QuillAudits

Audit Report
May, 2021

Contents

Introduction

This audit report highlights the overall security of the GLONK for the code deployed in BSCScan at this link. With this report, we have tried to ensure the reliability of the smart contract by completing the assessment of their system's architecture and smart contract codebase.

Auditing Approach and Methodologies applied

In this audit, we consider the following crucial features of the code.

- Whether the implementation of ERC 20 standards.
- Whether the code is secure.
- Gas Optimization
- Whether the code meets the best coding practices.
- Whether the code meets the SWC Registry issue.

The audit has been performed according to the following procedure:

Manual Audit

- Inspecting the code line by line and revert the initial algorithms of the protocol and then compare them with the specification
- Manually analyzing the code for security vulnerabilities.
- Gas Consumption and optimisation
- Assessing the overall project structure, complexity & quality.
- Checking SWC Registry issues in the code.
- Unit testing by writing custom unit testing for each function.
- Checking whether all the libraries used in the code of the latest version.
- Analysis of security on-chain data.
- Analysis of the failure preparations to check how the smart contract performs in case of bugs and vulnerability.

Automated analysis

- Scanning the project's code base with Mythril, Slither, Echidna, Manticore, others.
- Manually verifying (reject or confirm) all the issues found by tools.
- Performing Unit testing.
- Manual Security Testing (SWC-Registry, Overflow)
- Running the tests and checking their coverage.

Audit Details

Project Name: GLONK
Token Symbol: GLONK
Codebase link: https://bscscan.com/address/0xbD5612F129e081E2a6289ADa05E04014ce7C0810#code
Language: Solidity
Platform and tools: HardHat, Remix, VScode, solhint and other tools mentioned in the automated analysis section.

Audit Goals

The focus of this audit was to verify whether the smart contract is secure, resilient, and working according to the standard specs. The audit activity can be grouped into three categories.

Security
Identifying security related issues within each contract and the system of contract.

Sound Architecture
Evaluating the architect of a system through the lens of established smart contract best practice and general software practice.

Code Correctness and Quality
A full review of the contract source code. The primary areas of focus include

- Correctness.
- Section of code with high complexity.
- Readability.
- Quantity and quality of test coverage.

Issue Categories

Every issue in this report was assigned a severity level from the following:

High severity issues

Issues on this level are critical to the smart contract's performance/ functionality and should be fixed before moving to a live environment.

Medium severity issues

Issues on this level could potentially bring problems and should eventually be fixed.

Low severity issues

Issues on this level are minor details and warnings that can remain unfixed but would be better fixed at some point in the future.

Informational

These are severity four issues which indicate an improvement request, a general question, a cosmetic or documentation error, or a request for information. There is low-to-no impact.

Number of issues per severity

	High	Medium	Low	Informational
Open	0	0	6	4
Closed	0	2	0	0

Manual Audit

SWC Registry test

We have tested some known SWC registry issues. Out of all tests only SWC 102, 103, 107, 113, 114, 116 were found. All are low priority. We have discussed it already below.

Serial No.	Description	Comments
SWC-132	Unexpected Ether balance	Pass: Avoided strict equality checks for the Ether balance in a contract
SWC-131	Presence of unused variables	Pass: No unused variables
SWC-128	DoS With Block Gas Limit	Pass
SWC-122	Lack of Proper Signature Verification	Pass
SWC-120	Weak Sources of Randomness from Chain Attributes	Pass
SWC-119	Shadowing State Variables	Pass: No ambiguous found.
SWC-118	Incorrect Constructor Name	Pass. No incorrect constructor name used
SWC-116	Timestamp Dependence	Found
SWC-115	Authorization through tx.origin	Pass: No tx.origin found
SWC-114	Transaction Order Dependence	Found

Serial No.	Description	Comments
SWC-113	DoS with Failed Call	Found
SWC-112	Delegatecall to Untrusted Callee	Pass
SWC-111	Use of Deprecated Solidity Functions	Pass: No deprecated function used
SWC-108	State Variable Default Visibility	Pass: Explicitly defined visibility for all state variables
SWC-107	Reentrancy	Found
SWC-106	Unprotected SELF-DESTRUCT Instruction	Pass: Not found any such vulnerability
SWC-104	Unchecked Call Return Value	Pass: Not found any such vulnerability
SWC-103	Floating Pragma	Found
SWC-102	Outdated Compiler Version	Found
SWC-101	Integer Overflow and Underflow	Pass

High level severity issues

No issues found

Medium level severity issues

2 medium severity issues found.

