SPORE Engineering
Smart Contract Audit Report

IMMUNE BYTES

Audits

March 24, 2021

This 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.

Introduction

1. About SPORE Engineering

SPORE Engineering Blockchain Factory focused on closing the gap between traditional fintech instruments and industry 4.0 powered by Ethereum smart contracts on Binance Smart Chain Network.

The SPORE Token is based on the BEP-20, written in Solidity, which is the native language of The Ethereum Smart Contracts. The SPORE Token uses only audited contracts provided by OpenZeppelin with no third-party libraries implications, it's backed and paired with BNB cryptocurrency and can be traded on Pancakeswap Protocol.

Visit https://spore.engineering/ to know more.

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

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

1. README.md

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: Spore Engineering
- Languages: Solidity(Smart contract), Javascript(Unit Testing)
- Github commits for audit:
  - https://github.com/spore-engineering/BSC-Contracts-Mainnet/blob/main/SporeToken.sol
  - https://github.com/spore-engineering/Spore-Stake-Farming-Rewards/blob/main/SporeStake.sol
  - https://github.com/spore-engineering/Spore-Stake-Farming-Rewards/blob/main/LiquidityFarming.sol
- Contract Deployment on the test network
  - SporeToken: 0xcebfd289273ebd1f2a4b594070671f284f737db2
  - SporeStake: 0x2a3138f6e436de36b2792d5aa1e8b695e5878830
  - Liquidity Farming: 0xcf651c746899f35fb96d900301c61535a8b3a6ac

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:

Admin/Owner Privileges can be misused either intentionally or unintentionally.

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	Admin/Owner Privileges	High	Medium	Low
Open	4	-	-	10
Closed	-	-	-	-

Admin/Owner Privileges

The **admin/owner** of **Spore** smart contracts has various privileges over the smart contracts. These privileges can be misused either intentionally or unintentionally (in case the admin's private key gets hacked). We assume that these extra rights will always be used appropriately. Some of these admin rights are listed below.

1. **In the SporeToken contract, the MINTER address can mint and burn any amount of Spore tokens.**
   The **SporeToken** contract contains a **mint**() function by which the **MINTER** account can mint any number of Spore tokens to any ethereum address. The contract also contains a **burn**() function by which **MINTER** can burn any number of Spore tokens from any ethereum account (until the account's balance becomes zero).

2. **MINTER role can be given to any address/account by the deployer (Default admin) of the SporeToken contract.**
   The deployer of the **SporeToken** contract possesses the **Default Admin role** for the contract. This role has the right to give a **MINTER** role to any ethereum address. Please note that there can be more than one **MINTER** for the SporeToken contract.

3. **Admin has the right to change the reward rate for SporeStake and LiquidityFarming contracts.**
   The Default Admin of **SporeStake and LiquidityFarming** contracts have the right to change the **_blockReward** using the **changeInterestRatePerBlock()** function. Any **unit256** value can be set as the **_blockReward**.

4. **Admin has the right to pause and unpause the working of SporeStake and LiquidityFarming contracts.**
   The Default Admin of **SporeStake and LiquidityFarming** contracts have the right to pause and unpause the working of smart contracts anytime. When **paused**, users won't be able to use the staking, unstaking, and claiming functionality of the smart contract.

   Recommendation:
   Consider hardcoding predefined ranges or validations for input variables in privileged access functions. Also, consider adding some governance for admin rights for smart contracts or use a multi-sig wallet as admin/owner address.

Low severity issues

1. **MINTER_ROLE should not be used for burning tokens.**
   In the **burn**() function of **SporeToken** contract in **SporeToken.sol**, **SporeLoot** contract in **SporeStake.sol** and **SporeToken** contract in **LiquidityFarming.sol** MINTER role is allowed to burn the tokens which are not correct as per the naming conventions. The MINTER role should only be used to mint new tokens and a BURNER role should be used for the burning of tokens.

   Recommendation:
   Consider adding a BURNER role in the contracts and use that role to burn tokens. Or consider renaming the MINTER role as MINTER_BURNER_ROLE.

2. **In the burn() function parameter *'to'* should be renamed as *'from'*.**
   The **burn**() function of **SporeToken** contract in **SporeToken.sol**, **SporeLoot** contract in **SporeStake.sol** and **SporeToken** contract in **LiquidityFarming.sol** takes an address variable "**to"** from which the specified amount of tokens are burned. This variable should be renamed as "**from"** which will make more accurate sense as per the function implementation.

   Recommendation:
   Consider renaming the input parameter.

