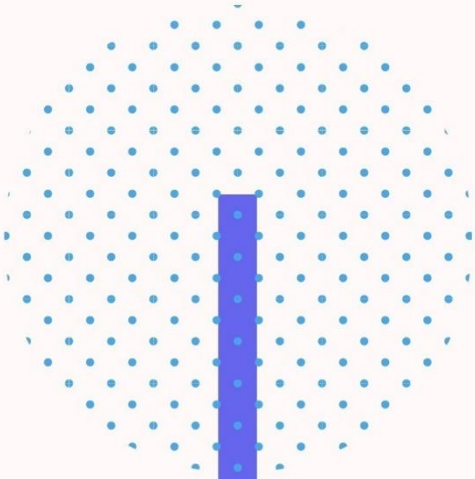
GOGO FINANCE

AUDIT REPORT



Disclaimer

This is a limited report on our findings based on our analysis, in accordance with good industry practice as at the date of this report, in relation to cybersecurity vulnerabilities and issues in the framework and algorithms based on smart contracts, the details of which are set out in this report. In order to get a full view of our analysis, it is crucial for you to read the full report. While we have done our best in conducting our analysis and producing this report, it is important to note that you should not rely on this report and cannot claim against us on the basis of what it says or doesn't say, or how we produced it, and it is important for you to conduct your own independent investigations before making any decisions. We go into more detail on this in the below disclaimer below – please make sure to read it in full.

DISCLAIMER: By reading this report or any part of it, you agree to the terms of this disclaimer. If you do not agree to the terms, then please immediately cease reading this report, and delete and destroy any and all copies of this report downloaded and/or printed by you. This report is provided for information purposes only and on a non-reliance basis, and does not constitute investment advice. No one shall have any right to rely on the report or its contents, and TechRate and its affiliates (including holding companies, shareholders, subsidiaries, employees, directors, officers and other representatives) (TechRate) owe no duty of care towards you or any other person, nor does TechRate make any warranty or representation to any person on the accuracy or completeness of the report. The report is provided "as is", without any conditions, warranties or other terms of any kind except as set out in this disclaimer, and TechRate hereby excludes all representations, warranties, conditions and other terms (including, without limitation, the warranties implied by law of satisfactory quality, fitness for purpose and the use of reasonable care and skill) which, but for this clause, might have effect in relation to the report. Except and only to the extent that it is prohibited by law, TechRate hereby excludes all liability and responsibility, and neither you nor any other person shall have any claim against TechRate, for any amount or kind of loss or damage that may result to you or any other person (including without limitation, any direct, indirect, special, punitive, consequential or pure economic loss or damages, or any loss of income, profits, goodwill, data, contracts, use of money, or business interruption, and whether in delict, tort (including without limitation negligence), contract, breach of statutory duty, misrepresentation (whether innocent or negligent) or otherwise under any claim of any nature whatsoever in any jurisdiction) in any way arising from or connected with this report and the use, inability to use or the results of use of this report, and any reliance on this report.

The analysis of the security is purely based on the smart contracts alone. No applications or operations were reviewed for security. No product code has been reviewed.

1. Introduction

1.1. About Project

- Project Name: **GOGO Finance**
- GOGO Finance is a cross-chain Automated Market Maker (AMM) connector that provides a decentralized Liquidity Tap for various tokens. The Liquidity Tap is the powerhouse of Liquidity Pools, deploying a dynamic algorithm that provides Liquidity Providers with fairer and more efficient incentives overtime.
- Based on its basic functions, GOGO also developed and employed the DeFi trading aggregator. This function will allow each liquidity provider to participate and withdraw with one-click. After comparing the asset prices and APY (annual percentage yield) of several liquidity pools in different AMM platforms, it was observed that through the most optimized path to decrease the trading slippage and gas fee on chains, the complexity of frequently switching between platforms and comprehensive trading cost can also be reduced.

1.2. Audit Goal

Category	Content	Result
Data Validation Issues	Incorrect Behavior Order: Early Validation,Permissive List of Allowed Inputs,Unchecked Input for Loop Condition	PASS
Random Number Issues	Small Space of Random Values,Incorrect Usage of Seeds in Pseudo-Random Number Generator (PRNG)	PASS
State Issues	External Control of System or Configuration Setting,Incomplete Internal State Distinction,Passing Mutable Objects to an Untrusted Method	PASS
Error Conditions, Return Values, Status Codes	Unchecked Return Value,Unexpected Status Code or Return Value,Reachable Assertion,Detection of Error Condition Without Action	PASS
Data Processing Errors	Collapse of Data into Unsafe Value,Improper Handling of Parameters, Comparison of Incompatible Types	PASS
Bad Coding Practices	Missing Default Case in Switch Statement,Excessive Index Range Scan for a Data Resource,Excessive Platform Resource Consumption within a Loop	PASS
Permission Issues	Incorrect Default Permissions,Incorrect Execution-Assigned Permissions,Improper Preservation of Permissions	PASS
Business Logic Errors	Unverified Ownership,Incorrect Ownership Assignment,Allocation of Resources Without Limits or Throttling	PASS

2. Findings

2.1. Data Validation Issues - PASS

Weaknesses in this category are related to a software system's components for input validation, output validation, or other kinds of validation. Validation is a frequently-used technique for ensuring that data conforms to expectations before it is further processed as input or output. There are many varieties of validation). Validation is distinct from other techniques that attempt to modify data before processing it, although developers may consider all attempts to product "safe" inputs or outputs as some kind of validation. Regardless, validation is a powerful tool that is often used to minimize malformed data from entering the system, or indirectly avoid code injection or other potentially-malicious patterns when generating output. The weaknesses in this category could lead to a degradation of the quality of data flow in a system if they are not addressed.