1. Contract gains to non-withdrawable BNB via the swapAndLiquify function [Line 763]

Description: The swapAndLiquify function converts half of the contractTokenBalance Glonk tokens to BNB. For every swapAndLiquify function call, a small amount of BNB remains in the contract. This amount grows over time with the swapAndLiquify function being called throughout the life of the contract. The Glonk contract does not contain a method to withdraw these funds, and the BNB will be locked in the Glonk contract forever.

```
760             _tokenTransfer(from,to,amount,takeFee);
761         }
762 
763     function swapAndLiquify(uint256 contractTokenBalance) private lockTheSwap {
764         // split the contract balance into halves
765         uint256 half = contractTokenBalance.div(2);
766         uint256 otherHalf = contractTokenBalance.sub(half);
767 
```

Status: CLOSED

Acknowledgement by the product developer; it can't be changed and the amount will be locked.

2. Centralized risk in addLiquidity [~Line 804]

This finding focuses on the addLiquidity function calls the uniswapV2Router.addLiquidityETH function with the to address specified as owner() for acquiring the generated LP tokens from the Glonk-pool. As a result, the _owner address will accumulate a significant portion of LP tokens over time.

```
801         );
802     }
803 
804     function addLiquidity(uint256 tokenAmount, uint256 ethAmount) private {
805         // approve token transfer to cover all possible scenarios
806         _approve(address(this), address(uniswapV2Router), tokenAmount);
807 
808         // add the liquidity
809         uniswapV2Router.addLiquidityETH{value: ethAmount}(
810             address(this),
811             tokenAmount,
812             0, // slippage is unavoidable
813             0, // slippage is unavoidable
814             owner(),
815             block.timestamp
816         );
```

This is one of the prevalent issues floating around the community and causing a great deal of concern by token holders.

Status: CLOSED
Acknowledgement by the product developer; it can't be changed and the amount will be locked.

Low level severity issues

There were 6 low severity issues found.

1. **Description: Costly Loop**

   The loop in the contract includes state variables like .length of a non-memory array, in the condition of the for loops.
   As a result, these state variables consume a lot more extra gas for every iteration of the 'for' loop.
   The below functions include such loops at the above-mentioned lines:
   includeInReward
   _getCurrentSupply()

```
    function _getCurrentSupply() private view returns(uint256, uint256) {
        uint256 rSupply = _rTotal;
        uint256 tSupply = _tTotal;
        for (uint256 i = 0; i < _excluded.length; i++) {
            if (_rOwned[_excluded[i]] > rSupply || _tOwned[_excluded[i]] > tSupply) return (_rTotal, _tTotal);
            rSupply = rSupply.sub(_rOwned[_excluded[i]]);
            tSupply = tSupply.sub(_tOwned[_excluded[i]]);
        }
        if (rSupply < _rTotal.div(_tTotal)) return (_rTotal, _tTotal);
        return (rSupply, tSupply);
    }
```

   Recommendation: It's quite effective to use a local variable instead of a state variable like .length in a loop.
   For instance,

```
uint256 local_variable = _groupInfo.addresses.length;
 for (uint256 i = 0; i < local_variable; i++) {
        if (_groupInfo.addresses[i] == msg.sender) {
            _isAddressExistInGroup = true;
            _senderIndex = i;
            break;
        }
    }
```

   Reading reference link: https://blog.b9lab.com/getting-loopy-with-solidity-1d51794622ad

2. Description → SWC 102: Outdated Compiler Version

```
pragma solidity ^0.6.12;
// SPDX-License-Identifier: Unlicensed
interface IERC20 {
```

Using an outdated compiler version can be problematic especially if there are publicly disclosed bugs and issues that affect the current compiler version.

Remediation
It is recommended to use a recent version of the Solidity compiler which is Version 0.8.4

3. **Description** → **SWC 103: Floating Pragma**

```
pragma solidity ^0.6.12;
// SPDX-License-Identifier: Unlicensed
interface IERC20 {
```

Contracts should be deployed with the same compiler version and flags that they have been tested with thoroughly. Locking the pragma helps to ensure that contracts do not accidentally get deployed using, for example, an outdated compiler version that might introduce bugs that affect the contract system negatively.

Remediation
Lock the pragma version and also consider known bugs (https://github.com/ethereum/solidity/releases) for the compiler version that is chosen.
Pragma statements can be allowed to float when a contract is intended for consumption by other developers, as in the case of contracts in a library or EthPM package. Otherwise, the developer would need to manually update the pragma in order to compile it locally.

