SMART CONTRACT CODE REVIEW AND SECURITY ANALYSIS REPORT

Customer: JetFuel Team
Prepared on: 08/03/2021
Platform: Binance Smart Chain
Language: Solidity
Audit Type: Standard

audit@etherauthority.io

Table of contents

Document 4

Introduction 4

Quick Stats 5

Executive Summary 6

Code Quality 6

Documentation 7

Use of Dependencies 7

AS-IS overview 8

Severity Definitions 11

Audit Findings 11

Conclusion 14

Our Methodology 15

Disclaimers 17

THIS IS SECURITY AUDIT REPORT DOCUMENT AND WHICH MAY CONTAIN INFORMATION WHICH IS CONFIDENTIAL. WHICH INCLUDES ANY POTENTIAL VULNERABILITIES AND MALICIOUS CODES WHICH CAN BE USED TO EXPLOIT THE SOFTWARE. THIS MUST BE REFERRED INTERNALLY AND ONLY SHOULD BE MADE AVAILABLE TO PUBLIC AFTER ISSUES ARE RESOLVED.

Document

Name	Smart Contract Code Review and Security Analysis Report for GFORCE
Platform	Binance Smart Chain / Solidity
File name 1	GFORCE.sol
MD5 hash	FF7300EF3725878077C6350AFF545203
SHA256 hash	84A81AAF0535B7CD3D2FB1F9FA70590F837 71271FF03C74E7895BB5C87B97821

Introduction

We were contracted by the JetFuel team to perform the Security audit of the smart contracts code. The audit has been performed using manual analysis as well as using automated software tools. This report presents all the findings regarding the audit performed on 08/03/2021.

Audit type was Standard Audit. Which means one senior auditor performing an audit for 2 days. So, this audit is concluded based on standard audit scope. And because the use case scenarios are unlimited, it is encouraged to perform an Extensive audit (which is performed by 2 or more auditors for about a week time) to come to a more solid conclusion.

Quick Stats:

Main Category	Subcategory	Result
Contract Programming	Solidity version not specified	Passed
	Solidity version too old	Moderated
	Integer overflow/underflow	Moderated
	Function input parameters lack of check	Passed
	Function input parameters check bypass	Passed
	Function access control lacks management	Passed
	Critical operation lacks event log	Passed
	Human/contract checks bypass	Passed
	Random number generation/use vulnerability	N/A
	Fallback function misuse	Passed
	Race condition	Passed
	Logical vulnerability	Passed
	Other programming issues	Passed
Code Specification	Function visibility not explicitly declared	Passed
	Var. storage location not explicitly declared	Passed
	Use keywords/functions to be deprecated	Passed
	Other code specification issues	Passed
Gas Optimization	Assert() misuse	Passed
	High consumption 'for/while' loop	Moderated
	High consumption 'storage' storage	Passed
	"Out of Gas" Attack	Passed
Business Risk	The maximum limit for mintage not set	Passed
	"Short Address" Attack	Passed
	"Double Spend" Attack	Passed

Overall Audit Result: PASSED

Executive Summary

According to the **standard** audit assessment, Customer`s solidity smart contract is **well secured**. Again, it is recommended to perform an Extensive audit assessment to bring a more assured conclusion.

You are here

We used various tools like SmartDec, Mythril, Slither and Remix IDE. At the same time this finding is based on critical analysis of the menual audit.
All issues found during automated analysis were manually reviewed and applicable vulnerabilities are presented in the Audit overview section. General overview is presented in AS-IS section and all found issues can be found in the Audit overview section.

We found 0 high, 0 medium and 0 low and some very low level issues.

Code Quality

GFORCE protocol consists of one smart contract file. These smart contracts also contain Libraries, Smart contract inherits and Interfaces. These are compact and well written contracts.

The libraries in the GFORCE protocol are part of its logical algorithm. A library is a different type of smart contract that contains reusable code. Once deployed on the blockchain (only once), it is assigned a specific address and its properties / methods can be reused many times by other contracts in the GFORCE protocol.

The GFORCE team has **not** provided scenario and unit test scripts, which would help to determine the integrity of the code in an automated way.

Overall, some code parts are well commented, while rest are **not**. Commenting can provide rich documentation for functions, return variables and more. Use of Ethereum Natural Language Specification Format (NatSpec) for commenting is recommended.

Documentation

We were given a GFORCE contract in the form of a file. The hash of that file is mentioned above in the table.