3. **Incorrect revert reason in burn() function.**
   The **burn**() function of **SporeToken** contract in **SporeToken.sol, SporeLoot** contract in **SporeStake.sol,** and **SporeToken** contract in **LiquidityFarming.sol** contains a **require** statement to check the role of the caller. However, they revert reason for this **require** statement is incorrect as it mentions the word **mint** instead of **burn.** The revert reasons are a part of the contract's bytecode so they must always be precise and correct.

   Recommendation:
   Consider changing the word **mint** with **burn**.

4. **Arithmetic addition should be done before substraction.**
   In the **claim**() function of **SporeStake** and **LiquidityFarming** contracts the **_pending** amount is calculated as follows:

```
uint256 _pending =
_amount.sub(staker.rewardDebt).add(staker.rewardPending);
```

The above statement performs a SafeMath subtraction before addition which may cause issues in some cases and result in SafeMath underflow revert. Since here we are only dealing with positive integers, it is always recommended to perform addition before subtraction.

Recommendation:
Consider performing arithmetic addition before subtraction in the above-mentioned statement.

5. **No external function present to update the *accAmountPerShare* and *lastRewardBlock* variables.**
In the **SporeStake** and **LiquidityFarming** contracts an **update** modifier is defined which runs before every staking, unstaking, and claiming transaction and updates the ***accAmountPerShare*** and ***lastRewardBlock*** variables. However, no external function is defined in the contracts to just recalculate and update these variables as per the current block's timestamp.

Recommendation:
Consider implementing a function which only recalculates and updates the **accAmountPerShare** and **lastRewardBlock** variables.

6. **Contract bytecode size can be reduced by using Ownable contract instead of AccessControl.**
The **SporeStake** and **LiquidityFarming** contracts inherit the **AccessControl** contract to implement some privileged access functions. **AccessControl** contract is used to provide different types and levels of permissions to different ethereum accounts. Since the **SporeStake** and **LiquidityFarming** contracts only need a **DEFAULT_ADMIN_ROLE**, the same functionality can be obtained by using **OpenZeppelin's Ownable** contract. This will reduce the bytecode size of the contract and will cost less gas in contract deployment.

Recommendation:
Consider using the **Ownable** contract instead of **AccessControl**.

7. **Spore token transfer should be performed at last in *unstake*() function.**
In the **unstake**() function of **SporeStake** and **LiquidityFarming** contracts, the transfer of Spore tokens is implemented very initially in the function. It is always recommended to

perform the external calls and transfer of funds after the contract's internal ledger is updated to avoid reentrancy and other vulnerabilities. Since the contract only performs external calls to trusted contracts there are negligible chances of reentrancy attack. However, it is still recommended to perform the transfer of tokens at last.

Recommendation:
Consider performing the Spore token transfer after the contract's internal state is updated.

8. **Revert reasons not provided in some statements.**
   In the **stake**() and **unstake**() functions of **SporeStake** and **LiquidityFarming** contracts, some **require** statements are implemented to perform the necessary validations and sanity checks. However, no revert reasons are provided in those **require** statements. Revert reasons are part of the contract's bytecode and become useful during debugging of transactions.

   Recommendation:
   Consider providing revert reasons in the above-mentioned statements.

9. **Same calculations are repeated in multiple functions.**
   In the **SporeStake** and **LiquidityFarming** contracts, the functions **take**(), **takeWithAddress**() and **takeWithBlock**() implement almost the same calculations. These calculations can be implemented in an **internal view** function which can be called by the **take**(), **takeWithAddress**(), and **takeWithBlock**(). This way the contract size will reduce making it more gas efficient.

10. ***PaymentRequested* event should be renamed.**
    In the **SporeStake** and **LiquidityFarming** contracts, an event named **PaymentRequested** is defined. This event is emitted whenever staking and liquidity farming rewards are claimed. The name of the event is not well aligned with its implementation and hence creates room for confusion for other developers/auditors. It should be renamed as PaymentDone or RewardsClaimed.

    Recommendation:
    Consider renaming the above-mentioned event.

Unit Test

No unit tests were provided by the SPORE team.

Recommendation:
Our team suggests that the developers should write extensive test cases for the contracts.

Coverage Report

Coverage report cannot be generated without unit test cases.

Recommendation:
We recommend 100% line and branch coverage for unit test cases.

Automated Auditing

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 contracts.

