Cyberscope

A TAC Security Company

Audit Report

Laika

August 2025

Network BSC

Address 0xaf48b7f9cE374B5d23fa18cf185DAB7DCd99961A

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	Unresolved
●	MT	Mints Tokens	Unresolved
●	BT	Burns Tokens	Unresolved
●	BC	Blacklists Addresses	Unresolved

Diagnostics

Critical Medium Minor / Informative

Severity	Code	Description	Status
●	UPM	Unrestricted Pair Modification	Unresolved
●	UTPD	Unverified Third Party Dependencies	Unresolved
●	CO	Code Optimization	Unresolved
●	CCR	Contract Centralization Risk	Unresolved
●	MEM	Missing Error Messages	Unresolved
●	MEE	Missing Events Emission	Unresolved
●	PLPI	Potential Liquidity Provision Inadequacy	Unresolved
●	PSU	Potential Subtraction Underflow	Unresolved
●	RCS	Redundant Conditional Statements	Unresolved
●	RSML	Redundant SafeMath Library	Unresolved
●	RSD	Redundant Swap Duplication	Unresolved
●	UVS	Unsanitized Variable Setters	Unresolved
●	ZD	Zero Division	Unresolved
●	L02	State Variables could be Declared Constant	Unresolved

●	L04	Conformance to Solidity Naming Conventions	Unresolved
●	L05	Unused State Variable	Unresolved
●	L07	Missing Events Arithmetic	Unresolved
●	L09	Dead Code Elimination	Unresolved
●	L13	Divide before Multiply Operation	Unresolved
●	L14	Uninitialized Variables in Local Scope	Unresolved
●	L16	Validate Variable Setters	Unresolved
●	L19	Stable Compiler Version	Unresolved
●	L22	Potential Locked Ether	Unresolved

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

Contract Name	LaiKa
Compiler Version	v0.8.24+commit.e11b9ed9
Optimization	200 runs
Explorer	https://bscscan.com/address/0xaf48b7f9ce374b5d23fa18cf185dab7dcd99961a
Address	0xaf48b7f9ce374b5d23fa18cf185dab7dcd99961a
Network	BSC
Symbol	TestB
Decimals	18
Badge Eligibility	Must Fix Criticals

Audit Updates

Initial Audit	15 Aug 2025

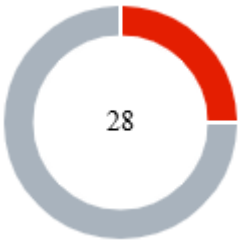
Source Files

Filename	SHA256
LaiKa.sol	1a1b0a162f17475d0996860b1d5452bdcbbc3428e8e1ba1c503ed6c5c9dbf4a1

Readability Comment

The audit scope is to identify security vulnerabilities, validate the business logic, and recommend potential optimizations. The codebase is incomplete, with convoluted functionalities and inconsistent logic. As such, the project cannot be considered production-ready. Furthermore, the contract does not adhere to core Solidity principles related to gas efficiency, code readability, and appropriate use of data structures. The development team is strongly advised to re-evaluate the business logic and align the implementation with established Solidity best practices to ensure both security and maintainability. Even if the identified issues are addressed and rectified, the contract would remain far from production-ready due to its convoluted and incomplete nature.

Findings Breakdown



Critical	7
Medium	0
Minor / Informative	21

Severity	Unresolved	Acknowledged	Resolved	Other
Critical	7	0	0	0
Medium	0	0	0	0
Minor / Informative	21	0	0	0

ST - Stops Transactions

Criticality	Critical
Location	LaiKa.sol#L1008
Status	Unresolved

Description

The contract contains several scenarios in which the `_transfer` function will revert for all users. Some of these are described with the `ZD` , `PSU` , `PLPI` , `ELFM` , `UVS` , `UPM` and `UTPD` findings. Additionally, admin functions allow the owner to stop transactions for all users excluding the owner. This is possible through the use of `setLaunch` and `setwhitelaunch` methods. As a result, the contract may operate as a honeypot.

```
function _transfer(
    address from,
    address to,
    uint256 amount
) internal override {
  ...
  require(launch, 'unlaunch');
  require(whitelaunch);
  ...
}
```

Recommendation

The team should follow the recommendations of the findings mentioned above. Additionally, 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.

