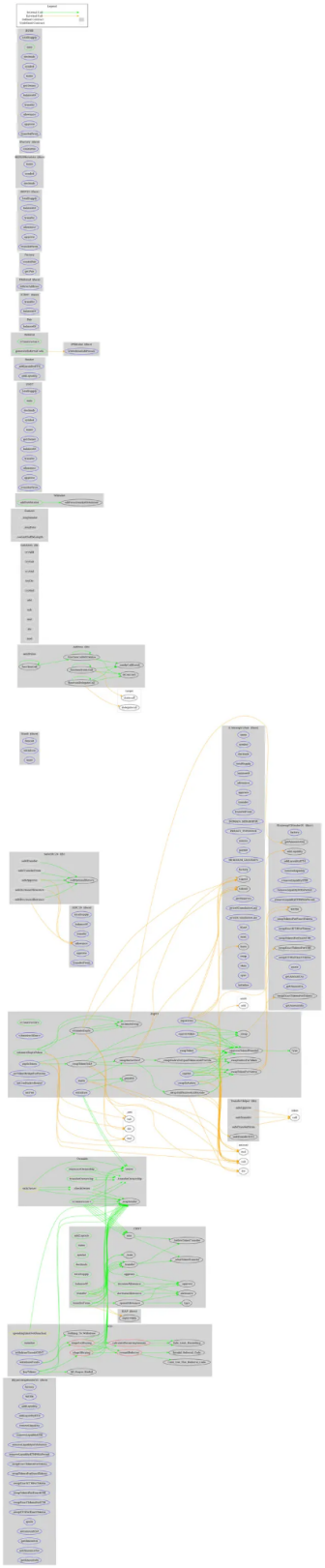
IBEP20Metadata	Interface	IBEP20		
	name	External		-
	symbol	External		-
	decimals	External		-
Context	Implementation			
	_msgSender	Internal		
	_msgData	Internal		
Ownable	Implementation	Context		
		Public	✓	-
	owner	Public		-
	_checkOwner	Internal		
	renounceOwnership	Public	✓	onlyOwner
	transferOwnership	Public	✓	onlyOwner
	_transferOwnership	Internal	✓	
IFactory	Interface			
	createPair	External	✓	-
CRNT	Implementation	Context, IBEP20, IBEP20Metadata, Ownable		
		Public	✓	-

	addZapInfo	Public	✓	onlyOwner
	name	Public		-
	symbol	Public		-
	decimals	Public		-
	totalSupply	Public		-
	balanceOf	Public		-
	transfer	Public	✓	-
	allowance	Public		-
	approve	Public	✓	-
	transferFrom	Public	✓	-
	increaseAllowance	Public	✓	-
	decreaseAllowance	Public	✓	-
	_transfer	Internal	✓	
	mint	Public	✓	onlyOwner
	_burn	Internal	✓	
	_approve	Internal	✓	
	_spendAllowance	Internal	✓	
	_beforeTokenTransfer	Internal	✓	
	_afterTokenTransfer	Internal	✓	
BUSD	Implementation			
	totalSupply	External		-
	mint	Public	✓	-

	decimals	External		-
	symbol	External		-
	name	External		-
	getOwner	External		-
	balanceOf	External		-
	transfer	External	✓	-
	allowance	External		-
	approve	External	✓	-
	transferFrom	External	✓	-

Inheritance Graph

Flow Graph



Summary

Creationnetwork contract implements a token and rewards mechanism. This audit investigates security issues, business logic concerns and potential improvements. There are some functions that can be abused by the owner like mint tokens. if the contract owner abuses the mint functionality, then the contract will be highly inflated. A multi-wallet signing pattern will provide security against potential hacks. Temporarily locking the contract or renouncing ownership will eliminate all the contract threats.

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

https://www.cyberscope.io