QuillAudits

Audit Report

October, 2020

GG FINANCE

Contents

Introduction

This Audit Report highlights the overall security of the GGFinance Smart Contract. With this report, we have tried to ensure the reliability of their smart contract by complete assessment of their system's architecture and the smart contract codebase.

Auditing Approach and Methodologies applied

The Quillhash team has performed thorough testing of the project starting with analysing the code design patterns in which we reviewed the smart contract architecture to ensure it is structured and safe use of third party smart contracts and libraries.

Our team then performed a formal line by line inspection of the Smart Contract to find any potential issue like race conditions, transaction-ordering dependence, timestamp dependence, and denial of service attacks.

In the Unit testing Phase, we coded/conducted Custom unit tests written for each function in the contract to verify that each function works as expected. In Automated Testing, We tested the Smart Contract with our in-house developed tools to identify vulnerabilities and security flaws.

The code was tested in collaboration of our multiple team members and this included -

- Testing the functionality of the Smart Contract to determine proper logic has been followed throughout the process.
- Analysing the complexity of the code by thorough, manual review of the code, line-by-line.
- Deploying the code on testnet using multiple clients to run live tests
- Analysing failure preparations to check how the Smart Contract performs in case of bugs and vulnerabilities.
- Checking whether all the libraries used in the code are on the latest version.
- Analysing the security of the on-chain data.

Audit Details

Project Name: GGFinance
Website/Etherscan Code(Mainnet): 0xa33529a0F240e5754BBFAD4786439365BD096dE3
Languages: Solidity (Smart contract)
Platforms and Tools: Remix IDE, Truffle, Truffle Team, Ganache, Solhint, VScode, Securify, Mythril, Contract Library, Slither, SmartCheck

Audit Goals

The focus of the audit was to verify that the smart contract system is secure, resilient and working according to its specifications. The audit activities can be grouped in the following three categories:

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

Sound Architecture
Evaluation of the architecture of this system through the lens of established smart contract best practices and general software best practices.

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

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

Security

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
They could potentially bring problems add should eventually be fixed.

Low severity issues
They are minor details and warnings that can remain unfixed but would be better fixed at some point in the future.

Number of issues per severity

	Low	Medium	High
Open	2	7	0
Closed	0	0	0

Manual Audit

For this section the code was tested/read line by line by our developers. We also used Remix IDE's JavaScript VM and Kovan networks to test the contract functionality.

Low Level Severity Issues

1. The pragama versions used within these contracts are too recent and are not locked as well. Consider using version 0.5.11 for deploying the contracts. Solidity source files indicate the versions of the compiler they can be compiled with.

   ```
   pragma solidity ^0.7.0; // bad: compiles with 0.7.0 and above
   pragma solidity 0.7.0; // good : compiles w 0.7.0 only
   ```

   It is recommended to follow the latter example, as future compiler versions may handle certain language constructions in a way the developer did not foresee.

2. Input parameter _newOwner is declared payable whilst the address parameter (newOwner) storing its value is not payable. It is recommended to remove the payable keyword from input argument declaration.

```
address newOwner;
function changeOwner(address payable _newOwner) public onlyOwner {
    newOwner = _newOwner;
}
```

Medium Severity Issues

1. Visibility of variables should be explicitly specified. The variables owner and newOwner have been defaulted to internal and therefore cannot be read via contract calls.

```
address owner;
address newOwner;
```

We recommend adding external keywords to both variable declarations

2. Visibility of changeOwner as well as acceptOwnership function should be updated to “external” from “public”. As these functions have not been called by any other contract function. This would help save gas during function calling.

```
function changeOwner(address payable _newOwner) public onlyOwner {
    newOwner = _newOwner;
}
function acceptOwnership() public {
    if (msg.sender == newOwner) {
        owner = newOwner;
    }
}
```

This is important because the owner is assigned the initial token supply, and can only transfer ownership to a new address. Therefore the address information is recommended to be made public via the contract.

The following list of functions should be made external for saving gas during function calls:

- changeOwner(address)
- acceptOwnership()
- balanceOf(address)
- transfer(address,uint256)
- transferFrom(address,address,uint256)
- approve(address,uint256)
- allowance(address,address)

Note: *There are four types of visibilities for functions and state variables. Functions can be specified as being external, public, internal or private, where the default is public. For state variables, external is not possible and the default is internal.*

3. Function acceptOwnership should revert if the message sender is not the newOwner. Instead of wasting 22311 gas as transaction cost. There is no simple way to identify if the function execution was successful or a failure.

```
function acceptOwnership() public {
    if (msg.sender==newOwner) {
        owner = newOwner;
    }
}
```

4. Unnecessary receive function is defined within the contracts, we recommend removing it because solidity reverts by default when there is no fallback or receive function defined. To know more read *this* document on receive function.

