Cyberscope

Audit Report

ZangAi

June 2023

Network	BSC
Address	0x734085e65B2F299a917a01f3EE09931e1d10553f
Audited by	© cyberscope

Analysis

● Critical ● Medium ● Minor / Informative ● Pass

Severity	Code	Description	Status
●	ST	Stops Transactions	Unresolved
●	OTUT	Transfers User's Tokens	Passed
●	ELFM	Exceeds Fees Limit	Passed
●	MT	Mints Tokens	Passed
●	BT	Burns Tokens	Passed
●	BC	Blacklists Addresses	Passed

Diagnostics

🔴 Critical 🟠 Medium ⚪ Minor / Informative

Severity	Code	Description	Status
⚪	RLF	Redundant Liquidity Feature	Unresolved
⚪	RSD	Redundant Swap Duplication	Unresolved
⚪	DKO	Delete Keyword Optimization	Unresolved
⚪	PVC	Price Volatility Concern	Unresolved
⚪	PTRP	Potential Transfer Revert Propagation	Unresolved
⚪	FSA	Fixed Swap Address	Unresolved
⚪	MEM	Misleading Error Messages	Unresolved
⚪	IDI	Immutable Declaration Improvement	Unresolved
⚪	L02	State Variables could be Declared Constant	Unresolved
⚪	L04	Conformance to Solidity Naming Conventions	Unresolved
⚪	L09	Dead Code Elimination	Unresolved
⚪	L11	Unnecessary Boolean equality	Unresolved
⚪	L14	Uninitialized Variables in Local Scope	Unresolved

●	L17	Usage of Solidity Assembly	Unresolved
●	L20	Succeeded Transfer Check	Unresolved

Table of Contents

Review

Contract Name	ZangAi
Compiler Version	v0.8.17+commit.8df45f5f
Optimization	200 runs
Explorer	https://bscscan.com/address/0x734085e65b2f299a917a01f3ee09931e1d10553f
Address	0x734085e65b2f299a917a01f3ee09931e1d10553f
Network	BSC
Symbol	ZangAi
Decimals	9
Total Supply	420,690,000,000,000

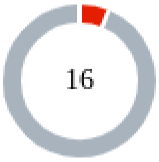
Audit Updates

Initial Audit	20 Jun 2023

Source Files

Filename	SHA256
ZangAi.sol	bd540540cac48950973c3b7d4689aa219eee715c17a28e03e78157fb782c6303

Findings Breakdown



●	Critical	1
●	Medium	0
●	Minor / Informative	15

Severity		Unresolved	Acknowledged	Resolved	Other
●	Critical	1	0	0	0
●	Medium	0	0	0	0
●	Minor / Informative	15	0	0	0

ST - Stops Transactions

Criticality	Critical
Location	ZangAi.sol#L694
Status	Unresolved

Description

The transactions are initially disabled for all users excluding the authorized addresses. The owner can enable the transactions for all users. Once the transactions are enable the owner will not be able to disable them again.

```
if(!_isExcludedFromFees[from] && !_isExcludedFromFees[to]) {
  require(tradingEnabled, "Trading is not enabled yet");
}
```

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. Some suggestions are:

- Introduce a multi-sign wallet so that many addresses will confirm the action.
- Introduce a governance model where users will vote about the actions.

RLF - Redundant Liquidity Feature

Criticality	Minor / Informative
Location	ZangAi.sol#L709
Status	Unresolved

Description

The smart contract code includes a liquidity mechanism that is intended to take a portion of each transaction, represented by `liquidityFeeonBuy` and `liquidityFeeonSell` , and add it to a liquidity pool. The `liquidityShare` is calculated as the sum of `liquidityFeeonBuy` and `liquidityFeeonSell` , and if `liquidityShare` is greater than 0, a corresponding number of tokens are converted into liquidity tokens via the `swapAndLiquify` function.