4. **Description: Potential use of "block.timestamp" as source of randomness**
Line no: 800 & 815

```
            path,
            address(this),
            block.timestamp
        );
    }

    function addLiquidity(uint256 tokenAmount, uint256 ethAmount) private {
        // approve token transfer to cover all possible scenarios
        _approve(address(this), address(uniswapV2Router), tokenAmount);

        // add the liquidity
        uniswapV2Router.addLiquidityETH{value: ethAmount}(
            address(this),
            tokenAmount,
            0, // slippage is unavoidable
            0, // slippage is unavoidable
            owner(),
            block.timestamp
        );
    }
```

Contracts often need access to time values to perform certain types of functionality. Values such as block.timestamp, and block.number can give you a sense of the current time or a time delta, however, they are not safe to use for most purposes.

In the case of block.timestamp, developers often attempt to use it to trigger time-dependent events. As Ethereum is decentralized, nodes can synchronize time only to some degree. Moreover, malicious miners can alter the timestamp of their blocks, especially if they can gain advantages by doing so. However, miners can't set a timestamp smaller than the previous one (otherwise the block will be rejected), nor can they set the timestamp too far ahead in the future. Taking all of the above into consideration, developers can't rely on the preciseness of the provided timestamp.

Remediation

Developers should write smart contracts with the notion that block values are not precise, and the use of them can lead to unexpected effects. Alternatively, they may make use of oracles.

References

- Safety: Timestamp dependence
- Ethereum Smart Contract Best Practices - Timestamp Dependence
- How do Ethereum mining nodes maintain a time consistent with the network?
- Solidity: Timestamp dependency, is it possible to do safely?

5. Description: Prefer external to public visibility level

A function with a public visibility modifier that is not called internally. Changing the visibility level to external increases code readability. Moreover, in many cases, functions with external visibility modifiers spend less gas compared to functions with public visibility modifiers. The function definition in the file which are marked as public are below

- "renounceOwnership"
- "transferOwnership"
- "geUnlockTime"
- "lock"
- "unlock"
- "symbol"
- "decimals"
- "totalSupply"
- "transfer"
- "allowance"
- "approve"
- "transferFrom"
- "increaseAllowance"
- "decreaseAllowance"
- "isExcludedFromReward"
- "totalFees"
- "Deliver"
- "reflectionFromToken"
- "excludeFromReward"
- "excludeFromFee"
- "includeInFee"
- "setSwapAndLiquifyEnabled"
- "isExcludedFromFee"

However, it is never directly called by another function in the same contract or in any of its descendants. Consider marking it as "external" instead.
Same issue was found by automated analysis Mythx as well.

Recommendations: Use the **external** visibility modifier for functions never called from the contract via internal call. Reading Link.

6. Using the approve function of the ERC-20 token standard → SWC: 114

The ‘approve’ function of the contract is vulnerable. Using a front-running attack one can spend approved tokens before the change of allowance value.

```
    }

    function approve(address spender, uint256 amount) public override returns (bool) {
        _approve(_msgSender(), spender, amount);
        return true;
    }
```

To prevent attack vectors described above, clients should make sure to create user interfaces in such a way that they set the allowance first to 0 before setting it to another value for the same spender. Though the contract itself shouldn't enforce it, to allow backward compatibility with contracts deployed before.

Detailed reading around it can be found at SWC114 & EIP20

Informational

1. The function signature **swapTokensForEth**(uint256 tokenAmount) does not properly convey the purpose of the function, as the underlying logic actually swaps the Glonk token, for **BNB**, and does not swap the token for **ETH**. It could easily be the case where the team originally set out to launch onto the Ethereum mainnet, but eventually settled on Binance Smart Chain. If this was the case, they could have easily just corrected the contract readability. It could also be the case where they simply copied this logic from another contract.

   The function also mentions the uniswap v2 router in the code, instead of pancakeswap, but Glonk interacts with pancakeswap, which is a clone of uniswap built for Binance Smart Chain. No logic is affected by these syntactic choices, but it only fuels the accusations of this contract mainly consisting of a simple copy-paste-change effort.

2. Variables like _tTotal, numTokensSellToAddToLiquidity, _name, _symbol and _decimals could be declared as constant since these state variables are never changed.