5. The ownership contract Owned is useless as the owner has no operation other than receiving initial supply and the new owner won't inherit this supply. Also there are no owner bound functions and therefore it is recommended to remove the Owned contract.

6. Function transferFrom should return an Approve event showcasing new Transfer value. During transferFrom we use the approved allowance of a token holder and spend it, this new reduced value should be emitted via an event in transferFrom.

7. Should use *OpenZeppelin's SafeMath.sol* to avoid possible integer underflow/overflow. An overflow/underflow happens when an arithmetic operation reaches the maximum or minimum size of a type.

```
function transferFrom(address _from,address _to,uint256 _amount) public returns (bool success) {
    require (balances[_from]>=_amount&&allowed[_from][msg.sender]>=_amount&&_amount>0&&balances[_to]+_amount>balances[_to]);
    balances[_from]-=_amount;
    allowed[_from][msg.sender]-=_amount;
    balances[_to]+=_amount;
    emit Transfer(_from, _to, _amount);
    return true;
}
```

Never write mathematical calculations using bare arithmetic operators like plus, minus, divide, and multiply. Unless you specifically check for under and overflow vulnerabilities, you can't guarantee the calculations will be safe. This is where SafeMath comes in. It performs all the required checks you need to be confident that your calculations run correctly, without introducing vulnerabilities to your code

High severity issues

No high severity issues

Recommendations

1. A check should be placed to revert if the input argument to a function is a zeroAddress. The check should be applied to wherever addresses are taken as input, which includes: transfer, transferFrom and approve. Like a normal Ethereum address, the zero-address is also 20 bytes long but contains only empty bytes. Hence its name zero-address, since it contains only 0x0 values. As this address cannot be recreated money sent to this address is lost forever. This check can also be added to the backend/frontend handling the contract and therefore is only a recommendation.

2. GGfinance contract does not use custom events specific to the contract functionality. Events should be fired with all state variable updates as good practise. This makes it easier to build dApps on top of the contract's using existing tools. For instance when the owner is changed or when the contract receives ethers.

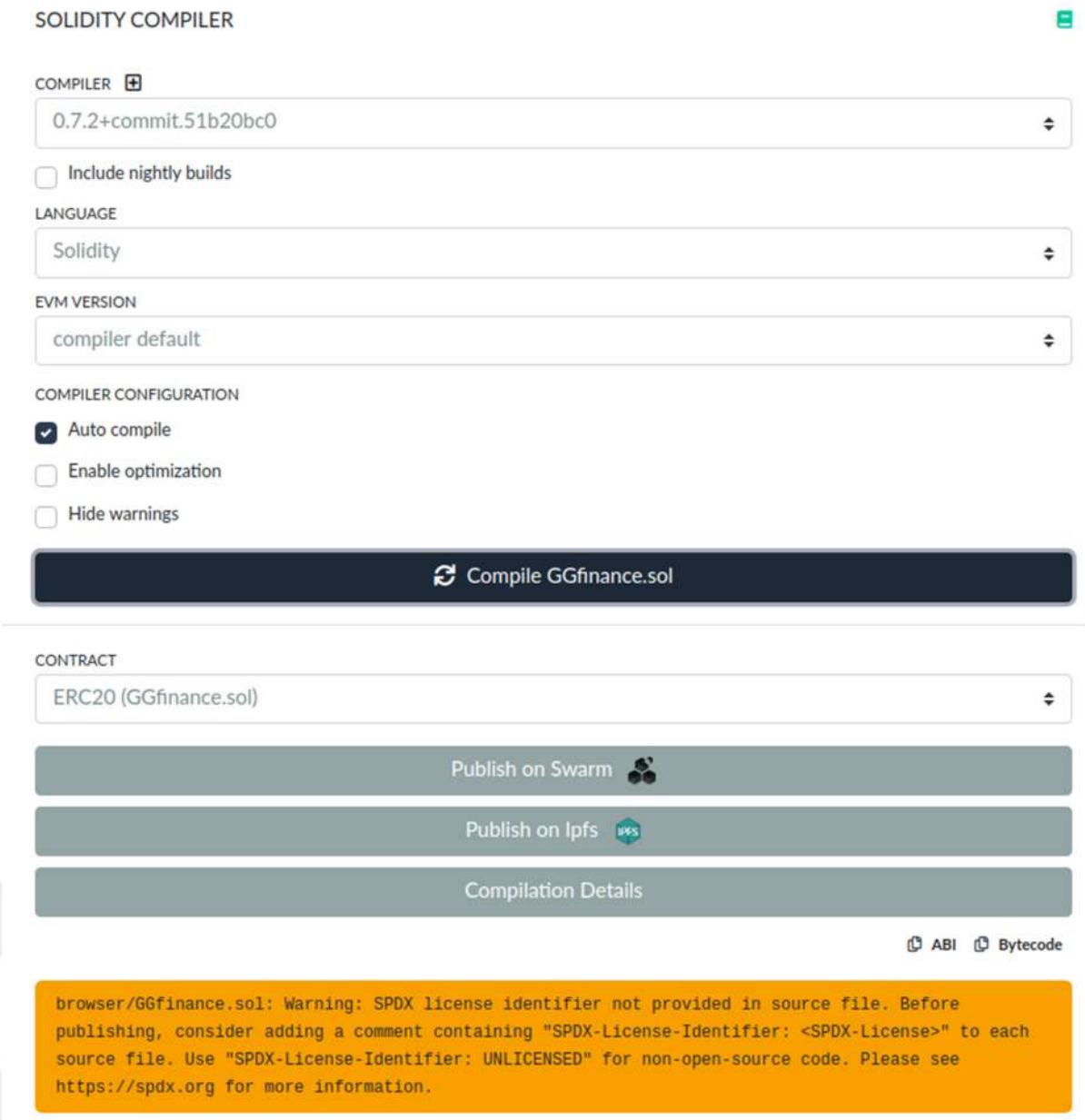
3. *Natspecs* should be used to improve code readability. NatSpec comments are a way to describe the behaviour of a function to end-users. It also allows us to provide more detailed information to contract readers. NatSpec includes the formatting for comments that the smart contract author will use, and which are understood by the Solidity compiler