However, both the `liquidityFeeonBuy` and `liquidityFeeonSell` are initialized to 0 in the constructor and are never updated in the contract. This means that `liquidityShare` will always be 0, and the condition `liquidityShare > 0` will never be true. As a result, the `swapAndLiquify` function will never be called, and the liquidity mechanism will never be activated.

This can lead to confusion for users and it also unnecessarily increases the complexity and gas costs of the contract.

```
uint256 marketingShare = marketingFeeonBuy +
marketingFeeonSell;
uint256 liquidityShare = liquidityFeeonBuy +
liquidityFeeonSell;
uint256 totalShare = marketingShare + liquidityShare;
if(totalShare > 0) {
    if(liquidityShare > 0) {
        uint256 liquidityTokens = (contractTokenBalance *
liquidityShare) / totalShare;
        swapAndLiquify(liquidityTokens);
    }
```

Recommendation

The contract should be updated to either make use of the liquidity mechanism or remove the redundant code.

1. If the intention is to use the liquidity mechanism, the contract should be updated to allow `liquidityFeeonBuy` and `liquidityFeeonSell` to be set to non-zero values. This could be done through a function that can be called by the contract owner or through a mechanism that adjusts the fees based on certain conditions.
2. If the liquidity mechanism is not needed, the code related to it, including the calculation of `liquidityShare` and the call to `swapAndLiquify`, should be removed from the contract. This will simplify the contract and reduce gas costs.

RSD - Redundant Swap Duplication

Criticality	Minor / Informative
Location	ZangAi.sol#L712
Status	Unresolved

Description

The contract contains multiple swap methods that individually perform token swaps and transfer promotional amounts to specific addresses and features. This redundant duplication of code introduces unnecessary complexity and increases dramatically the gas consumption. By consolidating these operations into a single swap method, the contract can achieve better code readability, reduce gas costs, and improve overall efficiency.

```
if(liquidityShare > 0) {
    uint256 liquidityTokens = (contractTokenBalance *
liquidityShare) / totalShare;
    swapAndLiquify(liquidityTokens);
}

if(marketingShare > 0) {
    uint256 marketingTokens = (contractTokenBalance *
marketingShare) / totalShare;
    swapAndSendMarketing(marketingTokens);
}
```

Recommendation

A more optimized approach could be adopted to perform the token swap operation once for the total amount of tokens and distribute the proportional amounts to the corresponding addresses, eliminating the need for separate swaps.

DKO - Delete Keyword Optimization

Criticality	Minor / Informative
Location	ZangAi.sol#L645
Status	Unresolved

Description

The contract resets variables to the default state by setting the initial values. Setting values to state variables increases the gas cost.

```
function removeAllFee() private {
    if(_taxFee == 0 && _liquidityFee == 0 && _marketingFee ==
0) return;

    _taxFee = 0;
    _marketingFee = 0;
    _liquidityFee = 0;
}
```

Recommendation

The team is advised to use the `delete` keyword instead of setting variables. This can be more efficient than setting the variable to a new value, using delete can reduce the gas cost associated with storing data on the blockchain.

PVC - Price Volatility Concern

Criticality	Minor / Informative
Location	ZangAi.sol#L781
Status	Unresolved

Description

The contract accumulates tokens from the taxes to swap them for ETH. The variable `swapTokensAtAmount` sets a threshold where the contract will trigger the swap functionality. If the variable is set to a big number, then the contract will swap a huge amount of tokens for ETH.

It is important to note that the price of the token representing it, can be highly volatile. This means that the value of a price volatility swap involving Ether could fluctuate significantly at the triggered point, potentially leading to significant price volatility for the parties involved.

```
    function setSwapTokensAtAmount(uint256 newAmount) external
onlyOwner() {
        require(newAmount > totalSupply() / 1e5,
"SwapTokensAtAmount must be greater than 0.001% of total
supply");
        swapTokensAtAmount = newAmount;
        emit SwapTokensAtAmountUpdated(newAmount);
    }
```

Recommendation

The contract could ensure that it will not sell more than a reasonable amount of tokens in a single transaction. A suggested implementation could check that the maximum amount should be less than a fixed percentage of the total supply. Hence, the contract will guarantee that it cannot accumulate a huge amount of tokens in order to sell them.