3. The return value of the function **addLiquidityETH** is not properly handled. Variables can be used to receive the return values of functions mentioned above and handle both success and failure cases if needed by the business logic.

```
            path,
            address(this),
            block.timestamp
        );
    }

    function addLiquidity(uint256 tokenAmount, uint256 ethAmount) private {
        // approve token transfer to cover all possible scenarios
        _approve(address(this), address(uniswapV2Router), tokenAmount);

        // add the liquidity
        uniswapV2Router.addLiquidityETH{value: ethAmount}(
            address(this),
            tokenAmount,
            0, // slippage is unavoidable
            0, // slippage is unavoidable
            owner(),
            block.timestamp
        );
    }
```

4. The coding style used to create the Glonk contract does not thoroughly implement the use of Event emitters. Event emitters are used to provide informational context of logic being performed by the contract. There are several functions in the Glonk contract that fail to emit events, but can change state variables used in the contract. One of the eg would be

```
    function includeInFee(address account) public onlyOwner {
        _isExcludedFromFee[account] = false;
    }

    function setTaxFeePercent(uint256 taxFee) external onlyOwner() {
        _taxFee = taxFee;
    }

    function setLiquidityFeePercent(uint256 liquidityFee) external onlyOwner() {
        _liquidityFee = liquidityFee;
    }

    function setMaxTxPercent(uint256 maxTxPercent) external onlyOwner() {
        _maxTxAmount = _tTotal.mul(maxTxPercent).div(
            10**2
        );
    }
```

With these functions being able to change state variables, and only being able to be invoked by the owner of the contract, without events being emitted, this means the contract owner can alter state variables without the public being able to easily see these changes. These changes can still be observed by any party willing to do the work. However, events being emitted make it easier.

Functional test

We did functional tests for different contracts as well manually. Below is the report.

- function transferOwnership: Transfers ownership of the contract to a new account (`newOwner`). Can only be called by the current owner.
  -- > PASS

- function renounceOwnership() : Leaves the contract without owner. It will not be possible to call onlyOwner` functions anymore. Can only be called by the current owner.
  --> PASS

- function owner(): Returns the address of the current owner
  --> PASS

- Function functionCallWithValue: dev Same as {xref-Address-functionCallWithValue-address-bytes-uint256-}
  [`functionCallWithValue`], but with `errorMessage` as a fallback revert reason when `target` reverts.
  --> PASS

- function functionCallWithValue: Same as {xref-Address-functionCall-address-bytes-}[`functionCall`],but also transferring `value` wei to `target`.
  --> PASS

- function functionCall(address target, bytes memory data, string memory errorMessage): Same as {xref-Address-functionCall-address-bytes-}
  [`functionCall`], but with `errorMessage` as a fallback revert reason when `target` reverts. PASS
  function sendValue: sends `amount` wei to `recipient`, forwarding all available gas and reverting on errors.
  --> PASS

- function lock: Locks the contract for the owner for the amount of time provided
  --> PASS

- function unlock() : Unlocks the contract for owner when _lockTime is exceeds
  --> PASS

- function swapAndLiquify: Split the contract balance into halves
  --> PASS

- function swapTokensForEth: generate the uniswap pair path of token -> weth
  --> PASS

- function addLiquidity: approve the token transfer to cover all possible scenario
  --> PASS

- function _tokenTransfer: responsible for taking all fee, if takeFee is true
  --> PASS