4. All the "require" statements used in the contract should also specify error messages for easy debugging.

Automated Audit

Remix Compiler Warnings

It throws warnings by Solidity's compiler. If it encounters any errors the contract cannot be compiled and deployed.



It is recommended to add the following line at the beginning of contract code:

// SPDX-License-Identifier: MIT

Other than this no error was thrown by the compiler.

Remix Compiler Warnings

Securify is a tool that scans Ethereum smart contracts for critical security vulnerabilities. Securify statically analyzes the EVM code of the smart contract to infer important semantic information (including control-flow and data-flow facts) about the contract. The report provided by Securify can be accessed here:

https://securify.chainsecurity.com/report/38ee32e659b817d11bc5c40452f27-60c431210e368587eba0269104028eced4d

Securify detected no major vulnerability.

Contract Library

Contract-library contains the most complete, high-level decompiled representation of all Ethereum smart contracts, with security analysis applied to these in real time.

We performed analysis using contract Library on the mainnet address of the GGfinance contract: 0xa33529a0F240e5754BBFAD4786439365BD096dE3

Analysis summary can be accessed here:
https://contract-library.com/contracts/Ethereum/0xa33529a0f240e5754bbfad4786439365bd096de3

It did not return any issue during the analysis.

SmartCheck

Smartcheck is a tool for automated static analysis of Solidity source code for security vulnerabilities and best practices. SmartCheck translates Solidity source code into an XML-based intermediate representation and checks it against XPath patterns. Smartcheck shows significant improvements over existing alternatives in terms of false discovery rate (FDR) and false negative rate (FNR). It gave the following result for the GGfinance contract:
https://tool.smartdec.net/scan/a4d00099f4614c4bbf18ddbe5d92b7b5.

SmartCheck did not detect any high severity issue. All the considerable issues raised by SmartCheck are already covered in the Manual Audit section of this report.

