Cyberscope

Audit Report

QAAGAI

January 2025

Network	BSC
Address	0x7f22a8af38bcf14b92cd65ae8b19260fc0beefdf
Audited by	© cyberscope

Analysis

Critical Medium Minor / Informative Pass

Severity	Code	Description	Status
●	ST	Stops Transactions	Passed
●	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
●	IDI	Immutable Declaration Improvement	Unresolved
●	RC	Redundant Comments	Unresolved
●	RCS	Redundant Conditional Statements	Unresolved
●	RSML	Redundant SafeMath Library	Unresolved
●	RV	Redundant Variables	Unresolved
●	L02	State Variables could be Declared Constant	Unresolved
●	L04	Conformance to Solidity Naming Conventions	Unresolved
●	L15	Local Scope Variable Shadowing	Unresolved
●	L16	Validate Variable Setters	Unresolved
●	L19	Stable Compiler Version	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	QAAGAI
Compiler Version	v0.8.26+commit.8a97fa7a
Optimization	200 runs
Explorer	https://bscscan.com/address/0x7f22a8af38bcf14b92cd65ae8b19260fc0beefdf
Address	0x7f22a8af38bcf14b92cd65ae8b19260fc0beefdf
Network	BSC
Symbol	QAAGAI
Decimals	18
Total Supply	100.000.000

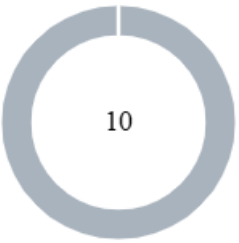
Audit Updates

Initial Audit	27 Jan 2025

Source Files

Filename	SHA256
QAAGAI.sol	16ab7333bb54a51afd03f5490820168d2daaf7ae446d0277586c9c34d77a60e6

Findings Breakdown



Critical	0
Medium	0
Minor / Informative	10

Severity	Unresolved	Acknowledged	Resolved	Other
Critical	0	0	0	0
Medium	0	0	0	0
Minor / Informative	10	0	0	0

IDI - Immutable Declaration Improvement

Criticality	Minor / Informative
Location	QAAGAI.sol#L591,592,593,594
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.

```
maxBuyAmount
maxSellAmount
maxWalletAmount
thresholdSwapAmount
```

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.

RC - Redundant Comments

Criticality	Minor / Informative
Location	QAAGAI.sol#L555,574,615,626,684,700,715,755,765
Status	Unresolved

Description

In the contract there are multiple commented out code segments or code explanations that are not accurate.

```
// exclude from fees and max transaction amount
// mapping(address => bool) private _isExcludedFromFees;
// mapping(address => bool) public
_isExcludedMaxTransactionAmount;
// mapping(address => bool) public _isExcludedMaxWalletAmount;
// store addresses that a automatic market maker pairs. Any
transfer *to* these addresses
// could be subject to a maximum transfer amount
//...
// exclude from paying fees or having max transaction amount
//...
```

Recommendation

It is recommended to remove unnecessary comments to enhance code readability.

RCS - Redundant Conditional Statements

Criticality	Minor / Informative
Location	QAAGAI.sol#L714,753,773,777,788,799,809,827,842
Status	Unresolved

Description

The contract contains redundant conditional statements that can be simplified or removed completely 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.

In the `_transfer` function there is an `if` statement with its block commented out.

```
if (sender != owner() && recipient != owner() && !isSwapping) {
    // The entire code block has been commented out
    //...
}
```

Also in the `_transfer` function, there is an `if` statement that checks if `swapEnabled` is `true` which is always the case.

```
if (canSwap && swapEnabled && !isSwapping &&
marketPair[recipient]) {
    // swapEnabled is always true
    //...
}
```

There are also multiple `if/else if` statements that the code within will never be executed:

```
if (takeFee) {
    uint256 fees = 0;
    if (block.number < taxTill) {
        // taxTill = 0
        // ..unreachable code
    } else if (marketPair[recipient] && _fees.sellTotalFees >
0) {
        // _fees.sellTotalFees = 0
        // ..unreachable code
    }
    // on buy
    else if (marketPair[sender] && _fees.buyTotalFees > 0) {
        // _fees.buyTotalFees = 0
        // ..unreachable code
    }
    if (fees > 0) {
        // fees = 0
        // ..unreachable code
    }
//...
}
```

In the `swapBack` function there is a check if `toSwap == 0` to return. Due the unexecuted code in the `_transfer` function `toSwap` will always be `0`. Therefore, the rest of the code will never be executed.

```
function swapBack() private {
    uint256 contractTokenBalance = balanceOf(address(this));
    uint256 toSwap = tokensForLiquidity + tokensForMarketing +
tokensForDev;
    bool success;
    if (contractTokenBalance == 0 || toSwap == 0) {
        return;
    }
    // tokensForLiquidity = 0
    // tokensForMarketing = 0
    // tokensForDev = 0
    // toSwap = 0
    // ..unreachable code
}
```

The `swapTokensForEth` and `addLiquidity` functions are `private` and are only called inside the `swapBack` function. Since the `swapBack` always returns before they are called the will never be executed.

```
function swapTokensForEth(uint256 tAmount) private {/*...*/}
function addLiquidity(uint256 tAmount, uint256 ethAmount)
private {/*...*/}
```

Recommendation

It is recommended to refactor conditional statements that return results by eliminating unnecessary code structures and directly returning the outcome of the expression. `if` statements that have empty code segment should removed completely. 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	QAAGAI.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.

