FRENCHIE PROTOCOL SMART CONTRACT CODE REVIEW AND SECURITY ANALYSIS REPORT

Customer:	Frenchie Team (https://frenchie.tech)
Prepared on:	29/04/2021
Platform:	Binance Smart Chain
Language:	Solidity
Audit Type:	Standard

audit@etherauthority.io

Table of contents

Project file

Name	Smart Contract Code Review and Security Analysis Report for Frenchie Protocol
Platform	Binance Smart Chain / Solidity
File 1	Frenchie.sol
File 1 MD5 hash	249ED36D995BE56E0A14093B16BAFC70
Smart Contract Code URL	https://bscscan.com/address/0x13958e1eb63dfb8540eaf6ed7dcbbc1a60fd52af#code
File 2	frenchielpfarm.sol
File 2 MD5 hash	07AE4FC8FC91547DFDA4F7FB41425CBB
Smart Contract Code URL	https://github.com/FrenchieNetwork/contract/blob/main/contracts/frenchielpfarm.sol

Introduction

We were contracted by the Frenchie team to perform the Security audit of the Token and LP Farming 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 29/04/2021.

The Audit type was Standard Audit. Which means this audit is concluded based on Standard audit scope, which is one security engineer performing an audit procedure for 2 days. This document outlines all the findings as well as an AS-IS overview of the smart contract codes.

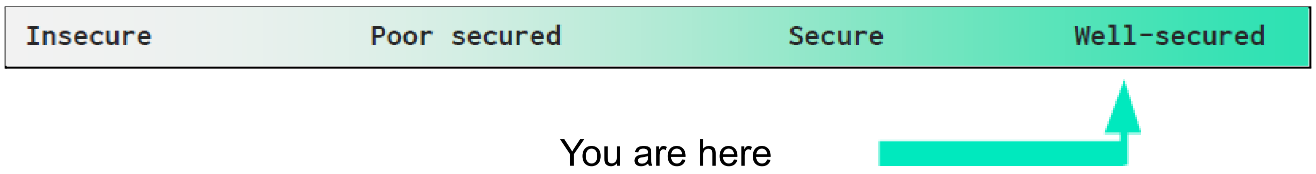
Quick Stats:

Main Category	Subcategory	Result
Contract Programming	Solidity version not specified	Passed
	Solidity version too old	Moderated
	Integer overflow/underflow	Passed
	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	Passed
	Fallback function misuse	Passed
	Race condition	Passed
	Logical vulnerability	Passed
	DoS possibility	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
	Tokenomics issues	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**. Since the use case scenarios of such smart contracts are unlimited, it is recommended to perform an **Intensive** audit to come to a more solid conclusion.



We used various tools like Mythril, Slither and Remix IDE. At the same time this finding is based on critical analysis of the manual 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 identified issues can be found in the Audit overview section.

We found 0 high, 0 medium and 1 low and some very low level issues.

These issues are fixed and/or acknowledged as non-problematic.

Code Quality

Frenchie smart contracts consist of 2 core solidity files. These smart contracts also contain Libraries, Smart contract inherits and Interfaces. These are compact and well written contracts.

The libraries in the Frenchie 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 Frenchie protocol.

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

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

Documentation

We were given Frenchie smart contracts code in the form of the files. The hash of those files and its web link are 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.

Another source of information was its official website frenchie.tech, which provided rich information about the project architecture and tokenomics.

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 their core code blocks are written well.

Apart from libraries, Frenchie LP farming smart contract interacts with another token contract as external smart contract calls.

AS-IS overview

Frenchie Token is a BEP20 standard token smart contract. And Frenchie LP Farm distributes the Tokens rewards based on staked LP to each user. Following are the main components of core smart contracts.

Frenchie.sol

(1) Inherits

(a) ERC20: Provides BEP20 token functions

(b) IERC20: An interface for the BEP20 standard

(2) Usages

(a) using SafeMath for uint256;

(3) Events

(a) event Transfer(address indexed from, address indexed to, uint256 value);

(b) event Approval(address indexed owner, address indexed spender, uint256 value);

(4) Functions

Sl.	Function	Type	Observation	Conclusion	Score
1	totalSupply	read	Passed	No Issue	Passed
2	balanceOf	read	Passed	No Issue	Passed
3	allowance	read	Passed	No Issue	Passed
4	approve	write	Passed	No Issue	Passed
5	transferFrom	write	Passed	No Issue	Passed
6	increaseAllowance	write	Passed	No Issue	Passed
7	decreaseAllowance	write	Passed	No Issue	Passed
8	_transfer	internal	Passed	No Issue	Passed
9	_mint	internal	Passed	No Issue	Passed
11	_burn	internal	Passed	No Issue	Passed

