BonFi

Smart Contract Audit

Final Report

April 12, 2021

IMMUNE BYTES

Introduction

1. About Bon Finance

BonFi platform will enable the crypto community to seize the advantages of Artificial Intelligence, Data Oracles, and Quantitative Analysis to build a more beneficial and decentralized open finance liquidity pool solution.

BonFi is complemented by the AI-powered BonVest, a liquidity pool service helping users optimize their yield farming portfolios. It expands the DeFi financial product offering using open source technology, decentralized governance, smart contract staking, and data oracles to optimize liquidity pool mining. The BNF governance token functions as an entry piece for the community to become members of the BonFi ecosystem. It concurrently functions as a gateway to directly influence the future direction of the organization.

Visit https://www.bon.finance/ to know more about,

2. About ImmuneBytes

ImmuneBytes is a security start-up to provide professional services in the blockchain space. The team has hands-on experience in conducting smart contract audits, penetration testing, and security consulting. ImmuneBytes's security auditors have worked on various A-league projects and have a great understanding of DeFi projects like AAVE, Compound, 0x Protocol, Uniswap, dydx.

The team has been able to secure 15+ blockchain projects by providing security services on different frameworks. ImmuneBytes team helps start-up with a detailed analysis of the system ensuring security and managing the overall project.

Visit http://immunebytes.com/ to know more about the services.

Documentation Details

Bon Finance team has provided documentation for the purpose of conducting the audit. The documents are:

1. https://bnf.gitbook.io/gitbook/whitepaper/bonfi-whitepaper

IMMUNE BYTES

Audit Process & Methodology

ImmuneBytes team has performed thorough testing of the project starting with analyzing 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 in order to find any potential issues like Signature Replay Attacks, Unchecked External Calls, External Contract Referencing, Variable Shadowing, Race conditions, Transaction-ordering dependence, timestamp dependence, DoS attacks, and others.

In the Unit testing phase, we run unit tests written by the developer in order to verify the functions work as intended. In Automated Testing, we tested the Smart Contract with our in-house developed tools to identify vulnerabilities and security flaws.

The code was audited by a team of independent auditors which includes -

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

Audit Details

- Project Name: Bon Finance
- Languages: Solidity(Smart contract)
- Github commit hash for audit: b1b3e449096f97cf81fb0a5c3c64dff4b343236e
- Etherscan Code (Testnet):
    - Legendary_v8: 0x12678d430298dA2035F528a333EaB006568D2795
- Platforms and Tools: Remix IDE, Truffle, Truffle Team, Ganache, Solhint, VScode, 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 into the following three categories:

1. Security: Identifying security-related issues within each contract and within the system of contracts.
2. Sound Architecture: Evaluation of the architecture of this system through the lens of established smart contract best practices and general software best practices.
3. Code Correctness and Quality: A full review of the contract source code. The primary areas of focus include:
    a. Correctness
    b. Readability
    c. Sections of code with high complexity
    d. Quantity and quality of test coverage

Security Level References

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

High severity issues will bring problems and should be fixed.

Medium severity issues could potentially bring problems and should eventually be fixed.

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

Issues	High	Medium	Low
Open	-	-	-
Closed	-	-	2

Audit Categories and Results:

No.	Categories	Subitems	Results
1	Coding Conventions	ERC20 Token Standards	Pass
		Compiler Version Security	Present
		Visibility Specifiers	Pass
		Gas Consumption	Present
		SafeMath Features	Pass
		Fallback Usage	Pass
		tx.origin Usage	Pass
		Deprecated Items	Pass
		Redundant Code	Pass
		Overriding Variables	Pass
2	Function Call Audit	Authorization of Function Call	Pass
		Low-level Function (call/delegate call) Security	Pass
		Returned Value Security	Pass
		selfdestruct Function Security	Pass
3	Business Security	Access Control of Owner	Pass
		Business Logics	Pass
		Business Implementations	Pass
4	Integer Overflow/underflow	-	Pass
5	Reentrancy	-	Pass
6	Exceptional Reachable state	-	Pass

7	Transaction-Ordering Dependence	-	Pass
8	Block Properties Dependence	-	Present
9	Pseudo-random Number Generator (PRNG)	-	Pass
10	DoS (Denial of Service)	-	Pass
11	Token Vesting Implementation	-	N/A
12	Fake Deposit	-	Pass
13	Event security	-	Pass

Notes

Compiler version security: The solc frequently releases new compiler versions. Using an old version prevents access to new Solidity security checks. There is a list of recommended versions under low-level security issues in this report which are considered secure and can be used to deploy the contract.