ELFM - Exceeds Fees Limit

Criticality	Critical
Location	LaiKa.sol#L1207
Status	Unresolved

Description

The contract owner has the authority to increase fees over the allowed limit of 25%. The owner may take advantage of it by calling the `setBuyFee` function with a high percentage value. In particular, the owner is able to increase the fees beyond 100%.

```
function setBuyFee(uint256 _burnFee,uint256 _fundFee) external
onlyOwner {
    burnFee = _burnFee;
    fundFee = _fundFee;
}
```

Recommendation

The contract could embody a check for the maximum acceptable value. 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.

MT - Mints Tokens

Criticality	Critical
Location	LaiKa.sol#L928
Status	Unresolved

Description

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

```
function mint(address _address,uint256 _amount)public  returns
(bool) {
    require(totalSupply()+ _amount <= alltotalsupply);
    require(msg.sender == MintAddress);
    _mint(_address, _amount);
    return true;
}
```

Recommendation

The team should carefully manage the private keys of the owner's and `MintAddress` 's accounts. 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.

BT - Burns Tokens

Criticality	Critical
Location	LaiKa.sol#L1172
Status	Unresolved

Description

The contract automatically burns a percentage of the liquidity pair token balance. This architecture is prone to many inconsistencies and vulnerabilities, including the risk of price manipulation and failed transactions. If a large amount of liquidity is removed from the pool through burning, it can cause a decrease in the liquidity of the pool, which can, in turn, result in increased volatility and price fluctuations of the tokens in the pair.

Additionally, the owner has the authority to change the `lpAddress` address, possibly allowing them to burn tokens from any address.

```
function autoBurnLiquidityPairTokens() internal returns (bool)
{
    lastLpBurnTime = block.timestamp;
    uint256 liquidityPairBalance = this.balanceOf(uniswapPair);
    uint256 amountToBurn =
liquidityPairBalance.mul(percentForLPBurn).div(10000);
    address lpAddress = uniswapPair;
    if (amountToBurn > 0) {
        uint256 burnAmount = amountToBurn * _burnRate / 10000;
        if (burnAmount > 0) {
            super._transfer(lpAddress, address(0xdead),
burnAmount);
        }
    }
    IUniswapV2Pair pair = IUniswapV2Pair(lpAddress);
    pair.sync();
    emit AutoNukeLP();
    return true;
}
```

Recommendation

It is recommended to review and adjust the parameters of the auto-liquidity burn mechanism to ensure that it operates optimally.

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.

BC - Blacklists Addresses

Criticality	Critical
Location	LaiKa.sol#L1256,1260
Status	Unresolved

Description

The contract owner has the authority to stop addresses from transactions. The owner may take advantage of it by calling the `blacklistAddress` function.

```
function setBlackList(address addr, bool enable) external
onlyOwner {
    _blackList[addr] = enable;
}
function batchSetBlackList(address [] memory addr, bool enable)
external onlyOwner {
    for (uint i = 0; i < addr.length; i++) {
        _blackList[addr[i]] = enable;
    }
}
```

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.

UPM - Unrestricted Pair Modification

Criticality	Critical
Location	LaiKa.sol#L986
Status	Unresolved

Description

The contract owner is able to modify the main pair variable, `uniswapPair`, to any address. This unrestricted ability can severely disrupt the contract's intended functionality, as highlighted in the `ST` and `BT` findings.

```
function setuniswappair(address pair) public onlyOwner{
    uniswapPair = pair;
}
```

Recommendation

It is recommended to restrict changes to the `uniswapPair` variable. 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.

UTPD - Unverified Third Party Dependencies

Criticality	Critical
Location	LaiKa.sol#L1038,1052,1403,1414
Status	Unresolved

Description

The contract uses an external contract in order to determine the transaction's flow. The external contract is untrusted. As a result, it may produce security issues and harm the transactions.

```
IPledge(pledge).quilitypledge(from,0,addLPLiquidity,amount);
IPledge(pledge).quilitypledge(to,1,removeLPLiquidity,amount);
IPledge(pledge).quilitypledge(accounts[i],0,lpAmount,lpAmount);
IPledge(pledge).quilitypledge(account,0,lpAmount,lpAmount);
```

Recommendation

The contract should use a trusted external source. A trusted source could be either a commonly recognized or an audited contract. The pointing addresses should not be able to change after the initialization.

CO - Code Optimization

Criticality	Minor / Informative
Location	LaiKa.sol
Status	Unresolved

Description

The code is currently not optimized, which affects both its efficiency and maintainability. Refactoring and optimizing the code will improve readability, reduce gas consumption, and make the logic easier to follow. Additionally, a well-optimized codebase facilitates thorough auditing and helps prevent potential vulnerabilities arising from complex or redundant operations.