Automated Testing

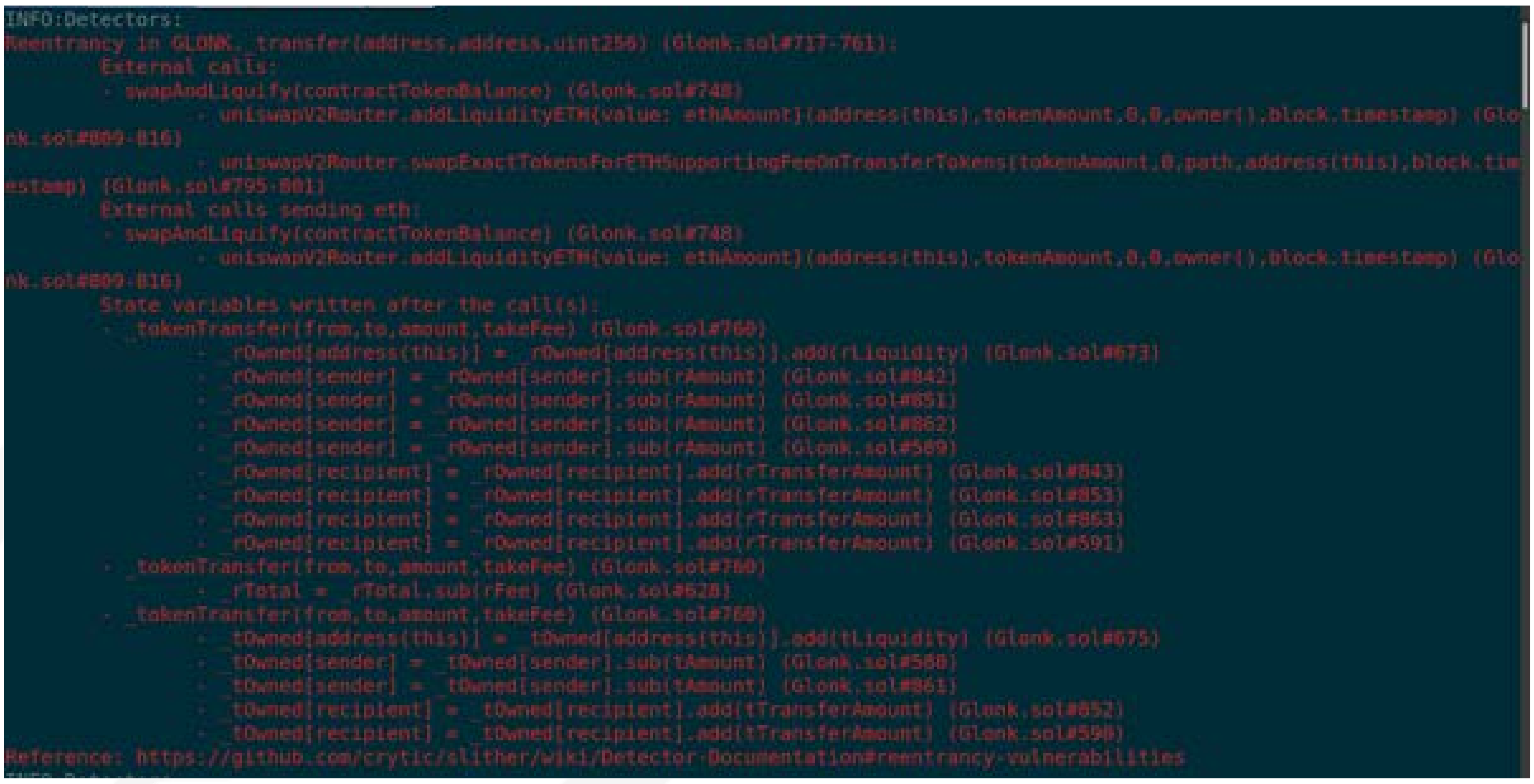
We have used multiple automated testing frameworks. This makes code more secure and common attacks. The results are below.

Remix IDE

Remix was able to compile code perfectly and was behaving according to the required property.