RV - Redundant Variables

Criticality	Minor / Informative
Location	QAAGAI.sol#L513,514,517,518,519,521,524,525,539,551,552,553,554
Status	Unresolved

Description

In the contract there are multiple variable that are not used or due to their initialization they do not serve any purpose:

- `devWallet` and `marketingWallet` are initialized in the `constructor` and can be changed via the function `setWallets` but the code segment were they are used is unreachable due to the contract's configuration. It should also be mentioned that the contract owner has transferred ownership to the dead address so `setWallets` cannot be used anymore.
- `maxBuyAmount` , `maxSellAmount` and `maxWalletAmount` are initialized in the `constructor` but are not used in the rest of the contract.
- `thresholdSwapAmount` is used in the `_transfer` function in order to initiate the swap but due to the contracts configuration the `swapBack` function always returns without actually doing the swap.
- `isTrading` is never used.
- `swapEnabled` is always true and cannot be changed. `_fees` are initialized in the constructor as zeros and are used in the `_transfer` function, but since the are zeros fees do not apply.
- `tokensForMarketing` , `tokensForLiquidity` , `tokensForDev` are updated in the `_tranfer` and `swapBack` function but the code segments are unreachable due to the code's configuration.
- `taxTill` is used as a time limit in the `_tranfer` function to check if the current timestamp is lower than it and apply huge fees but `taxTill` is `0` .

Recommendation

It is recommended to add only the necessary state variable in a contract since when they are called in functions they increase gas costs. Removing unnecessary variables will also increase code readability.

L02 - State Variables could be Declared Constant

Criticality	Minor / Informative
Location	QAAGAI.sol#L524,525,554
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.

```
bool private isTrading = true
bool public swapEnabled = true
uint256 private taxTill
```

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	QAAGAI.sol#L331,539,693,694
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.

```
Fees public _fees =
        Fees({
            buyTotalFees: 0,
            buyMarketingFee: 0,
            buyDevFee: 0,
            buyLiquidityFee: 0,
            sellTotalFees: 0,
            sellMarketingFee: 0,
            sellDevFee: 0,
            sellLiquidityFee: 0
        })
address _marketingWallet
address _devWallet
```

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.

L15 - Local Scope Variable Shadowing

Criticality	Minor / Informative
Location	QAAGAI.sol#L589
Status	Unresolved

Description

Local scope variable shadowing occurs when a local variable with the same name as a variable in an outer scope is declared within a function or code block. When this happens, the local variable "shadows" the outer variable, meaning that it takes precedence over the outer variable within the scope in which it is declared.

```
uint256 totalSupply = 1e8 * 1e18
```

Recommendation

It's important to be aware of shadowing when working with local variables, as it can lead to confusion and unintended consequences if not used correctly. It's generally a good idea to choose unique names for local variables to avoid shadowing outer variables and causing confusion.

L16 - Validate Variable Setters

Criticality	Minor / Informative
Location	QAAGAI.sol#L696,697
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.

```
marketingWallet = _marketingWallet
devWallet = _devWallet
```