Recommendation

The team is advised to refactor the contract to enhance its runtime performance. This will not only improve the overall efficiency and responsiveness of the code but also help reduce gas costs associated with executing its functions.

CCR - Contract Centralization Risk

Criticality	Minor / Informative
Location	LaiKa.sol#L928,935,944,947,954,964,971,986,994,998,1207,1223,1234,1238,1242,1260,1266,1271,1394,1411
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 mint(address _address,uint256 _amount)public  returns
(bool)
function setminBalanceSwapToken(uint256 _minBalanceSwapToken)
external onlyOwner
function setMintAddress(address _MintAddress) public onlyOwner
function setContract(address _pledge,address _reward) public
onlyOwner
function setAutoLPBurnSettings(uint256 _frequencyInSeconds,
uint256 _percent, bool _Enabled) external onlyOwner
function excludedFromFees(address account, bool excluded)
external onlyOwner
function setAutomatedMarketMakerPair(address pair, bool value)
public onlyOwner
function setuniswappair(address pair) public onlyOwner
function setLaunch(bool flag) public onlyOwner
function setwhitelaunch(bool flag) public onlyOwner
function setBuyFee(uint256 _burnFee,uint256 _fundFee) external
onlyOwner
function setFundAddress(address _addr) external onlyOwner
function setLPBurnRate(uint256 r) external onlyOwner
function setLastLPBurnTime(uint256 t) external onlyOwner
function setBurnRate(uint256 r) external onlyOwner
function setBlackList(address addr, bool enable) external
onlyOwner
function batchSetBlackList(address [] memory addr, bool enable)
external onlyOwner
function batchExcludedFees(address [] memory addr, bool enable)
external onlyOwner
function batchwhiteList(address [] memory addr, bool enable)
external onlyOwner
function initLPAmounts(address[] memory accounts, uint256
lpAmount) public onlyOwner
function updateLPAmount(address account, uint256 lpAmount)
public onlyOwner
```

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.

MEM - Missing Error Messages

Criticality	Minor / Informative
Location	LaiKa.sol#L929,930,1050,1065,1324
Status	Unresolved

Description

The contract is missing error messages. Specifically, there are no error messages to accurately reflect the problem, making it difficult to identify and fix the issue. As a result, the users will not be able to find the root cause of the error.

```
require(totalSupply()+ _amount <= alltotalsupply)
require(msg.sender == MintAddress)
require(userInfo.lpAmount >= removeLPLiquidity)
require(whitelaunch)
require(balanceOther >= amountOther + rOther)
```

Recommendation

The team is suggested to provide a descriptive message to the errors. This message can be used to provide additional context about the error that occurred or to explain why the contract execution was halted. This can be useful for debugging and for providing more information to users that interact with the contract.

MEE - Missing Events Emission

Criticality	Minor / Informative
Location	LaiKa.sol#L935,944,947,954,964,986,994,998,1207,1223,1234,1238,1242,1256,1266,1271,1394,1411
Status	Unresolved

Description

The contract performs actions and state mutations from external methods that do not result in the emission of events. Emitting events for significant actions is important as it allows external parties, such as wallets or dApps, to track and monitor the activity on the contract. Without these events, it may be difficult for external parties to accurately determine the current state of the contract.