Contract Library

Contract-library contains the most complete, high-level decompiled representation of all Ethereum smart contracts, with security analysis applied to them in real-time. We performed analysis using contract Library on the Kovan address of the SporeToken, SporeStake, and LiquidityFarming contracts used during manual testing:

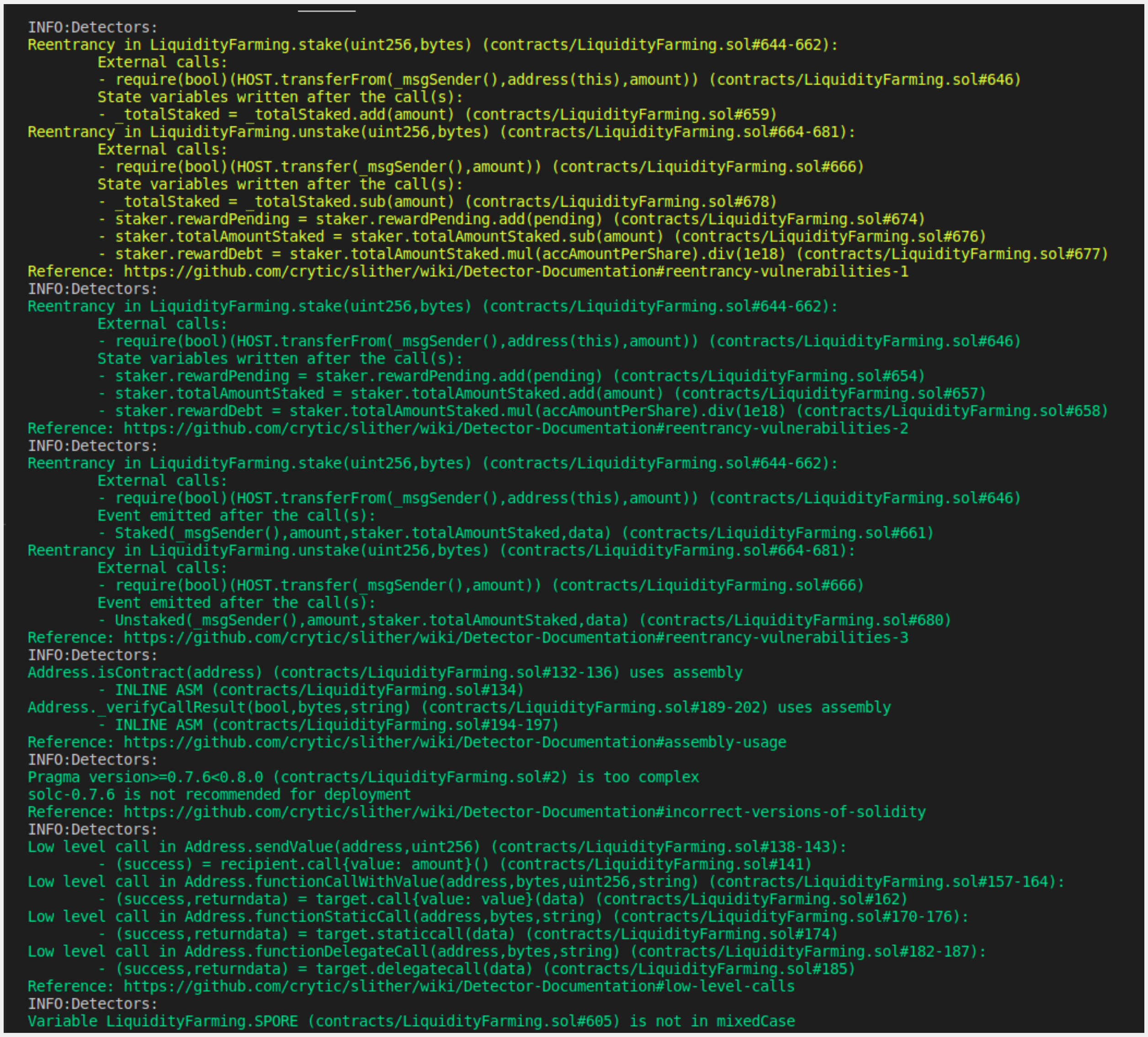
● SporeToken:	0xcebfd289273ebd1f2a4b594070671f284f737db2
● SporeStake:	0x2a3138f6e436de36b2792d5aa1e8b695e5878830
● Liquidity Farming:	0xcf651c746899f35fb96d900301c61535a8b3a6ac

It raises no major concern for the contracts.