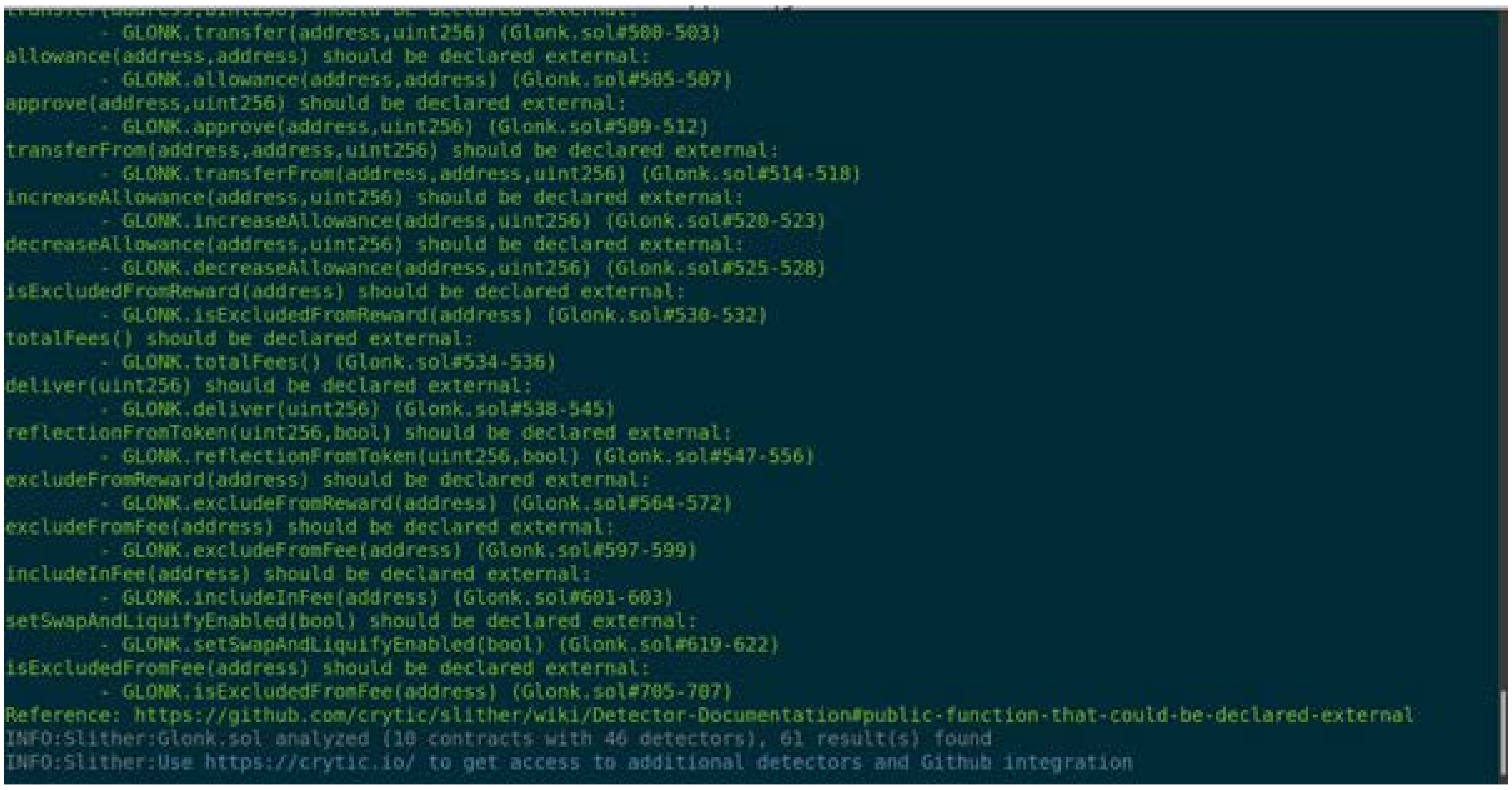
Slither

Slither is a Solidity static analysis framework that runs a suite of vulnerability detectors, prints visual information about contract details, and provides an API to easily write custom analyses. Slither enables developers to find vulnerabilities, enhance their code comprehension, and quickly prototype custom analyses. After running Slither we got the results below.



```
GLONK.addLiquidity(uint256,uint256) (Glonk.sol#804-817) ignores return value by uniswapV2Router.addLiquidityETH{value: ethAmoun
t}(address(this),tokenAmount,0,0,owner(),block.timestamp) (Glonk.sol#809-816)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#unused-return
INFO:Detectors:
GLONK.allowance(address,address).owner (Glonk.sol#505) shadows:
        - Ownable.owner() (Glonk.sol#157-159) (function)
GLONK._approve(address,address,uint256).owner (Glonk.sol#709) shadows:
        - Ownable.owner() (Glonk.sol#157-159) (function)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#local-variable-shadowing
INFO:Detectors:
Reentrancy in GLONK._transfer(address,address,uint256) (Glonk.sol#717-761):
        External calls:
        - swapAndLiquify(contractTokenBalance) (Glonk.sol#748)
                - uniswapV2Router.addLiquidityETH{value: ethAmount}(address(this),tokenAmount,0,0,owner(),block.timestamp) (Glo
nk.sol#809-816)
                - uniswapV2Router.swapExactTokensForETHSupportingFeeOnTransferTokens(tokenAmount,0,path,address(this),block.tim
estamp) (Glonk.sol#795-801)
        External calls sending eth:
        - swapAndLiquify(contractTokenBalance) (Glonk.sol#748)
                - uniswapV2Router.addLiquidityETH{value: ethAmount}(address(this),tokenAmount,0,0,owner(),block.timestamp) (Glo
nk.sol#809-816)
        State variables written after the call(s):
        - _tokenTransfer(from,to,amount,takeFee) (Glonk.sol#760)
                - _liquidityFee = _previousLiquidityFee (Glonk.sol#702)
                - _liquidityFee = 0 (Glonk.sol#697)
        - _tokenTransfer(from,to,amount,takeFee) (Glonk.sol#760)
                - _previousLiquidityFee = _liquidityFee (Glonk.sol#694)
        - _tokenTransfer(from,to,amount,takeFee) (Glonk.sol#760)
                - _previousTaxFee = _taxFee (Glonk.sol#693)
        - _tokenTransfer(from,to,amount,takeFee) (Glonk.sol#760)
                - _tFeeTotal = _tFeeTotal.add(tFee) (Glonk.sol#629)
        - _tokenTransfer(from,to,amount,takeFee) (Glonk.sol#760)
                - _taxFee = _previousTaxFee (Glonk.sol#701)
```

```
        External calls:
        - uniswapV2Pair = IUniswapV2Factory(_uniswapV2Router.factory()).createPair(address(this),_uniswapV2Router.WETH()) (Glon
k.sol#466-467)
        State variables written after the call(s):
        - _isExcludedFromFee[owner()] = true (Glonk.sol#473)
        - _isExcludedFromFee[address(this)] = true (Glonk.sol#474)
        - uniswapV2Router = _uniswapV2Router (Glonk.sol#470)
Reentrancy in GLONK.swapAndLiquify(uint256) (Glonk.sol#763-784):
        External calls:
        - swapTokensForEth(half) (Glonk.sol#775)
                - uniswapV2Router.swapExactTokensForETHSupportingFeeOnTransferTokens(tokenAmount,0,path,address(this),block.tim
estamp) (Glonk.sol#795-801)
        - addLiquidity(otherHalf,newBalance) (Glonk.sol#781)
                - uniswapV2Router.addLiquidityETH{value: ethAmount}(address(this),tokenAmount,0,0,owner(),block.timestamp) (Glo
nk.sol#809-816)
        External calls sending eth:
        - addLiquidity(otherHalf,newBalance) (Glonk.sol#781)