```
function setminBalanceSwapToken(uint256 _minBalanceSwapToken)
external onlyOwner
function setMintAddress(address _MintAddress) public onlyOwner
function setContract(address _pledge,address _reward) public
onlyOwner
function setAutoLPBurnSettings(uint256 _frequencyInSeconds,
uint256 _percent, bool _Enabled) external onlyOwner
function excludedFromFees(address account, bool excluded)
external onlyOwner
function setuniswappair(address pair) public onlyOwner
function setLaunch(bool flag) public onlyOwner
function setwhitelaunch(bool flag) public onlyOwner
function setBuyFee(uint256 _burnFee,uint256 _fundFee) external
onlyOwner
function setFundAddress(address _addr) external onlyOwner
function setLPBurnRate(uint256 r) external onlyOwner
function setLastLPBurnTime(uint256 t) external onlyOwner
function setBurnRate(uint256 r) external onlyOwner
function setBlackList(address addr, bool enable) external
onlyOwner
function batchSetBlackList(address [] memory addr, bool enable)
external onlyOwner
function batchExcludedFees(address [] memory addr, bool enable)
external onlyOwner
function batchwhiteList(address [] memory addr, bool enable)
external onlyOwner
function initLPAmounts(address[] memory accounts, uint256
lpAmount) public onlyOwner
function updateLPAmount(address account, uint256 lpAmount)
public onlyOwner
```

Recommendation

It is recommended to include events in the code that are triggered each time a significant action is taking place within the contract. These events should include relevant details such as the user's address and the nature of the action taken. By doing so, the contract will be more transparent and easily auditable by external parties. It will also help prevent potential issues or disputes that may arise in the future.

PLPI - Potential Liquidity Provision Inadequacy

Criticality	Minor / Informative
Location	LaiKa.sol#L1148
Status	Unresolved

Description

The contract operates under the assumption that liquidity is consistently provided to the pair between the contract's token and the native currency. However, there is a possibility that liquidity is provided to a different pair. This inadequacy in liquidity provision in the main pair could expose the contract to risks. Specifically, during eligible transactions, where the contract attempts to swap tokens with the main pair, a failure may occur if liquidity has been added to a pair other than the primary one. Consequently, transactions triggering the swap functionality will result in a revert.

```
uniswapRouter.swapExactTokensForTokensSupportingFeeOnTransferTo
kens(
    tokenAmount/2,
    0,
    path,
    fundAddress,
    block.timestamp
);

uniswapRouter.swapExactTokensForTokensSupportingFeeOnTransferTo
kens(
    tokenAmount/2,
    0,
    path,
    reward,
    block.timestamp
);
```

Recommendation

The team is advised to implement a runtime mechanism to check if the pair has adequate liquidity provisions. This feature allows the contract to omit token swaps if the pair does not have adequate liquidity provisions, significantly minimizing the risk of potential failures.

Furthermore, the team could ensure the contract has the capability to switch its active pair in case liquidity is added to another pair.

Additionally, the contract could be designed to tolerate potential reverts from the swap functionality, especially when it is a part of the main transfer flow. This can be achieved by executing the contract's token swaps in a non-reversible manner, thereby ensuring a more resilient and predictable operation.

PSU - Potential Subtraction Underflow

Criticality	Minor / Informative
Location	LaiKa.sol#L1108,1151,1319,1323,1366
Status	Unresolved

Description

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

```
amount -= fees;
...
liquidity = amount * ISwapPair(uniswapPair).totalSupply() /
(balanceOf(uniswapPair) - amount);
...
userInfo.lpAmount -= removeLPLiquidity;
...
amountOther = amount * rOther / (rThis - amount);
...
liquidity = Math.sqrt(amount0 * amount) - 1000;
```

Recommendation

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

RCS - Redundant Conditional Statements

Criticality	Minor / Informative
Location	LaiKa.sol#L1020,1346
Status	Unresolved

Description

The contract contains redundant conditional statements that can be simplified to improve code efficiency and performance. Conditional statements that merely return the result of an expression are unnecessary and lead to larger code size, increased memory usage, and slower execution times. By directly returning the result of the expression, the code can be made more concise and efficient, reducing gas costs and improving runtime performance. Such redundancies are common when simple comparisons or checks are performed within conditional statements, leading to redundant operations.

```
if (address(uniswapRouter) != from)
....
if (address(uniswapRouter) ==
address(0x10ED43C718714eb63d5aA57B78B54704E256024E)) {
    numerator = pairTotalSupply * (rootK - rootKLast) * 8;
    denominator = rootK * 17 + (rootKLast * 8);
} else if (address(uniswapRouter) ==
address(0xD99D1c33F9fC3444f8101754aBC46c52416550D1)) {
    numerator = pairTotalSupply * (rootK - rootKLast);
    denominator = rootK * 3 + rootKLast;
} else if (address(uniswapRouter) ==
address(0xE9d6f80028671279a28790bb4007B10B0595Def1)) {
    numerator = pairTotalSupply * (rootK - rootKLast) * 3;
    denominator = rootK * 5 + rootKLast;
} else {
    numerator = pairTotalSupply * (rootK - rootKLast);
    denominator = rootK * 5 + rootKLast;
}
```

Recommendation

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

RSML - Redundant SafeMath Library

Criticality	Minor / Informative
Location	LaiKa.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 in cases where the explanatory error message is not used.

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

Recommendation

The team is advised to remove the SafeMath library in cases where the revert error message is not used. 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/stable/080-breaking-changes.html#solidity-v0-8-0-breaking-changes.

RSD - Redundant Swap Duplication

Criticality	Minor / Informative
Location	LaiKa.sol#L1148
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.