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	QAAGAI.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 ^0.8.2;
```

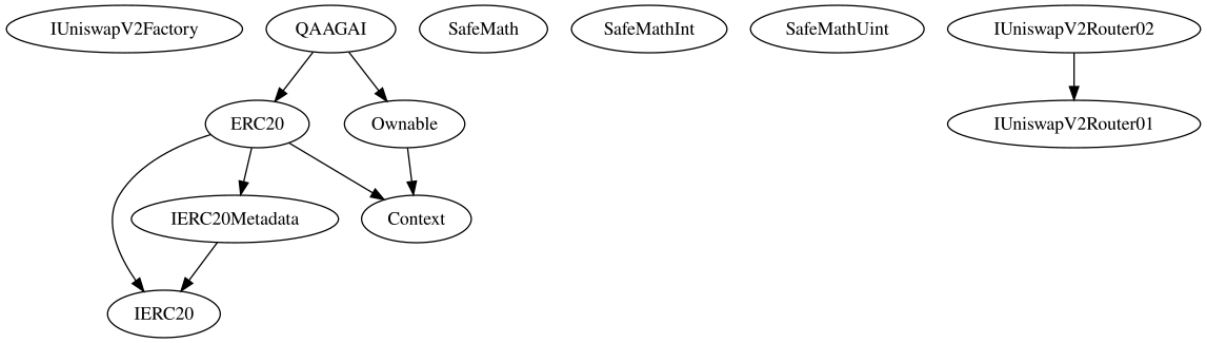
Recommendation

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.

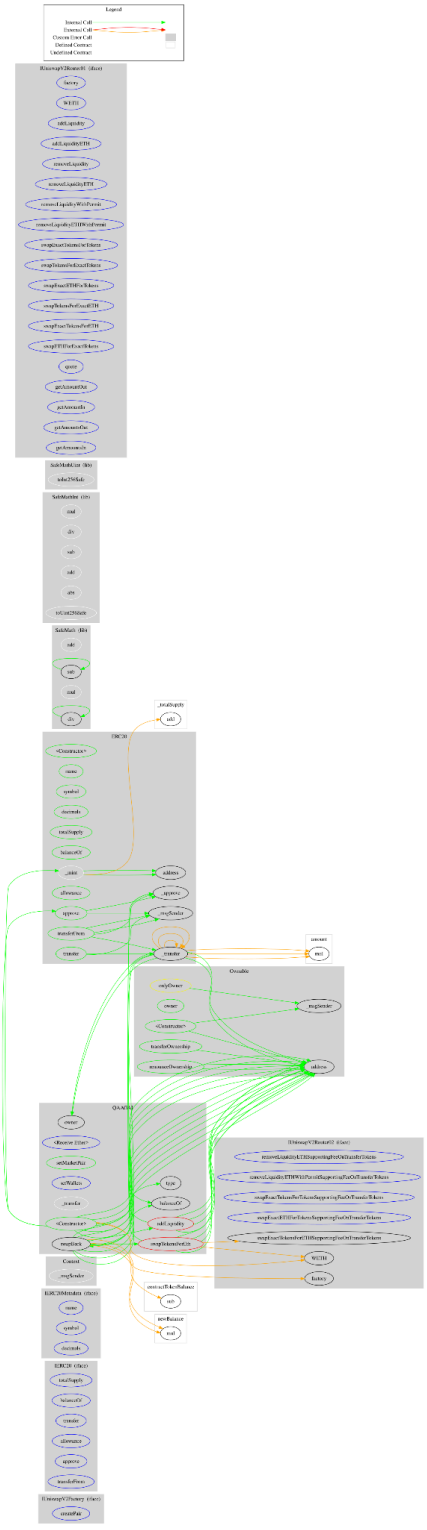
Functions Analysis

Contract	Type	Bases		
	Function Name	**Visibility**	**Mutability**	**Modifiers**
QAAGAI	Implementation	ERC20, Ownable		
		Public	✓	ERC20
		External	Payable	-
	setMarketPair	Public	✓	onlyOwner
	setWallets	External	✓	onlyOwner
	_transfer	Internal	✓	
	swapTokensForEth	Private	✓	
	addLiquidity	Private	✓	
	swapBack	Private	✓	

Inheritance Graph



Flow Graph



Summary

QAAGAI contract implements a token mechanism. This audit investigates security issues, business logic concerns and potential improvements. QAAGAI is an interesting project that has a friendly and growing community. The Smart Contract analysis reported no compiler error or critical issues. The contract Owner has transferred ownership to the dead address. While the contract's code implements fees, they are permanently set to zero.

Disclaimer

The information provided in this report does not constitute investment, financial or trading advice and you should not treat any of the document's content as such. This report may not be transmitted, disclosed, referred to or relied upon by any person for any purposes nor may copies be delivered to any other person other than the Company without Cyberscope's prior written consent. This report is not nor should be considered an "endorsement" or "disapproval" of any particular project or team. This report is not nor should be regarded as an indication of the economics or value of any "product" or "asset" created by any team or project that contracts Cyberscope to perform a security assessment. This document does not provide any warranty or guarantee regarding the absolute bug-free nature of the technology analyzed, nor do they provide any indication of the technologies proprietors' business, business model or legal compliance. This report should not be used in any way to make decisions around investment or involvement with any particular project. This report represents an extensive assessment process intending to help our customers increase the quality of their code while reducing the high level of risk presented by cryptographic tokens and blockchain technology.

Blockchain technology and cryptographic assets present a high level of ongoing risk Cyberscope's position is that each company and individual are responsible for their own due diligence and continuous security Cyberscope's goal is to help reduce the attack vectors and the high level of variance associated with utilizing new and consistently changing technologies and in no way claims any guarantee of security or functionality of the technology we agree to analyze. The assessment services provided by Cyberscope are subject to dependencies and are under continuing development. You agree that your access and/or use including but not limited to any services reports and materials will be at your sole risk on an as-is where-is and as-available basis Cryptographic tokens are emergent technologies and carry with them high levels of technical risk and uncertainty. The assessment reports could include false positives false negatives and other unpredictable results. The services may access and depend upon multiple layers of third parties.

About Cyberscope

Cyberscope is a blockchain cybersecurity company that was founded with the vision to make web3.0 a safer place for investors and developers. Since its launch, it has worked with thousands of projects and is estimated to have secured tens of millions of investors' funds.

Cyberscope is one of the leading smart contract audit firms in the crypto space and has built a high-profile network of clients and partners.

The Cyberscope team

cyberscope.io