PTRP - Potential Transfer Revert Propagation

Criticality	Minor / Informative
Location	ZangAi.sol#L776
Status	Unresolved

Description

The contract sends funds to a `marketingWallet` as part of the transfer flow. This address can either be a wallet address or a contract. If the address belongs to a contract then it may revert from incoming payment. As a result, the error will propagate to the token’s contract and revert the transfer.

```
payable(marketingWallet).sendValue(newBalance);
```

Recommendation

The contract should tolerate the potential revert from the underlying contracts when the interaction is part of the main transfer flow. This could be achieved by not allowing set contract addresses or by sending the funds in a non-revertable way.

FSA - Fixed Swap Address

Criticality	Minor / Informative
Location	ZangAi.sol#L409
Status	Unresolved

Description

The swap address is assigned once and it can not be changed. It is a common practice in decentralized exchanges to create new swap versions. A contract that cannot change the swap address may not be able to catch up to the upgrade. As a result, the contract will not be able to migrate to a new liquidity pool pair or decentralized exchange.

```
constructor ()  {
    ...
    IUniswapV2Router02 _uniswapV2Router =
IUniswapV2Router02(router);
    uniswapV2Pair =
IUniswapV2Factory(_uniswapV2Router.factory())
    .createPair(address(this), _uniswapV2Router.WETH());
    uniswapV2Router = _uniswapV2Router;
    ...
}
```

Recommendation

The team is advised to add the ability to change the pair and router address in order to cover potential liquidity pool migrations. It would be better to support multiple pair addresses so the token will be able to have the same behavior in all the decentralized liquidity pairs.

MEM - Misleading Error Messages

Criticality	Minor / Informative
Location	ZangAi.sol#L558
Status	Unresolved

Description

The contract is using misleading error messages. These error messages do not accurately reflect the problem, making it difficult to identify and fix the issue.

The use of the term "native tokens" in the error message is misleading. The intent behind the message is to indicate that the owner of the contract should not be able to claim tokens associated with the contract itself, rather than referring to tokens native to the blockchain.

```
require(token != address(this), "Owner cannot claim native
tokens");
```

Recommendation

The team is suggested to modify the error message and remove the term `native` . This modification will help users understand that the prohibition pertains to tokens specifically linked to the contract, rather than implying a restriction on native tokens.

IDI - Immutable Declaration Improvement

Criticality	Minor / Informative
Location	ZangAi.sol#L410,412,416,417,419,420,422,423,425,426
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.

```
uniswapV2Pair
uniswapV2Router
taxFeeonBuy
taxFeeonSell
liquidityFeeonBuy
liquidityFeeonSell
marketingFeeonBuy
marketingFeeonSell
totalBuyFees
totalSellFees
```

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	ZangAi.sol#L349,350,351,376
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.

```
string private _name     = "ZangAi";
string private _symbol   = "ZangAi";
uint8  private _decimals = 9;

address private DEAD =
0x000000000000000000000000000000000000dEaD;
```

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	ZangAi.sol#L168,169,185,204,367,368,369,376,633,637,641,787,870
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 DOMAIN_SEPARATOR() external view returns (bytes32);
function PERMIT_TYPEHASH() external pure returns (bytes32);
function MINIMUM_LIQUIDITY() external pure returns (uint);
function WETH() external pure returns (address);     uint256
public _taxFee;
uint256 public _liquidityFee;
uint256 public _marketingFee;
address private DEAD =
0x000000000000000000000000000000000000dEaD;
_amount
_enabled
_marketingWallet