```
uniswapRouter.swapExactTokensForTokensSupportingFeeOnTransferTo
kens(
    tokenAmount/2,
    0,
    path,
    fundAddress,
    block.timestamp
);

uniswapRouter.swapExactTokensForTokensSupportingFeeOnTransferTo
kens(
    tokenAmount/2,
    0,
    path,
    reward,
    block.timestamp
);
```

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.

UVS - Unsanitized Variable Setters

Criticality	Minor / Informative
Location	LaiKa.sol#L935,944,947,954,964,971,986,994,998,1207,1223,1234,1238,1242,1260,1266,1271,1394,1411
Status	Unresolved

Description

The contract includes several functions that modify critical configuration state variables. These variables are not properly validated or do not implement strict restrictions, which could compromise the contract’s functionality.

```
function setminBalanceSwapToken(uint256 _minBalanceSwapToken)
external onlyOwner
function setMintAddress(address _MintAddress) public onlyOwner
function setContract(address _pledge,address _reward) public
onlyOwner
function setAutoLPBurnSettings(uint256 _frequencyInSeconds,
uint256 _percent, bool _Enabled) external onlyOwner
function excludedFromFees(address account, bool excluded)
external onlyOwner
function setAutomatedMarketMakerPair(address pair, bool value)
public onlyOwner
function setuniswappair(address pair) public onlyOwner
function setLaunch(bool flag) public onlyOwner
function setwhitelaunch(bool flag) public onlyOwner
function setBuyFee(uint256 _burnFee,uint256 _fundFee) external
onlyOwner
function setFundAddress(address _addr) external onlyOwner
function setLPBurnRate(uint256 r) external onlyOwner
function setLastLPBurnTime(uint256 t) external onlyOwner
function setBurnRate(uint256 r) external onlyOwner
function setBlackList(address addr, bool enable) external
onlyOwner
function batchSetBlackList(address [] memory addr, bool enable)
external onlyOwner
function batchExcludedFees(address [] memory addr, bool enable)
external onlyOwner
function batchwhiteList(address [] memory addr, bool enable)
external onlyOwner
function initLPAmounts(address[] memory accounts, uint256
lpAmount) public onlyOwner
function updateLPAmount(address account, uint256 lpAmount)
public onlyOwner
```

Recommendation

The team should refactor the setter functions to ensure that all state variables are properly validated and sanitized.

ZD - Zero Division

Criticality	Minor / Informative
Location	LaiKa.sol#L1319,1323,1359
Status	Unresolved

Description

The contract is using variables that may be set to zero as denominators. This can lead to unpredictable and potentially harmful results, such as a transaction revert.

```
liquidity = amount * ISwapPair(uniswapPair).totalSupply() /
(balanceOf(uniswapPair) - amount);
amountOther = amount * rOther / (rThis - amount);
feeToLiquidity = numerator / denominator;
```

Recommendation

It is important to handle division by zero appropriately in the code to avoid unintended behavior and to ensure the reliability and safety of the contract. The contract should ensure that the divisor is always non-zero before performing a division operation. It should prevent the variables to be set to zero, or should not allow the execution of the corresponding statements.

L02 - State Variables could be Declared Constant

Criticality	Minor / Informative
Location	LaiKa.sol#L898,899,1006,1285
Status	Unresolved

Description

State variables can be declared as constant using the constant keyword. This means that the value of the state variable cannot be changed after it has been set. Additionally, the constant variables decrease gas consumption of the corresponding transaction.

```
uint256 public alltotalsupply = 21000000 * 10 **18
address public USDTAddress =
0x55d398326f99059fF775485246999027B3197955
address private _lastMaybeAddLPAddress
bool public _strictCheck = true
```

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	LaiKa.sol#L6,295,297,301,679,836,892,897,899,928,936,944,947,955,956,957,1004,1208,1224,1230,1254,1283,1285,1290,1375,1380
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.