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:
Reentrancy in LiquidityFarming.stake(uint256,bytes) (contracts/LiquidityFarming.sol#644-662):
        External calls:
        - require(bool)(HOST.transferFrom(_msgSender(),address(this),amount)) (contracts/LiquidityFarming.sol#646)
        State variables written after the call(s):
        - _totalStaked = _totalStaked.add(amount) (contracts/LiquidityFarming.sol#659)
Reentrancy in LiquidityFarming.unstake(uint256,bytes) (contracts/LiquidityFarming.sol#664-681):
        External calls:
        - require(bool)(HOST.transfer(_msgSender(),amount)) (contracts/LiquidityFarming.sol#666)
        State variables written after the call(s):
        - _totalStaked = _totalStaked.sub(amount) (contracts/LiquidityFarming.sol#678)
        - staker.rewardPending = staker.rewardPending.add(pending) (contracts/LiquidityFarming.sol#674)
        - staker.totalAmountStaked = staker.totalAmountStaked.sub(amount) (contracts/LiquidityFarming.sol#676)
        - staker.rewardDebt = staker.totalAmountStaked.mul(accAmountPerShare).div(1e18) (contracts/LiquidityFarming.sol#677)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#reentrancy-vulnerabilities-1
INFO:Detectors:
Reentrancy in LiquidityFarming.stake(uint256,bytes) (contracts/LiquidityFarming.sol#644-662):
        External calls:
        - require(bool)(HOST.transferFrom(_msgSender(),address(this),amount)) (contracts/LiquidityFarming.sol#646)
        State variables written after the call(s):
        - staker.rewardPending = staker.rewardPending.add(pending) (contracts/LiquidityFarming.sol#654)
        - staker.totalAmountStaked = staker.totalAmountStaked.add(amount) (contracts/LiquidityFarming.sol#657)
        - staker.rewardDebt = staker.totalAmountStaked.mul(accAmountPerShare).div(1e18) (contracts/LiquidityFarming.sol#658)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#reentrancy-vulnerabilities-2
INFO:Detectors:
Reentrancy in LiquidityFarming.stake(uint256,bytes) (contracts/LiquidityFarming.sol#644-662):
        External calls:
        - require(bool)(HOST.transferFrom(_msgSender(),address(this),amount)) (contracts/LiquidityFarming.sol#646)
        Event emitted after the call(s):
        - Staked(_msgSender(),amount,staker.totalAmountStaked,data) (contracts/LiquidityFarming.sol#661)
Reentrancy in LiquidityFarming.unstake(uint256,bytes) (contracts/LiquidityFarming.sol#664-681):
        External calls:
        - require(bool)(HOST.transfer(_msgSender(),amount)) (contracts/LiquidityFarming.sol#666)
        Event emitted after the call(s):
        - Unstaked(_msgSender(),amount,staker.totalAmountStaked,data) (contracts/LiquidityFarming.sol#680)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#reentrancy-vulnerabilities-3
INFO:Detectors:
Address.isContract(address) (contracts/LiquidityFarming.sol#132-136) uses assembly
        - INLINE ASM (contracts/LiquidityFarming.sol#134)
Address._verifyCallResult(bool,bytes,string) (contracts/LiquidityFarming.sol#189-202) uses assembly
        - INLINE ASM (contracts/LiquidityFarming.sol#194-197)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#assembly-usage
INFO:Detectors:
Pragma version>=0.7.6<0.8.0 (contracts/LiquidityFarming.sol#2) is too complex
solc-0.7.6 is not recommended for deployment
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#incorrect-versions-of-solidity
INFO:Detectors:
Low level call in Address.sendValue(address,uint256) (contracts/LiquidityFarming.sol#138-143):
        - (success) = recipient.call{value: amount}() (contracts/LiquidityFarming.sol#141)
Low level call in Address.functionCallWithValue(address,bytes,uint256,string) (contracts/LiquidityFarming.sol#157-164):
        - (success,returndata) = target.call{value: value}(data) (contracts/LiquidityFarming.sol#162)
Low level call in Address.functionStaticCall(address,bytes,string) (contracts/LiquidityFarming.sol#170-176):
        - (success,returndata) = target.staticcall(data) (contracts/LiquidityFarming.sol#174)
Low level call in Address.functionDelegateCall(address,bytes,string) (contracts/LiquidityFarming.sol#182-187):
        - (success,returndata) = target.delegatecall(data) (contracts/LiquidityFarming.sol#185)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#low-level-calls
INFO:Detectors:
Variable LiquidityFarming.SPORE (contracts/LiquidityFarming.sol#605) is not in mixedCase
```

Concluding Remarks

While conducting the audits of the SPORE Engineering smart contract(s), it was observed that the contracts contain Admin/Owner Privileges and Low severity issues, along with several areas of recommendations.

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

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 SPORE Engineering 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.