...
```

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.

L09 - Dead Code Elimination

Criticality	Minor / Informative
Location	ZangAi.sol#L77,96,100,104,108,113
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 isContract(address account) internal view returns
(bool) {
  // According to EIP-1052, 0x0 is the value returned for
not-yet created accounts
  // and
0xc5d2460186f7233c927e7db2dcc703c0e500b653ca82273b7bfad8045d85a
470 is returned
  // for accounts without code, i.e. `keccak256('')`
  bytes32 codehash;
  ...
}

function functionCall(address target, bytes memory data)
internal returns (bytes memory) {
  return functionCall(target, data, "Address: low-level call
failed");
}

function functionCall(address target, bytes memory data, string
memory errorMessage) internal returns (bytes memory) {
  return _functionCallWithValue(target, data, 0, errorMessage);
}

function functionCallWithValue(address target, bytes memory
data, uint256 value) internal returns (bytes memory) {
  return functionCallWithValue(target, data, value, "Address:
low-level call with value failed");
}

function functionCallWithValue(address target, bytes memory
data, uint256 value, string memory errorMessage) internal
returns (bytes memory) {
  require(address(this).balance >= value, "Address:
insufficient balance for call");
...
}

function _functionCallWithValue(address target, bytes memory
data, uint256 weiValue, string memory errorMessage) private
returns (bytes memory) {
  require(isContract(target), "Address: call to non-contract");

...
}
```

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	ZangAi.sol#L682
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.

```
require(tradingEnabled == false, "Trading is already enabled");
```

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	ZangAi.sol#L398
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.

```
address router;
```

Recommendation

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

L17 - Usage of Solidity Assembly

Criticality	Minor / Informative
Location	ZangAi.sol#L84,126
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 { codehash := extcodehash(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.