As mentioned above, most code parts are **not** well commented. so anyone can not quickly understand the programming flow as well as complex code logic. Comments are very helpful in understanding the overall architecture of the protocol. It also provided a clear overview of the system components, including helpful details, like the lifetime of the background script.

Use of Dependencies

As per our observation, the libraries are used in this smart contract infrastructure that are based on well known industry standard open source projects. And even core code blocks are written well and systematically.

Apart from libraries, GFORCE smart contract depends on external smart contracts like PancakeV2Router, PancakeV2Factory, LpStakingPool, JetsStakingPool, etc. smart contracts.

AS-IS overview

GFORCE.sol contract overview

GFORCE is a BEP20 token smart contract. It has a refraction feature, which means 1% of every transaction is distributed to all the token holders. It also has in-built swapping functions. Following are the main components whose details are explicitly recorded.

(1) Libraries
- (a) SafeMath
- (b) Address

(2) Interfaces
- (a) IERC20
- (b) IPancakeV2Factory
- (c) IPancakeV2Pair
- (d) IPancakeV2Router01
- (e) IPancakeV2Router02

(3) Abstracts
- (a) Context

(4) External Contracts
- (a) Ownable
- (b) Balancer
- (c) Swaper

(5) Usages
- (a) SafeMath for uint256
- (b) Address for address

(6) Events
- (a) event FeeDecimalsUpdated(uint256 taxFeeDecimals);
- (b) event TaxFeeUpdated(uint256 taxFee);
- (c) event LockFeeUpdated(uint256 lockFee);
- (d) event MaxTxAmountUpdated(uint256 maxTxAmount);
- (e) event PoolAndPairTokenUpdated(address indexed poolAddress, address indexed pairTokenAddress);

(f) event StakingPoolUpdated(address indexed lpStakingPool,address indexed jetsStakingPool);
(g) event TradingEnabled();
(h) event SwapAndLiquifyEnabledUpdated(bool enabled);
(i) event SwapAndLiquify(address indexed pairTokenAddress, uint256 tokensSwapped, uint256 pairTokenReceived, uint256 tokensIntoLiqudity);
(j) event Rebalance(uint256 tokenBurnt);
(k) event MinTokensBeforeSwapUpdated(uint256 minTokensBeforeSwap);
(l) event AutoSwapCallerFeeUpdated(uint256 autoSwapCallerFee);

(7) Functions

Sl.	Function	Type	Observation	Conclusion	Score
1	constructor	write	Passed	No Issue	Passed
2	name	read	Passed	No Issue	Passed
3	symbol	read	Passed	No Issue	Passed
4	decimals	read	Passed	No Issue	Passed
5	totalSupply	read	Passed	No Issue	Passed
6	balanceOf	read	Passed	No Issue	Passed
7	transfer	write	Passed	No Issue	Passed
8	allowance	read	Passed	No Issue	Passed
9	approve	write	Passed	No Issue	Passed
10	transferFrom	write	Passed	No Issue	Passed
11	increaseAllowance	write	Passed	No Issue	Passed
12	decreaseAllowance	write	Passed	No Issue	Passed
13	isExcluded	read	Passed	No Issue	Passed
14	isWhiteListed	read	Passed	No Issue	Passed
15	materialize	write	Overflow happens initially	Overflow is resolved after first token transfer	Passed
16	reflectionFromToken	read	Passed	No Issue	Passed
	tokenFromReflection	read	Passed	No Issue	Passed
	excludeAccount	write	Passed	No Issue	Passed
	includeAccount	write	Infinite loop possibility	Owner must exclude 100 wallets or less	Passed
	includeWhiteList	write	Passed	No Issue	Passed
	excludeWhiteList	write	Passed	No Issue	Passed
	_approve	write	Passed	No Issue	Passed
	_transfer	write	Passed	No Issue	Passed
	swapAndLiquifyForEth	write	Passed	No Issue	Passed
	swapTokensForEth	write	Passed	No Issue	Passed
	addLiquidityForEth	write	Passed	No Issue	Passed