Gas Consumption: The contract uses for loop in _calculate() function. Due to the block gas limit, transactions can only consume a certain amount of gas. Either explicitly or just due to normal operation, the number of iterations in a loop can grow beyond the block gas limit which can cause the complete contract to be stalled at a certain point. This may not apply to view functions that are only executed to read data from the blockchain. Still, such functions may be called by other contracts as part of on-chain operations and stall those. Please be explicit about such cases in the documentation of your contracts.

Block Properties Dependence: The contract uses block.timestamp at various places. Be aware that the timestamp of the block can be manipulated by the miner, and all direct and indirect uses of the timestamp should be considered. Block numbers and average block time can be used to estimate time, but this is not future proof as block times may change (such as the changes expected during Casper).

Low severity issues

1. The compiler version for the Legendary contract is ***^0.5.0.***
   It is recommended to fix the compiler version to avoid confusion on the solidity version of the file while deploying the contract.

   Deploy with any of the following Solidity versions:
   - 0.5.11 - 0.5.13,
   - 0.5.15 - 0.5.17,
   - 0.6.8,
   - 0.6.10 - 0.6.11. Use a simple pragma version that allows any of these versions. Consider using the latest version of Solidity for testing.

   Amended (April 10th 2021): Issue was fixed by the BonFi team.

2. The contract is failing with the following compiler errors:

```
> Compiling ./contracts/Legendary_v8.sol
    DocstringParsingError: Documented parameter "name_," not found
in the parameter list of the function.
    , DocstringParsingError: Documented parameter "'from'" not found
in the parameter list of the function.
    Compilation failed. See above.
    Truffle v5.3.0 (core: 5.3.0)
    Node v13.14.0
```

   The natspecs should be updated as suggested below:

```
For constructor()   /**
    *   @param
    *   name_ name of the contract
    *   tokenAddress_ contract address of the token
    *   rate_ rate multiplied by 100
    *   lockduration_ duration in days
    *   rookieAddress_ address of the rookie staking contract
    *   professionalAddress_ address of the professional staking
contract
    */
```

   Amended (April 10th 2021): Issue was fixed by the BonFi team.

Recommendations

1. Boolean constants can be used directly and do not need to be compared to true or false. It is recommended to remove the equality to boolean constant.

```
require(hasStaked[from] == false, "Already Staked"); (Line# 351)
require(deposits[from].paid == false, "Already paid out");
(Line#392)
require(deposits[from].paid == false, "Already paid out");
(Line#424)
Can be updated to:
require(!hasStaked[from], "Already Staked");
require(!deposits[from].paid, "Already paid out");
require(!deposits[from].paid, "Already paid out");
```

   Amended (April 10th 2021): Issue was fixed by the BonFi team.

2. A function with a ***public*** visibility modifier that is not called internally should be set to ***external*** visibility to increase code readability. Moreover, in many cases, functions with ***external*** visibility modifiers spend less gas compared to functions with ***public*** visibility modifiers. Following functions can be declared external:
   - ***calculate(address) (Legendary_v8.sol#452-454)***

   Amended (April 10th 2021): Issue was fixed by the BonFi team.

Automated Audit

Solhint Linting Violations

Solhint is an open-source project for linting solidity code, providing both security and style guide validations. It integrates seamlessly into most mainstream IDEs. We used Solhint as a plugin within our VScode for this analysis. Multiple Linting violations were detected by Solhint, it is recommended to use Solhint's npm package to lint the contract.

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 the contract Library on the Kovan address of the Legendary_v8 contract used during manual testing:

- Legendary_v8: 0x12678d430298dA2035F528a333EaB006568D2795

It raises no major concern for the contracts.