12	_approve	internal	Passed	No Issue	Passed
13	_burnFrom	internal	Not used anywhere	Remove it	Passed
14	constructor	write	Passed	No Issue	Passed
15	burn	write	Passed	No Issue	Passed
16	name	read	Passed	No Issue	Passed
17	symbol	read	Passed	No Issue	Passed
18	decimals	read	Passed	No Issue	Passed

Frenchielpfarm.sol

(1) Imports

(a) IERC20.sol

(b) SafeERC20.sol

(c) EnumerableSet.sol

(d) SafeMath.sol

(e) Ownable.sol

(2) Inherits

(a) Ownable: Provides ownership functions

(3) Usages

(a) using SafeMath for uint256;

(b) using SafeERC20 for IERC20;

(4) Events

(a) event Deposit(address indexed user, uint256 indexed pid, uint256 amount);

(b) event Withdraw(address indexed user, uint256 indexed pid, uint256 amount);

(c) event EmergencyWithdraw(address indexed user, uint256 indexed pid, uint256 amount);

(5) Functions

Sl.	Function	Type	Observation	Conclusion	Score
1	constructor	write	start and end block values are the same	No Issue, as that is updated in fund method	Passed with client consent
2	poolLength	read	Passed	No Issue	Passed
3	fund	write	Passed	No Issue	Passed
4	add	write	Passed	No Issue	Passed
5	set	write	Passed	No Issue	Passed
6	deposited	read	Passed	No Issue	Passed
7	pending	read	Passed	No Issue	Passed
8	totalPending	read	Passed	No Issue	Passed
9	massUpdatePools	write	Infinite loop possibility	Keep array length limited	Passed with consent
11	updatePool	write	Passed	No Issue	Passed
12	deposit	write	Passed	No Issue	Passed
13	withdraw	write	Passed	No Issue	Passed
14	emergencyWithdraw	write	Passed	No Issue	Passed
15	erc20Transfer	internal	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

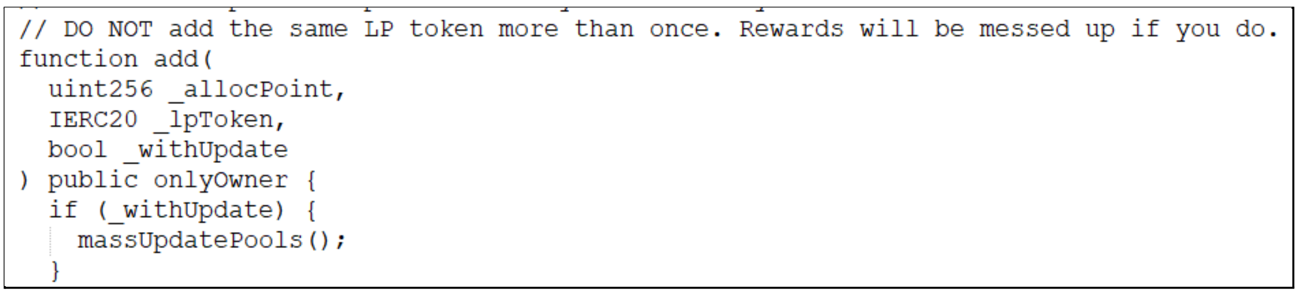
No high severity vulnerabilities were found.

Medium

No medium severity vulnerabilities were found.

Low

(1) Input validation missing in frenchielpfarm contract



```
// DO NOT add the same LP token more than once. Rewards will be messed up if you do.
function add(
  uint256 _allocPoint,
  IERC20 _lpToken,
  bool _withUpdate
) public onlyOwner {
  if (_withUpdate) {
    massUpdatePools();
  }
```

It is recommended to add an input validation to prevent adding the same LP tokens again.

Resolution: This issue is fixed in the patched version of the smart contract at the commit: b70ef3a

Very Low / Best practices

(1) Infinite loop possibility in frenchielpfarm contract

```
// Update reward variables for all pools. Be careful of gas spending!
function massUpdatePools() public {
  uint256 length = poolInfo.length;
  for (uint256 pid = 0; pid < length; ++pid) {
    updatePool(pid);
  }
}
```

This only creates problems when pool size becomes too large. So, It is recommended to input some starting and end index of the array. so this process can be done in batches.

Resolution: We got confirmation from the Frenchie team that pool size will never be too large in real use case scenario. Thus, this will never create any problems.

(2) Use the latest solidity version. Although this does not raise major issues, it's better to use the latest version to take advantage of the latest solidity compiler.

(3) All functions which are not called internally, must be declared as external. It is more efficient as sometimes it saves some gas.
https://ethereum.stackexchange.com/questions/19380/external-vs-public-best-practices

Conclusion

We were given a contract code. And we have used all possible tests based on given objects as files. The contracts are written so systematically, that we did not find any major issues. Some issues were observed, which were rectified by the Frenchie team. **So it is good to go for the production.**

Since possible test cases can be unlimited for such extensive smart contract protocol, 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 contracts, 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.