Test results: No related vulnerabilities in smart contract code. Safety advice:None.

2.2. Random Number Issues - PASS

Weaknesses in this category are related to a software system's random number generation. The number of possible random values is smaller than needed by the product, making it more susceptible to brute force attacks. The code uses a Pseudo-Random Number Generator (PRNG) that does not correctly manage seeds.

Test results: No related vulnerabilities in smart contract code. Safety advice:None.

2.3. State Issues - PASS

Weaknesses in this category are related to improper management of system state. One or more system settings or configuration elements can be externally controlled by a user. The software does not properly determine which state it is in, causing it to assume it is in state X when in fact it is in state Y, causing it to perform incorrect operations in a security-relevant manner.

Test results: No related vulnerabilities in smart contract code. Safety advice:None.

2.4. Error Conditions, Return Values, Status Codes - PASS

This category includes weaknesses that occur if a function does not generate the correct return/status code, or if the application does not handle all possible return/status codes that could be generated by a function. This type of problem is most often found in conditions that are rarely encountered during the normal operation of the product. Presumably, most bugs related to common conditions are found and eliminated during development and testing. In some cases, the attacker can directly control or influence the environment to trigger the rare conditions.

Test results: No related vulnerabilities in smart contract code. Safety advice:None.

2.5. Data Processing Errors - PASS

Weaknesses in this category are typically found in functionality that processes data. Data processing is the manipulation of input to retrieve or save information. The software filters data in a way that causes it to be reduced or "collapsed" into an unsafe value that violates an expected security property. The software does not properly handle when the expected number of values for parameters, fields, or arguments is not provided in input, or if those values are undefined.

Test results: No related vulnerabilities in smart contract code. Safety advice:None.

2.6. Bad Coding Practices - PASS

Weaknesses in this category are related to coding practices that are deemed unsafe and increase the chances that an exploitable vulnerability will be present in the application. These weaknesses do not directly introduce a vulnerability, but indicate that the product has not been carefully developed or maintained. If a program is complex, difficult to maintain, not portable, or shows evidence of neglect, then there is a higher likelihood that weaknesses are buried in the code.

Test results: No related vulnerabilities in smart contract code. Safety advice:None.

2.7. Permission Issues - PASS

Weaknesses in this category are related to improper assignment or handling of permissions. While it is executing, the software sets the permissions of an object in a way that violates the intended permissions that have been specified by the user. The software does not preserve permissions or incorrectly preserves permissions when copying, restoring, or sharing objects, which can cause them to have less restrictive permissions than intended.

Test results: No related vulnerabilities in smart contract code. Safety advice:None.

2.8. Business Logic Errors - PASS

Weaknesses in this category identify some of the underlying problems that commonly allow attackers to manipulate the business logic of an application. Errors in business logic can be devastating to an entire application. They can be difficult to find automatically, since they typically involve legitimate use of the application's functionality. However, many business logic errors can exhibit patterns that are similar to well-understood implementation and design weaknesses.

Test results: No related vulnerabilities in smart contract code. Safety advice:None.

2.9. Complexity Issues - PASS

Weaknesses in this category are associated with things being overly complex. The code uses a loop with a control flow condition based on a value that is updated within the body of the loop or contains a callable or other code grouping in which the nesting / branching is too deep.

Test results: No related vulnerabilities in smart contract code. Safety advice:None.