	swapAndLiquifyForTokens	write	Passed	No Issue	Passed
	addLiquidityForTokens	write	Passed	No Issue	Passed
	_transferStandard	write	Passed	No Issue	Passed
	_transferToExcluded	write	Passed	No Issue	Passed
	_transferFromExcluded	write	Passed	No Issue	Passed
	_transferBothExcluded	write	Passed	No Issue	Passed
	_reflectFee	write	Passed	No Issue	Passed
	_getValues	read	Passed	No Issue	Passed
	_getTValues	read	Passed	No Issue	Passed
	_getRValues	read	Passed	No Issue	Passed
	_getRate	read	Passed	No Issue	Passed
	_getCurrentSupply	read	Infinite loop possibility	Owner must exclude 100 wallets or less	**Passed**
	getCurrentPoolAddress	read	Passed	No Issue	Passed
	getCurrentPairTokenAddress	read	Passed	No Issue	Passed
	_setFeeDecimals	write	Passed	No Issue	Passed
	_setTaxFee	write	Passed	No Issue	Passed
	_setLockFee	write	Passed	No Issue	Passed
	_setMaxTxAmount	write	Passed	No Issue	Passed
	_setMinTokensBeforeSwap	write	Passed	No Issue	Passed
	_setAutoSwapCallerFee	write	Passed	No Issue	Passed
	updateSwapAndLiquifyEnabled	write	Passed	No Issue	Passed
	_updatePoolAndPairToken	write	Passed	No Issue	Passed
	_updateStakingPool	write	Passed	No Issue	Passed
	_enableTrading	write	Passed	No Issue	Passed

Severity Definitions

Risk Level	Description
Critical	Critical vulnerabilities are usually straightforward to exploit and can lead to tokens loss etc.
High	High-level vulnerabilities are difficult to exploit; however, they also have significant impact on smart contract execution, e.g. public access to crucial functions
Medium	Medium-level vulnerabilities are important to fix; however, they can't lead to tokens lose
Low	Low-level vulnerabilities are mostly related to outdated, unused etc. code snippets, that can't have significant impact on execution
Lowest / Code Style / Best Practice	Lowest-level vulnerabilities, code style violations and info statements can't affect smart contract execution and can be ignored.

Audit Findings

Critical

No critical severity vulnerabilities were found.

High

No high severity vulnerabilities were found.

Medium

No Medium severity vulnerabilities were found.

Low

No Low severity vulnerabilities were found.

Very Low

(1) Infinite loop possibility:

```
    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);
}
```

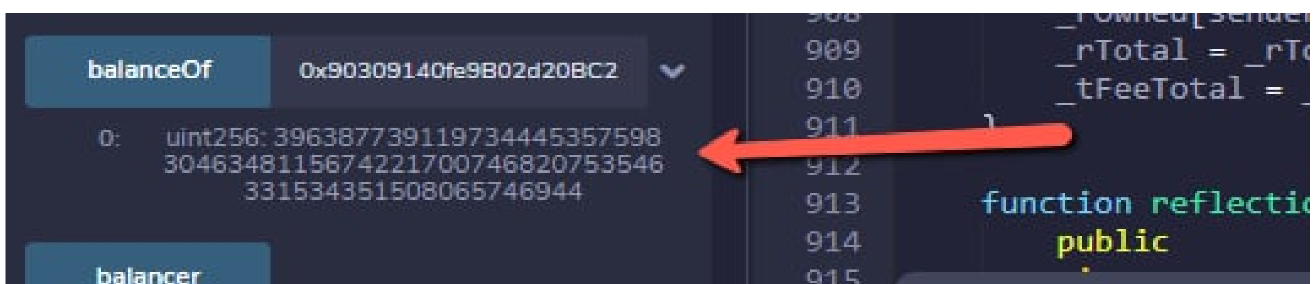
If the _excluded array length is increased, then it might hit the gas limit. Another function also would be affected with this is: includeAccount

Solution: please keep this excluded wallet as minimum as possible. Ideally under 100 wallets.

(2) Overflow possibility:

```
function materialize(uint256 tAmount) public {
    address sender = _msgSender();
    require(
        !_isExcluded[sender],
        "Gforce: Excluded addresses cannot call this function"
    );
    (uint256 rAmount, , , , ) = _getValues(tAmount);
    _rOwned[sender] = _rOwned[sender].sub(rAmount);
    _rTotal = _rTotal.sub(rAmount);
    _tFeeTotal = _tFeeTotal.add(tAmount);
}
```

When the smart contract is deployed, then calling this function with full token supply gives overflow.



This issue is resolved after its first token transfer. This is because its relevant variables would be updated and this is not happening again.

Solution: Please do some token transfers after contract deployment, before going to public launch.