L20 - Succeeded Transfer Check

Criticality	Minor / Informative
Location	ZangAi.sol#L565
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.

```
ERC20token.transfer(msg.sender, 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**
Context	Implementation			
	_msgSender	Internal		
	_msgData	Internal		
Ownable	Implementation	Context		
		Public	✓	-
	owner	Public		-
	renounceOwnership	Public	✓	onlyOwner
	transferOwnership	Public	✓	onlyOwner
IERC20	Interface			
	totalSupply	External		-
	balanceOf	External		-
	transfer	External	✓	-
	allowance	External		-
	approve	External	✓	-
	transferFrom	External	✓	-

Address	Library			
	isContract	Internal		
	sendValue	Internal	✓	
	functionCall	Internal	✓	
	functionCall	Internal	✓	
	functionCallWithValue	Internal	✓	
	functionCallWithValue	Internal	✓	
	_functionCallWithValue	Private	✓	
IUniswapV2Factory	Interface			
	feeTo	External		-
	feeToSetter	External		-
	getPair	External		-
	allPairs	External		-
	allPairsLength	External		-
	createPair	External	✓	-
	setFeeTo	External	✓	-
	setFeeToSetter	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		-
	burn	External	✓	-
	swap	External	✓	-
	skim	External	✓	-
	sync	External	✓	-
	initialize	External	✓	-

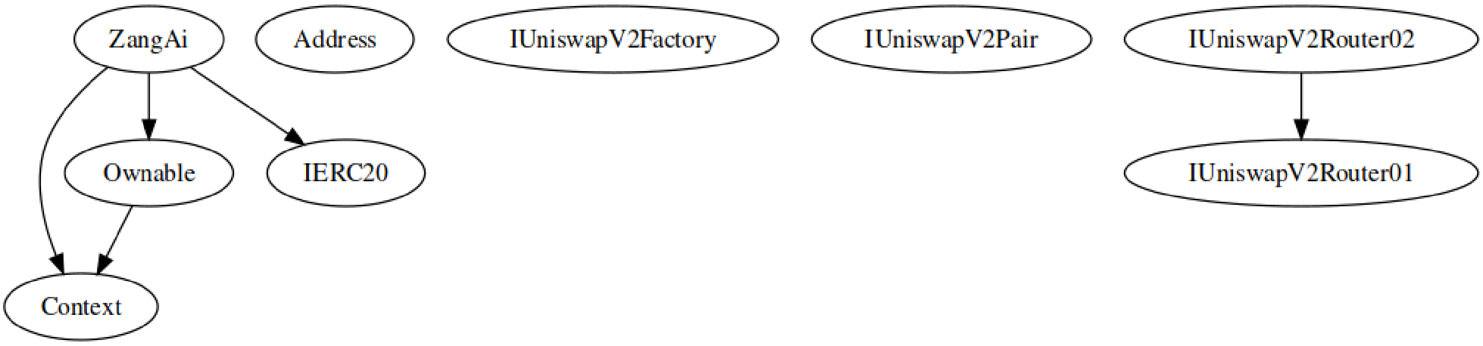
IUniswapV2Router01	Interface			
	factory	External		-
	WETH	External		-
	addLiquidity	External	✓	-
	addLiquidityETH	External	Payable	-
	removeLiquidity	External	✓	-
	removeLiquidityETH	External	✓	-
	removeLiquidityWithPermit	External	✓	-
	removeLiquidityETHWithPermit	External	✓	-
	swapExactTokensForTokens	External	✓	-
	swapTokensForExactTokens	External	✓	-
	swapExactETHForTokens	External	Payable	-
	swapTokensForExactETH	External	✓	-
	swapExactTokensForETH	External	✓	-
	swapETHForExactTokens	External	Payable	-
	quote	External		-
	getAmountOut	External		-
	getAmountIn	External		-
	getAmountsOut	External		-
	getAmountsIn	External		-
IUniswapV2Router02	Interface	IUniswapV2Router01		

	removeLiquidityETHSupportingFeeOnTransferTokens	External	✓	-
	removeLiquidityETHWithPermitSupportingFeeOnTransferTokens	External	✓	-
	swapExactTokensForTokensSupportingFeeOnTransferTokens	External	✓	-
	swapExactETHForTokensSupportingFeeOnTransferTokens	External	Payable	-
	swapExactTokensForETHSupportingFeeOnTransferTokens	External	✓	-
ZangAi	Implementation	Context, IERC20, Ownable		
		Public	✓	-