```
uint256 LpAmount
function DOMAIN_SEPARATOR() external view returns (bytes32);
function PERMIT_TYPEHASH() external pure returns (bytes32);
function MINIMUM_LIQUIDITY() external pure returns (uint256);
function WETH() external pure returns (address);

interface burnedFiAbi {
    function launch() external view returns (bool);

    function setLaunch(bool flag) external;
}
mapping(address => address) public Send
address public MintAddress
address public USDTAddress =
0x55d398326f99059fF775485246999027B3197955
uint256 _amount

...
```

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.

L05 - Unused State Variable

Criticality	Minor / Informative
Location	LaiKa.sol#L1006
Status	Unresolved

Description

An unused state variable is a state variable that is declared in the contract, but is never used in any of the contract's functions. This can happen if the state variable was originally intended to be used, but was later removed or never used.

Unused state variables can create clutter in the contract and make it more difficult to understand and maintain. They can also increase the size of the contract and the cost of deploying and interacting with it.

```
address private _lastMaybeAddLPAddress
```

Recommendation

To avoid creating unused state variables, it's important to carefully consider the state variables that are needed for the contract's functionality, and to remove any that are no longer needed. This can help improve the clarity and efficiency of the contract.

L07 - Missing Events Arithmetic

Criticality	Minor / Informative
Location	LaiKa.sol#L938,959,1210,1235,1239,1243
Status	Unresolved

Description

Events are a way to record and log information about changes or actions that occur within a contract. They are often used to notify external parties or clients about events that have occurred within the contract, such as the transfer of tokens or the completion of a task.

It's important to carefully design and implement the events in a contract, and to ensure that all required events are included. It's also a good idea to test the contract to ensure that all events are being properly triggered and logged.

```
minBalanceSwapToken = _minBalanceSwapToken
lpBurnFrequency = _frequencyInSeconds
burnFee = _burnFee
percentForLPBurn = r
lastLpBurnTime = t
_burnRate = r
```

Recommendation

By including all required events in the contract and thoroughly testing the contract's functionality, the contract ensures that it performs as intended and does not have any missing events that could cause issues with its arithmetic.

L09 - Dead Code Elimination

Criticality	Minor / Informative
Location	LaiKa.sol#L617
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 _burn(address account, uint256 amount) internal
virtual {
        require(account != address(0), "ERC20: burn from the
zero address");

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

        _balances[account] = _balances[account].sub(
            amount,
            "ERC20: burn amount exceeds balance"
        );
        _totalSupply = _totalSupply.sub(amount);
        emit Transfer(account, address(0), amount);
    }
```

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.

L13 - Divide before Multiply Operation

Criticality	Minor / Informative
Location	LaiKa.sol#L1177,1183
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 amountToBurn =
liquidityPairBalance.mul(percentForLPBurn).div(
            10000
        )
uint256 burnAmount = amountToBurn * _burnRate / 10000
```

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.

L14 - Uninitialized Variables in Local Scope

Criticality	Minor / Informative
Location	LaiKa.sol#L1018,1031,1045,1067,1068,1307
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.

```
bool takeFee
uint256 addLPLiquidity
uint256 removeLPLiquidity
uint256 fees
uint256 taxFee
uint256 amountOther
```

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	LaiKa.sol#L945,948,949,989,1226
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.

```
MintAddress = _MintAddress
pledge = _pledge
reward = _reward
uniswapPair = pair
fundAddress = _addr
```

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	LaiKa.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 ^0.8.19;
```