3. Source code

```
/**
 *Submitted for verification at Etherscan.io on 2021-01-06
*/

pragma solidity ^0.4.24;

// ----------------------------------------------------------------------------
// Sample token contract
//
// Symbol        : GOGO
// Name          : GOGO Finance Token
// Total supply  : 265000000000000000000
// Decimals      : 18
// Owner Account : 0x3F31A226e0fCb57d696316BF948E3E391e86CC04
//
// Enjoy.
//
// ----------------------------------------------------------------------------



// ----------------------------------------------------------------------------
// Lib: Safe Math
// ----------------------------------------------------------------------------
contract SafeMath {

    function safeAdd(uint a, uint b) public pure returns (uint c) {
        c = a + b;
        require(c >= a);
    }

    function safeSub(uint a, uint b) public pure returns (uint c) {
        require(b <= a);
        c = a - b;
    }

    function safeMul(uint a, uint b) public pure returns (uint c) {
        c = a * b;
        require(a == 0 || c / a == b);
    }

    function safeDiv(uint a, uint b) public pure returns (uint c) {
        require(b > 0);
        c = a / b;
    }
```

```
}


/**
ERC Token Standard #20 Interface
https://github.com/ethereum/EIPs/blob/master/EIPS/eip-20-token-standard.md
*/
contract ERC20Interface {
    function totalSupply() public constant returns (uint);
    function balanceOf(address tokenOwner) public constant returns (uint balance);
    function allowance(address tokenOwner, address spender) public constant returns (uint remaining);
    function transfer(address to, uint tokens) public returns (bool success);
    function approve(address spender, uint tokens) public returns (bool success);
    function transferFrom(address from, address to, uint tokens) public returns (bool success);

    event Transfer(address indexed from, address indexed to, uint tokens);
    event Approval(address indexed tokenOwner, address indexed spender, uint tokens);
}


/**
Contract function to receive approval and execute function in one call

Borrowed from MiniMeToken
*/
contract ApproveAndCallFallBack {
    function receiveApproval(address from, uint256 tokens, address token, bytes data) public;
}

/**
ERC20 Token, with the addition of symbol, name and decimals and assisted token transfers
*/
contract GOGOToken is ERC20Interface, SafeMath {
    string public symbol;
    string public  name;
    uint8 public decimals;
    uint public _totalSupply;

    mapping(address => uint) balances;
    mapping(address => mapping(address => uint)) allowed;


    // ------------------------------------------------------------------------
    // Constructor
    // ------------------------------------------------------------------------
    constructor() public {
        symbol = "GOGO";
```

```
    name = "GOGO Finance Token";
    decimals = 18;
    _totalSupply = 265000000000000000000;
    balances[0x3F31A226e0fCb57d696316BF948E3E391e86CC04] = _totalSupply;
    emit Transfer(address(0), 0x3F31A226e0fCb57d696316BF948E3E391e86CC04, _totalSupply);
}


// ------------------------------------------------------------------------
// Total supply
// ------------------------------------------------------------------------
function totalSupply() public constant returns (uint) {
    return _totalSupply  - balances[address(0)];
}


// ------------------------------------------------------------------------
// Get the token balance for account tokenOwner
// ------------------------------------------------------------------------
function balanceOf(address tokenOwner) public constant returns (uint balance) {
    return balances[tokenOwner];
}


// ------------------------------------------------------------------------
// Transfer the balance from token owner's account to to account
// - Owner's account must have sufficient balance to transfer
// - 0 value transfers are allowed
// ------------------------------------------------------------------------
function transfer(address to, uint tokens) public returns (bool success) {
    balances[msg.sender] = safeSub(balances[msg.sender], tokens);
    balances[to] = safeAdd(balances[to], tokens);
    emit Transfer(msg.sender, to, tokens);
    return true;
}


// ------------------------------------------------------------------------
// Token owner can approve for spender to transferFrom(...) tokens
// from the token owner's account
//
// https://github.com/ethereum/EIPs/blob/master/EIPS/eip-20-token-standard.md
// recommends that there are no checks for the approval double-spend attack
// as this should be implemented in user interfaces
// ------------------------------------------------------------------------
function approve(address spender, uint tokens) public returns (bool success) {
    allowed[msg.sender][spender] = tokens;
```

```
    emit Approval(msg.sender, spender, tokens);
    return true;
}


// ------------------------------------------------------------------------
// Transfer tokens from the from account to the to account
//
// The calling account must already have sufficient tokens approve(...)-d
// for spending from the from account and
// - From account must have sufficient balance to transfer
// - Spender must have sufficient allowance to transfer
// - 0 value transfers are allowed
// ------------------------------------------------------------------------
function transferFrom(address from, address to, uint tokens) public returns (bool success) {
    balances[from] = safeSub(balances[from], tokens);
    allowed[from][msg.sender] = safeSub(allowed[from][msg.sender], tokens);
    balances[to] = safeAdd(balances[to], tokens);
    emit Transfer(from, to, tokens);
    return true;
}


// ------------------------------------------------------------------------
// Returns the amount of tokens approved by the owner that can be
// transferred to the spender's account
// ------------------------------------------------------------------------
function allowance(address tokenOwner, address spender) public constant returns (uint remaining) {
    return allowed[tokenOwner][spender];
}


// ------------------------------------------------------------------------
// Token owner can approve for spender to transferFrom(...) tokens
// from the token owner's account. The spender contract function
// receiveApproval(...) is then executed
// ------------------------------------------------------------------------
function approveAndCall(address spender, uint tokens, bytes data) public returns (bool success) {
    allowed[msg.sender][spender] = tokens;
    emit Approval(msg.sender, spender, tokens);
    ApproveAndCallFallBack(spender).receiveApproval(msg.sender, tokens, this, data);
    return true;
}


// ------------------------------------------------------------------------
// Don't accept ETH
```

```
    // ----------------------------------------------------------------------
    function () public payable {
        revert();
    }
}
```

4. Conclusion

Smart contract does not contain any high severity issues, which could affect its logic.