	name	Public		-
	symbol	Public		-
	decimals	Public		-
	totalSupply	Public		-
	balanceOf	Public		-
	transfer	Public	✓	-
	allowance	Public		-
	approve	Public	✓	-
	transferFrom	Public	✓	-
	increaseAllowance	Public	✓	-
	decreaseAllowance	Public	✓	-
	isExcludedFromReward	Public		-
	totalReflectionDistributed	Public		-
	deliver	Public	✓	-

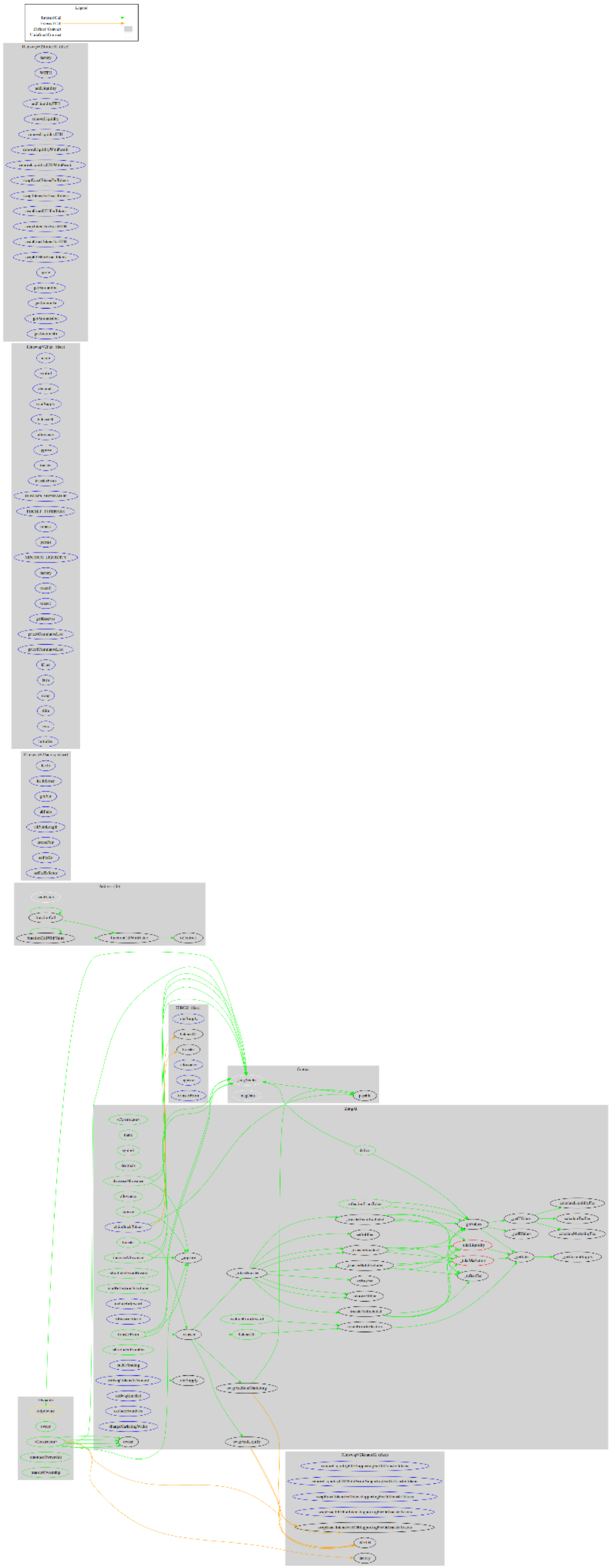
	reflectionFromToken	Public		-
	tokenFromReflection	Public		-
	excludeFromReward	Public	✓	onlyOwner
	includeInReward	External	✓	onlyOwner
		External	Payable	-
	claimStuckTokens	External	✓	onlyOwner
	_reflectFee	Private	✓	
	_getValues	Private		
	_getTValues	Private		
	_getRValues	Private		
	_getRate	Private		
	_getCurrentSupply	Private		
	_takeLiquidity	Private	✓	
	_takeMarketing	Private	✓	
	calculateTaxFee	Private		
	calculateLiquidityFee	Private		
	calculateMarketingFee	Private		
	removeAllFee	Private	✓	
	setBuyFee	Private	✓	
	setSellFee	Private	✓	
	isExcludedFromFee	Public		-
	_approve	Private	✓	
	enableTrading	External	✓	onlyOwner

	_transfer	Private	✓	
	swapAndLiquify	Private	✓	
	swapAndSendMarketing	Private	✓	
	setSwapTokensAtAmount	External	✓	onlyOwner
	setSwapEnabled	External	✓	onlyOwner
	_tokenTransfer	Private	✓	
	_transferStandard	Private	✓	
	_transferToExcluded	Private	✓	
	_transferFromExcluded	Private	✓	
	_transferBothExcluded	Private	✓	
	excludeFromFees	External	✓	onlyOwner
	changeMarketingWallet	External	✓	onlyOwner

Inheritance Graph



Flow Graph


Legend
Internal Call
External Call
Defined Contract
Undefined Contract
IUniswapV2Router01Router
factory
WETH
addLiquidity
addLiquidityETH
removeLiquidity
removeLiquidityETH
removeLiquidityWithPermit
removeLiquidityETHWithPermit
swapExactTokensForTokens
swapTokensForExactTokens
swapExactETHForTokens
swapTokensForExactETH
swapExactTokensForETH
swapETHForExactTokens
quote
getAmountOut
getAmountIn
getAmountsOut
getAmountsIn
IUniswapV2Pair
ZangAi

Summary

ZangAi contract implements a token mechanism. This audit investigates security issues, business logic concerns and potential improvements. There are some functions that can be abused by the owner like stop transactions. A multi-wallet signing pattern will provide security against potential hacks. A fee of 3% is applied on both buy and sell transactions.

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