(3) Ownership transfer function. It is good practice to implement acceptOwnership style to prevent the ownership sent to invalid address by human error. Code flow similar to below:

```
    function transferOwnership(address payable _newOwner) external onlyOwner {
        newOwner = _newOwner;
    }

    //this flow is to prevent transferring ownership to wrong wallet by mistake
    function acceptOwnership() external {
        require(msg.sender == newOwner);
        emit OwnershipTransferred(owner, newOwner);
        owner = newOwner;
        newOwner = payable(0);
    }
}
```

(4) User latest solidity version while contract deployment to prevent any compiler version level bugs.

Discussion:

(1) Overpowered functions: There are some functions which are authorised persons (excludeAccount, includeAccount, includeWhiteList, etc) only. And it would be troublesome if the private key of that owner wallet would be compromised.

(2) Approve of ERC20 standard: This can be used to front run. From the client side, only use this function to change the allowed amount to 0 or from 0 (wait till transaction is mined and approved). This should be done from the client side.

Conclusion

We were given contract files. And we have used all possible tests based on given objects as files. The contracts are written so systematic, that we did not find any major issues. **So it is good to go for production.**

Since possible test cases can be unlimited for such extensive smart contract protocol, so we provide no such guarantee of future outcomes. We have used all the latest static tools and manual observations to cover maximum possible test cases to scan everything.

Smart contracts within the scope were manually reviewed and analyzed with static analysis tools. Smart Contract's high level description of functionality was presented in As-is overview section of the report.

Audit report contains all found security vulnerabilities and other issues in the reviewed code.

Security state of the reviewed contract, based on standard audit procedure scope is "**Well Secured**".

Our Methodology

We like to work with a transparent process and make our reviews a collaborative effort. The goals of our security audits are to improve the quality of systems we review and aim for sufficient remediation to help protect users. The following is the methodology we use in our security audit process.

Manual Code Review:

In manually reviewing all of the code, we look for any potential issues with code logic, error handling, protocol and header parsing, cryptographic errors, and random number generators. We also watch for areas where more defensive programming could reduce the risk of future mistakes and speed up future audits. Although our primary focus is on the in-scope code, we examine dependency code and behavior when it is relevant to a particular line of investigation.

Vulnerability Analysis:

Our audit techniques included manual code analysis, user interface interaction, and whitebox penetration testing. We look at the project's web site to get a high level understanding of what functionality the software under review provides. We then meet with the developers to gain an appreciation of their vision of the software. We install and use the relevant software, exploring the user interactions and roles. While we do this, we brainstorm threat models and attack surfaces. We read design documentation, review other audit results, search for similar projects, examine source code dependencies, skim open issue tickets, and generally investigate details other than the implementation.

Documenting Results:

We follow a conservative, transparent process for analyzing potential security vulnerabilities and seeing them through successful remediation. Whenever a potential issue is discovered, we immediately create an Issue entry for it in this document, even though we have not yet verified the feasibility and impact of the issue. This process is conservative because we document our suspicions early even if they are later shown to not represent exploitable vulnerabilities. We generally follow a process of first documenting the suspicion with unresolved questions, then confirming the issue through code analysis, live experimentation, or automated tests. Code analysis is the most tentative, and we strive to provide test code, log captures, or screenshots demonstrating our confirmation. After this we analyze the feasibility of an attack in a live system.

Suggested Solutions:

We search for immediate mitigations that live deployments can take, and finally we suggest the requirements for remediation engineering for future releases. The mitigation and remediation recommendations should be scrutinized by the developers and deployment engineers, and successful mitigation and remediation is an ongoing collaborative process after we deliver our report, and before the details are made public.

Disclaimers

EtherAuthority.io Disclaimer

EtherAuthority team has analyzed this smart contract in accordance with the best industry practices at the date of this report, in relation to: cybersecurity vulnerabilities and issues in smart contract source code, the details of which are disclosed in this report, (Source Code); the Source Code compilation, deployment and functionality (performing the intended functions).

Due to the fact that the total number of test cases are unlimited, so the audit makes no statements or warranties on security of the code. It also cannot be considered as a sufficient assessment regarding the utility and safety of the code, bugfree status or any other statements of the contract. While we have done our best in conducting the analysis and producing this report, it is important to note that you should not rely on this report only. We also suggest to conduct a bug bounty program to confirm the high level of security of this smart contract.

Technical Disclaimer

Smart contracts are deployed and executed on blockchain platform. The platform, its programming language, and other software related to the smart contract can have their own vulnerabilities that can lead to hacks. Thus, the audit can't guarantee explicit security of the audited smart contracts.

EtherAuthority