                - uniswapV2Router.addLiquidityETH{value: ethAmount}(address(this),tokenAmount,0,0,owner(),block.timestamp) (Glo
nk.sol#809-816)
        State variables written after the call(s):
        - addLiquidity(otherHalf,newBalance) (Glonk.sol#781)
                - _allowances[owner][spender] = amount (Glonk.sol#713)
Reentrancy in GLONK.transferFrom(address,address,uint256) (Glonk.sol#514-518):
        External calls:
        - _transfer(sender,recipient,amount) (Glonk.sol#515)
                - uniswapV2Router.addLiquidityETH{value: ethAmount}(address(this),tokenAmount,0,0,owner(),block.timestamp) (Glo
nk.sol#809-816)
                - uniswapV2Router.swapExactTokensForETHSupportingFeeOnTransferTokens(tokenAmount,0,path,address(this),block.tim
estamp) (Glonk.sol#795-801)
        External calls sending eth:
        - _transfer(sender,recipient,amount) (Glonk.sol#515)
                - uniswapV2Router.addLiquidityETH{value: ethAmount}(address(this),tokenAmount,0,0,owner(),block.timestamp) (Glo
nk.sol#809-816)
```



```
transfer(address,uint256) should be declared external:
        - GLONK.transfer(address,uint256) (Glonk.sol#500-503)
allowance(address,address) should be declared external:
        - GLONK.allowance(address,address) (Glonk.sol#505-507)
approve(address,uint256) should be declared external:
        - GLONK.approve(address,uint256) (Glonk.sol#509-512)
transferFrom(address,address,uint256) should be declared external:
        - GLONK.transferFrom(address,address,uint256) (Glonk.sol#514-518)
increaseAllowance(address,uint256) should be declared external:
        - GLONK.increaseAllowance(address,uint256) (Glonk.sol#520-523)
decreaseAllowance(address,uint256) should be declared external:
        - GLONK.decreaseAllowance(address,uint256) (Glonk.sol#525-528)
isExcludedFromReward(address) should be declared external:
        - GLONK.isExcludedFromReward(address) (Glonk.sol#530-532)
totalFees() should be declared external:
        - GLONK.totalFees() (Glonk.sol#534-536)
deliver(uint256) should be declared external:
        - GLONK.deliver(uint256) (Glonk.sol#538-545)
reflectionFromToken(uint256,bool) should be declared external:
        - GLONK.reflectionFromToken(uint256,bool) (Glonk.sol#547-556)
excludeFromReward(address) should be declared external:
        - GLONK.excludeFromReward(address) (Glonk.sol#564-572)
excludeFromFee(address) should be declared external:
        - GLONK.excludeFromFee(address) (Glonk.sol#597-599)
includeInFee(address) should be declared external:
        - GLONK.includeInFee(address) (Glonk.sol#601-603)
setSwapAndLiquifyEnabled(bool) should be declared external:
        - GLONK.setSwapAndLiquifyEnabled(bool) (Glonk.sol#619-622)
isExcludedFromFee(address) should be declared external:
        - GLONK.isExcludedFromFee(address) (Glonk.sol#705-707)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#public-function-that-could-be-declared-external
INFO:Slither:Glonk.sol analyzed (10 contracts with 46 detectors), 61 result(s) found
INFO:Slither:Use https://crytic.io/ to get access to additional detectors and Github integration
```

Manticore

Manticore is a symbolic execution tool for the analysis of smart contracts and binaries. It executes a program with symbolic inputs and explores all the possible states it can reach. It also detects crashes and other failure cases in binaries and smart contracts.
Manticore results throw the same warning which is similar to the Slither warning.