SmartCheck

Smart check 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 Legendary_v8 contract.

```
contracts/Legendary_v8.sol
jdeploy-bundle/smartcheck-2.0-jar-with-dependencies.jar!/solidity-rules.xmlline 106:57 extraneous input 'payable'
expecting {',', ')'}
ruleId: SOLIDITY_ADDRESS_HARDCODED
patternId: a91b18
severity: 1
line: 131
column: 8
content: _owner=address(0)

ruleId: SOLIDITY_EXTRA_GAS_IN_LOOPS
patternId: d3j11j
severity: 1
line: 474
column: 8
```

```
content:
for(uint64i=userIndex;i<index;i++){if(endTime<rates[i+1].timeStamp){break;}else{time=rates[i+1].timeStamp.sub(depositTime);interest=amount.mul(rates[i].newInterestRate).mul(time).div(_lockduration.mul(10000));amount+=interest;depositTime=rates[i+1].timeStamp;userIndex++;}}

ruleId: SOLIDITY_OVERPOWERED_ROLE
patternId: j83hf7
severity: 2
line: 255
column: 4
content: functionsetRate(uint64rate_)externalonlyOwner{require(rate_!=0,"Zero interest
rate");rate=rate_;index++;rates[index]=Rates(rate_,block.timestamp);}

ruleId: SOLIDITY_OVERPOWERED_ROLE
patternId: j83hf7
severity: 2
line: 280
column: 4
content:
functionsetEligibilityAmount(uint256eligibilityAmount_)externalonlyOwner{eligibilityAmount=eligibilityAmount_;}

ruleId: SOLIDITY_PRAGMAS_VERSION
patternId: 23fc32
severity: 1
line: 1
column: 16
content: ^

ruleId: SOLIDITY_PRAGMAS_VERSION
patternId: 23fc32
severity: 1
line: 40
column: 16
content: ^

ruleId: SOLIDITY_PRAGMAS_VERSION
patternId: 23fc32
severity: 1
line: 82
column: 16
content: ^

ruleId: SOLIDITY_PRAGMAS_VERSION
patternId: 23fc32
severity: 1
line: 101
column: 16
```

```
content: ^

ruleId: SOLIDITY_PRAGMAS_VERSION
patternId: 23fc32
severity: 1
line: 152
column: 16
content: ^

ruleId: SOLIDITY_PRIVATE_MODIFIER_DONT_HIDE_DATA
patternId: 5616b2
severity: 1
line: 104
column: 12
content: private

ruleId: SOLIDITY_PRIVATE_MODIFIER_DONT_HIDE_DATA
patternId: 5616b2
severity: 1
line: 177
column: 33
content: private

ruleId: SOLIDITY_PRIVATE_MODIFIER_DONT_HIDE_DATA
patternId: 5616b2
severity: 1
line: 179
column: 29
content: private

ruleId: SOLIDITY_SAFEMATH
patternId: 837cac
severity: 1
line: 155
column: 4
content: usingSafeMathforuint256;

ruleId: SOLIDITY_SHOULD_RETURN_STRUCT
patternId: 83hf3l
severity: 1
line: 317
column: 16
content: (uint256,uint256,uint256,uint256,bool)

SOLIDITY_SAFEMATH :1
SOLIDITY_OVERPOWERED_ROLE :2
SOLIDITY_PRAGMAS_VERSION :5
```

```
SOLIDITY_PRIVATE_MODIFIER_DONT_HIDE_DATA :3
SOLIDITY_EXTRA_GAS_IN_LOOPS :1
SOLIDITY_ADDRESS_HARDCODED :1
SOLIDITY_SHOULD_RETURN_STRUCT :1
```