Slither Tool Result

Slither is an open-source static analysis framework. This tool provides rich information about Ethereum smart contracts and has the critical properties. While Slither is built as a security-oriented static analysis framework, it is also used to enhance the user's understanding of smart contracts, assist in code reviews, and detect missing optimizations.

```
 GGfinance git:(master)  slither .
'npx truffle@5.1.39 compile --all' running (use --truffle-version truffle@x.x.x to use specific version)

Compiling your contracts...
===========================
> Compiling ./contracts/GGfinance.sol
> Compilation warnings encountered:

    /home/rails/work/audit/GGfinance/contracts/GGfinance.sol: Warning: SPDX license identifier not provided in source file. Before publishing, consider adding a comment containing "SPDX-License-Identifier: <SPDX-License>" to each source file. Use "SPDX-License-Identifier: UNLICENSED" for non-open-source code.
Please see https://spdx.org for more information.

> Artifacts written to /home/rails/work/audit/GGfinance/build/contracts
> Compiled successfully using:
   - solc: 0.7.0+commit.9e61f92b.Emscripten.clang


- Fetching solc version list from solc-bin. Attempt #1
- Fetching solc version list from solc-bin. Attempt #2
- Fetching solc version list from solc-bin. Attempt #3

INFO:Detectors:
Contract locking ether found in :
        Contract GGfinance (GGfinance.sol#64-82) has payable functions:
         - GGfinance.receive() (GGfinance.sol#77-79)
        But does not have a function to withdraw the ether
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#contracts-that-lock-ether
INFO:Detectors:
Pragma version^0.7.0 (GGfinance.sol#5) necessitates versions too recent to be trusted. Consider deploying with 0.5.11
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#incorrect-versions-of-solidity
INFO:Detectors:
Parameter Owned.changeOwner(address)._newOwner (GGfinance.sol#14) is not in mixedCase
Parameter ERC20.balanceOf(address)._owner (GGfinance.sol#34) is not in mixedCase
Parameter ERC20.transfer(address,uint256)._to (GGfinance.sol#36) is not in mixedCase
Parameter ERC20.transfer(address,uint256)._amount (GGfinance.sol#36) is not in mixedCase
Parameter ERC20.transferFrom(address,address,uint256)._from (GGfinance.sol#44) is not in mixedCase
Parameter ERC20.transferFrom(address,address,uint256)._to (GGfinance.sol#44) is not in mixedCase
Parameter ERC20.transferFrom(address,address,uint256)._amount (GGfinance.sol#44) is not in mixedCase
Parameter ERC20.approve(address,uint256)._spender (GGfinance.sol#53) is not in mixedCase
Parameter ERC20.approve(address,uint256)._amount (GGfinance.sol#53) is not in mixedCase
Parameter ERC20.allowance(address,address)._owner (GGfinance.sol#59) is not in mixedCase
Parameter ERC20.allowance(address,address)._spender (GGfinance.sol#59) is not in mixedCase
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#conformance-to-solidity-naming-conventions
INFO:Detectors:
GGfinance.constructor(address) (GGfinance.sol#67-75) uses literals with too many digits:
        - totalSupply = 2000000000000000000000 (GGfinance.sol#71)
```

```
GGfinance.constructor(address) (GGfinance.sol#67-75) uses literals with too many digits:
	- maxSupply = 2000000000000000000000 (GGfinance.sol#72)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#too-many-digits
INFO:Detectors:
changeOwner(address) should be declared external:
	- Owned.changeOwner(address) (GGfinance.sol#14-16)
acceptOwnership() should be declared external:
	- Owned.acceptOwnership() (GGfinance.sol#17-21)
balanceOf(address) should be declared external:
	- ERC20.balanceOf(address) (GGfinance.sol#34)
transfer(address,uint256) should be declared external:
	- ERC20.transfer(address,uint256) (GGfinance.sol#36-42)
transferFrom(address,address,uint256) should be declared external:
	- ERC20.transferFrom(address,address,uint256) (GGfinance.sol#44-51)
approve(address,uint256) should be declared external:
	- ERC20.approve(address,uint256) (GGfinance.sol#53-57)
allowance(address,address) should be declared external:
	- ERC20.allowance(address,address) (GGfinance.sol#59-61)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#public-function-that-could-be-declared-as-external
INFO:Slither:. analyzed (3 contracts with 46 detectors), 22 result(s) found
INFO:Slither:Use https://crytic.io/ to get access to additional detectors and Github integration
```

Slither raised one medium severity issue which has already been covered in the Manual report.

Disclaimer

Quillhash audit is not a security warranty, investment advice, or an endorsement of the GGfinance contract. Securing smart contracts is a multistep process, therefore running a bug bounty program as a complement to this audit is strongly recommended.

Summary

Use case of the smart contract is simple and the code is relatively small. Altogether, the code is written and demonstrates effective use of abstraction, separation of concerns, and modularity. But there are a number of issues/ vulnerabilities to be tackled in various severity levels, it is recommended to fix them before implementing a live version.

GG FINANCE

QuillAudits

448-A EnKay Square, Opposite Cyber Hub,
Gurugram, Harayana, India - 122016

audits.quillhash.com

hello@quillhash.com