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	LaiKa.sol#L1201
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 {}
```

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.

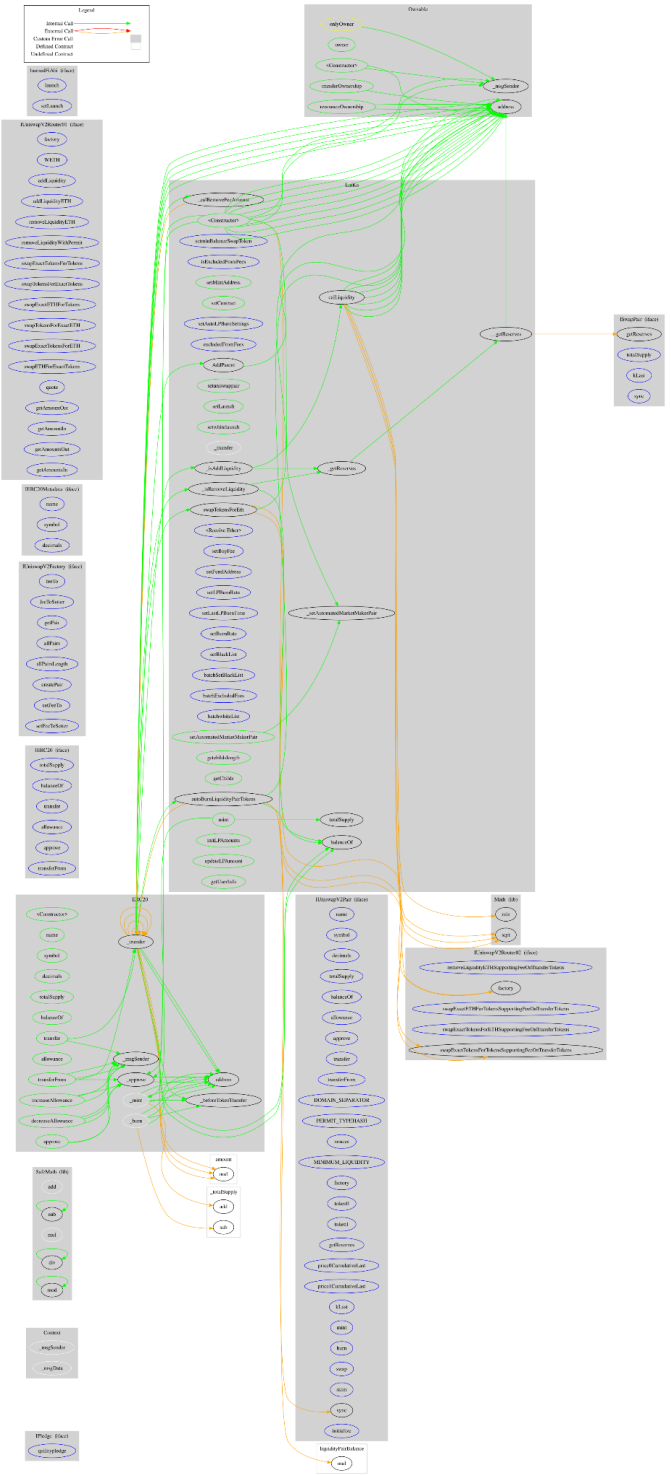
Functions Analysis

Contract	Type	Bases		
	Function Name	**Visibility**	**Mutability**	**Modifiers**
LaiKa	Implementation	ERC20, Ownable		
		Public	✓	ERC20
	mint	Public	✓	-
	setminBalanceSwapToken	External	✓	onlyOwner
	isExcludedFromFees	External		-
	setMintAddress	Public	✓	onlyOwner
	setContract	Public	✓	onlyOwner
	setAutoLPBurnSettings	External	✓	onlyOwner
	excludedFromFees	External	✓	onlyOwner
	setAutomatedMarketMakerPair	Public	✓	onlyOwner
	_setAutomatedMarketMakerPair	Private	✓	
	setuniswappair	Public	✓	onlyOwner
	setLaunch	Public	✓	onlyOwner
	setwhitelaunch	Public	✓	onlyOwner
	_transfer	Internal	✓	
	_calRemoveFeeAmount	Private	✓	
	swapTokensForEth	Private	✓	
	autoBurnLiquidityPairTokens	Internal	✓	
		External	Payable	-
	setBuyFee	External	✓	onlyOwner

	setFundAddress	External	✓	onlyOwner
	setLPBurnRate	External	✓	onlyOwner
	setLastLPBurnTime	External	✓	onlyOwner
	setBurnRate	External	✓	onlyOwner
	setBlackList	External	✓	onlyOwner
	batchSetBlackList	External	✓	onlyOwner
	batchExcludedFees	External	✓	onlyOwner
	batchwhiteList	External	✓	onlyOwner
	AddParent	Private	✓	
	getchildslength	Public		-
	getChilds	Public		-
	_isAddLiquidity	Internal		
	_isRemoveLiquidity	Internal		
	calLiquidity	Private		
	_getReserves	Public		-
	__getReserves	Public		-
	initLPAmounts	Public	✓	onlyOwner
	updateLPAmount	Public	✓	onlyOwner
	getUserInfo	Public		-

Inheritance Graph

Flow Graph



Summary

Laika 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, manipulate the fees, mint tokens, burn tokens from any address and massively blacklist addresses. if the contract owner abuses the mint functionality, then the contract will be highly inflated. If the contract owner abuses the burn functionality, then the users could lose their tokens. 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 TAC 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.

A **TAC Security** Company

The Cyberscope team

cyberscope.io