MythX

MythX is a security analysis tool and API that performs static analysis, dynamic analysis, symbolic execution, and fuzzing on Ethereum smart contracts. MythX checks for and reports on the common security vulnerabilities in open industry-standard SWC Registry.
There are many contracts within the whole file. We have separately put them for analysis. Below are the reports generated for each contract separately.
Report 1
Report 2

Analysis a316d448-619c-4246-bfc8-789cd9067903 MythX

Started	Fri May 28 2021 08:19:10 GMT+0000 (Coordinated Universal Time)
Finished	Fri May 28 2021 08:34:32 GMT+0000 (Coordinated Universal Time)
Mode	Standard
Client Tool	Remythx
Main Source File	Quill/Glonk.Sol

DETECTED VULNERABILITIES

HIGH	MEDIUM	LOW
0	26	3

All the issues found are already discussed in manual analysis.

Anchain

Anchain sandbox audits the security score of any Solidity-based smart contract, having analyzed the source code of every mainnet EVM smart contract plus the 1M + unique, user-uploaded smart contracts. Code has been analyzed there and got the report below.

ETH Smart Contract Audit Report

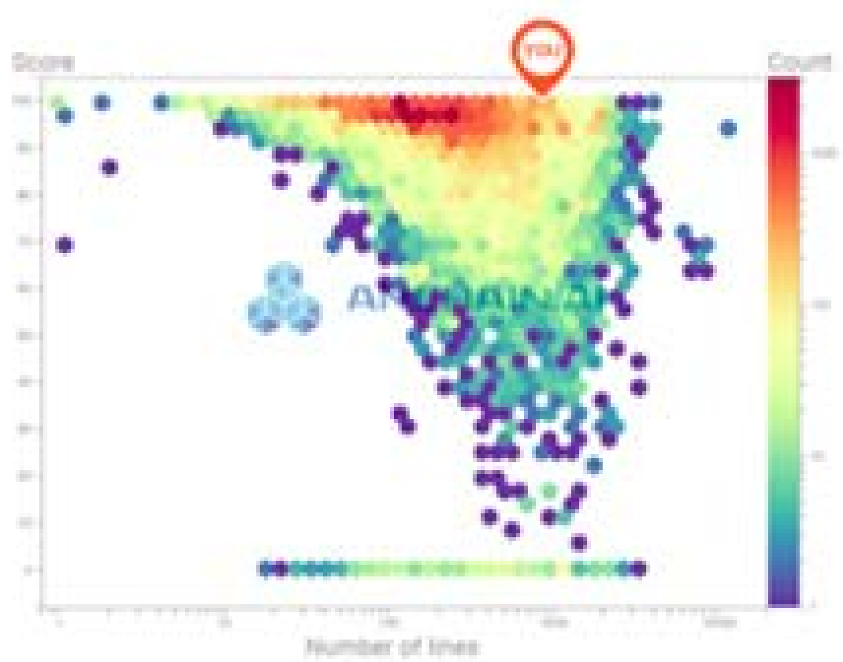
MD5:07c2c3598c5a749826dffc0b9cdd4aad Runtime:7

Scored higher than 100% of similar code
amongst 50k smart contracts audited by Anchain.AI .

Score
100

Threat Level
Low

Number of lines
872



Overview

Code Class	EVM Coverage
Address	55.6%
Context	0%
IERC20	0%
IUniswapV2Pair	0%
IUniswapV2Router02	0%
SafeMath	58.8%

0 Vulnerabilities Found

High Risk Medium Risk Low R

Vulnerability Checklist

Address

- Integer Underflow
- Integer Overflow
- Parity Multisig Bug
- Callstack Depth Attack
- Transaction-Ordering Dependency
- Timestamp Dependency
- Re-Entrancy

Context

- Integer Underflow
- Integer Overflow
- Parity Multisig Bug
- Callstack Depth Attack
- Transaction-Ordering Dependency
- Timestamp Dependency
- Re-Entrancy

IERC20

- Integer Underflow
- Integer Overflow
- Parity Multisig Bug
- Callstack Depth Attack
- Transaction-Ordering Dependency

Disclaimer

The audit does not give any warranties on the security of the code. One audit cannot be considered enough. We always recommend proceeding with several independent audits and
a public bug bounty program to ensure the security of the code. Besides a security audit, please don’t consider this report as investment advice.

Summary

The use of smart contracts is simple and the code is relatively small. Altogether the code is written and demonstrates effective use of abstraction, separation of concern, and modularity. But there are a few issues/vulnerabilities to be tackled at various security levels, it is recommended to fix them before deploying the contract on the main network. Given the subjective nature of some assessments, it will be up to the GLONK team to decide whether any changes should be made.

QuillAudits

Canada, India, Singapore and Uniked Kingdom

audits.quillhash.com

audits@quillhash.com