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

Slither, an open-source static analysis framework. This tool provides rich information about Ethereum smart contracts and has 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.
The concerns slither raises have already been covered in the manual audit section.

```
INFO:Detectors:
Legendary_v8.emergencyWithdraw() (Legendary_v8.sol#417-427) uses a dangerous strict equality:
	- require(bool,string)(deposits[from].paid == false,Already paid out) (Legendary_v8.sol#424)
Legendary_v8.withdraw() (Legendary_v8.sol#385-394) uses a dangerous strict equality:
	- require(bool,string)(deposits[from].paid == false,Already paid out) (Legendary_v8.sol#392)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#dangerous-strict-equalities
INFO:Detectors:
Legendary_v8._stake(address,uint256).stakerEligibility (Legendary_v8.sol#361) is a local variable never initialized
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#uninitialized-local-variables
INFO:Detectors:
Legendary_v8.addReward(uint256) (Legendary_v8.sol#292-307) should emit an event for:
	- rewardBalance = rewardBalance.add(rewardAmount) (Legendary_v8.sol#305)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#missing-events-arithmetic
INFO:Detectors:
Reentrancy in Legendary_v8._stake(address,uint256) (Legendary_v8.sol#355-380):
	External calls:
	- ! _payMe(from,amount) (Legendary_v8.sol#356)
		- ERC20Interface.transferFrom(allower,receiver,amount) (Legendary_v8.sol#515)
	State variables written after the call(s):
	- deposits[from] =
Deposits(amount,block.timestamp,block.timestamp.add((lockDuration.mul(86400))),index,false,stakerEligibility)
(Legendary_v8.sol#366-373)
	- hasStaked[from] = true (Legendary_v8.sol#360)
	- stakedBalance = stakedBalance.add(amount) (Legendary_v8.sol#377)
	- stakedTotal = stakedTotal.add(amount) (Legendary_v8.sol#378)
Reentrancy in Legendary_v8.addReward(uint256) (Legendary_v8.sol#292-307):
	External calls:
	- ! _payMe(from,rewardAmount) (Legendary_v8.sol#300)
		- ERC20Interface.transferFrom(allower,receiver,amount) (Legendary_v8.sol#515)
	State variables written after the call(s):
	- rewardBalance = rewardBalance.add(rewardAmount) (Legendary_v8.sol#305)
	- totalReward = totalReward.add(rewardAmount) (Legendary_v8.sol#304)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#reentrancy-vulnerabilities-2
```

```
INFO:Detectors:
Reentrancy in Legendary_v8._emergencyWithdraw(address) (Legendary_v8.sol#429-443):
	External calls:
	- principalPaid = _payDirect(from,amount) (Legendary_v8.sol#438)
		- ERC20Interface.transfer(to,amount) (Legendary_v8.sol#520)
	Event emitted after the call(s):
	- PaidOut(tokenAddress,from,amount,0) (Legendary_v8.sol#440)
Reentrancy in Legendary_v8._stake(address,uint256) (Legendary_v8.sol#355-380):
	External calls:
	- ! _payMe(from,amount) (Legendary_v8.sol#356)
		- ERC20Interface.transferFrom(allower,receiver,amount) (Legendary_v8.sol#515)
	Event emitted after the call(s):
	- Staked(tokenAddress,from,amount) (Legendary_v8.sol#375)
Reentrancy in Legendary_v8._withdraw(address) (Legendary_v8.sol#396-415):
	External calls:
	- _payDirect(from,payOut) (Legendary_v8.sol#410)
		- ERC20Interface.transfer(to,amount) (Legendary_v8.sol#520)
	Event emitted after the call(s):
	- PaidOut(tokenAddress,from,amount,reward) (Legendary_v8.sol#411)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#reentrancy-vulnerabilities-3
INFO:Detectors:
Legendary_v8.withdraw() (Legendary_v8.sol#385-394) uses timestamp for comparisons
	Dangerous comparisons:
	- require(bool,string)(block.timestamp >= deposits[from].endTime,Requesting before lock time)
(Legendary_v8.sol#388-391)
	- require(bool,string)(deposits[from].paid == false,Already paid out) (Legendary_v8.sol#392)
Legendary_v8._withdraw(address) (Legendary_v8.sol#396-415) uses timestamp for comparisons
	Dangerous comparisons:
	- require(bool,string)(reward <= rewardBalance,Not enough rewards) (Legendary_v8.sol#400)
Legendary_v8.emergencyWithdraw() (Legendary_v8.sol#417-427) uses timestamp for comparisons
	Dangerous comparisons:
	- require(bool,string)(block.timestamp >= deposits[from].endTime,Requesting before lock time)
(Legendary_v8.sol#420-423)
	- require(bool,string)(deposits[from].paid == false,Already paid out) (Legendary_v8.sol#424)
Legendary_v8._emergencyWithdraw(address) (Legendary_v8.sol#429-443) uses timestamp for comparisons
	Dangerous comparisons:
	- require(bool,string)(principalPaid,Error paying) (Legendary_v8.sol#439)
Legendary_v8._calculate(address) (Legendary_v8.sol#456-503) uses timestamp for comparisons
	Dangerous comparisons:
	- i < index (Legendary_v8.sol#474)
	- endTime < rates[i + 1].timeStamp (Legendary_v8.sol#476)
	- depositTime < endTime (Legendary_v8.sol#490)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#block-timestamp
INFO:Detectors:
Legendary_v8.stake(uint256) (Legendary_v8.sol#343-353) compares to a boolean constant:
	-require(bool,string)(hasStaked[from] == false,Already Staked) (Legendary_v8.sol#351)
Legendary_v8.withdraw() (Legendary_v8.sol#385-394) compares to a boolean constant:
	-require(bool,string)(hasStaked[from] == true,No stakes found for user) (Legendary_v8.sol#387)
Legendary_v8.withdraw() (Legendary_v8.sol#385-394) compares to a boolean constant:
	-require(bool,string)(deposits[from].paid == false,Already paid out) (Legendary_v8.sol#392)
Legendary_v8.emergencyWithdraw() (Legendary_v8.sol#417-427) compares to a boolean constant:
	-require(bool,string)(deposits[from].paid == false,Already paid out) (Legendary_v8.sol#424)
Legendary_v8.emergencyWithdraw() (Legendary_v8.sol#417-427) compares to a boolean constant:
	-require(bool,string)(hasStaked[from] == true,No stakes found for user) (Legendary_v8.sol#419)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#boolean-equality
INFO:Detectors:
Pragma version^0.5.0 (Legendary_v8.sol#1) allows old versions
```

```
Pragma version^0.5.0 (Legendary_v8.sol#40) allows old versions
Pragma version^0.5.0 (Legendary_v8.sol#82) allows old versions
Pragma version^0.5.0 (Legendary_v8.sol#101) allows old versions
Pragma version^0.5.0 (Legendary_v8.sol#152) allows old versions
solc-0.5.0 is not recommended for deployment
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#incorrect-versions-of-solidity
INFO:Detectors:
Legendary_v8 (Legendary_v8.sol#154-546) should inherit from IRookieProfessionalCheck
(Legendary_v8.sol#148-150)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#missing-inheritance
INFO:Detectors:
Contract Legendary_v8 (Legendary_v8.sol#154-546) is not in CapWords
Variable Legendary_v8.ERC20Interface (Legendary_v8.sol#194) is not in mixedCase
Variable Legendary_v8.RookieInstance (Legendary_v8.sol#195) is not in mixedCase
Variable Legendary_v8.ProfessionalInstance (Legendary_v8.sol#196) is not in mixedCase
Modifier Legendary_v8._hasAllowance(address,uint256) (Legendary_v8.sol#523-529) is not in mixedCase
Modifier Legendary_v8._isEligible(address) (Legendary_v8.sol#535-545) is not in mixedCase
Reference:
https://github.com/crytic/slither/wiki/Detector-Documentation#conformance-to-solidity-naming-conventions
INFO:Detectors:
Redundant expression "this (Legendary_v8.sol#94)" inContext (Legendary_v8.sol#84-97)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#redundant-statements
INFO:Detectors:
Variable Legendary_v8._calculate(address)._lockduration (Legendary_v8.sol#473) is too similar to
Legendary_v8.constructor(string,address,uint64,uint256,address,address).lockDuration_ (Legendary_v8.sol#230)
Variable Legendary_v8._calculate(address)._lockduration (Legendary_v8.sol#473) is too similar to
Legendary_v8.changeLockDuration(uint256).lockduration_ (Legendary_v8.sol#267)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#variable-names-are-too-similar
INFO:Detectors:
owner() should be declared external:
        - Ownable.owner() (Legendary_v8.sol#116-118)
renounceOwnership() should be declared external:
        - Ownable.renounceOwnership() (Legendary_v8.sol#129-132)
transferOwnership(address) should be declared external:
        - Ownable.transferOwnership(address) (Legendary_v8.sol#134-136)
calculate(address) should be declared external:
        - Legendary_v8.calculate(address) (Legendary_v8.sol#452-454)
Reference:
https://github.com/crytic/slither/wiki/Detector-Documentation#public-function-that-could-be-declared-external
INFO:Slither:. analyzed (6 contracts with 72 detectors), 39 result(s) found
INFO:Slither:Use https://crytic.io/ to get access to additional detectors and Github integration
```

Concluding Remarks

While conducting the audits of the Bon Finance smart contract, it was observed that the contract contained only Low severity issues, along with several areas of recommendations.

Our auditors suggest that Low severity issues should be resolved by Bon Finance developers. Resolving the areas of recommendations are up to the team's discretion. The recommendations given will improve the operations of the smart contract.

Note: BonFi team has fixed the issues based on the auditor's recommendation.

Disclaimer

ImmuneBytes's audit does not provide a security or correctness guarantee of the audited smart contract. Securing smart contracts is a multistep process, therefore running a bug bounty program as a complement to this audit is strongly recommended.

Our team does not endorse the Bon Finance platform or its product neither this audit is investment advice.
Notes:

- Please make sure contracts deployed on the mainnet are the ones audited.
- Check for the code refactor by the team on critical issues.

ImmuneBytes Pvt Ltd.