Mozart Finance

SousChef

Smart Contract Audit Report

IMMUNE BYTES

Audits

March 26, 2021

IMMUNE BYTES

Introduction

1. About Mozart Finance

Mozart Finance is a fork from Goose Finance & astonishing DeFi project running on Binance Smart Chain with lots of other features that let you earn and win tokens.

Mozart Finance is trying to create a splendid perpetual deflation token that performs like a symphony. Their native token PIANO will provide a stable price pump with a sufficient burn mechanism. They are not trying to replace the swap & exchange but rather to add value into our system and create a sustainable environment for people to yield farm with high APR and later even more than that.

Learn more about Mozart Finance here: https://mozartfinance.gitbook.io/mozart-finance/

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

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

1. https://mozartfinance.gitbook.io/mozart-finance/

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: Mozart Finance
- Languages: Solidity(Smart contract), Javascript(Unit Testing)
- Platforms and Tools: Remix IDE, Truffle, Truffle Team, Ganache, Solhint, VScode, Contract Library, Slither, SmartCheck
- Github commits hash/Smart Contract Address for audit: https://testnet.bscscan.com/address/0x99eAeA516cD8C9a36E9D6F3c3B02b92310DB7c2D
- BscScan Code (Testnet):
  SousChef:
  https://testnet.bscscan.com/address/0x56894e113067E9D714B9Bbd8Cd99aB37ca496bbE

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	-	-	-

High severity issues

None

Medium severity issues

None

Low severity issues

None

Recommendations

1. Compiler version for the contract is **0.6.12.** It is recommended to use the latest solidity version to include potential gas and security improvements. This is not a security issue, but just a recommendation to improve the product.

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:
   - ***deposit(uint256) (SousChef.sol#688-702)***
   - ***withdraw(uint256) (SousChef.sol#705-718)***
   - ***emergencyWithdraw() (SousChef.sol#721-728)***

3. The use of the SafeMath library can be improved as suggested below to reduce gas consumption.

   The following lines have used the div() function from the SafeMath library. It can be removed as the divisor is non-zero in these cases hence we can save the gas.

```
Line# 664
tokenReward.mul(1e12).div(stakedSupply)

Line# 682
poolInfo.accRewardPerShare = poolInfo.accRewardPerShare.add(
            tokenReward.mul(1e12).div(foamSupply)
        );
```

```
Line# 697
user.amount.mul(accRewardPerShare).div(1e12).sub(user.rewardDebt)
.add(user.rewardPending);

Line# 699
user.rewardDebt =
user.amount.mul(poolInfo.accRewardPerShare).div(1e12);
```

Unit Test

```
> PancakeSwap@1.0.0 test
> npx builder test

You probably meant to type buidler. We got you.
All contracts have already been compiled, skipping compilation.


  Contract: SousChef
    ✓ sous chef now (2506ms)
    ✓ try syrup (928ms)
    ✓ emergencyWithdraw (94ms)


  3 passing (4s)
```

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 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 SousChef contract, used during manual testing:

- SousChef: 0xFd608F31C5980Cf309d204dfd7a393B1e0cB0c44

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:
SousChef.updatePool() (SousChef.sol#670-684) uses a dangerous strict equality:
	- foamSupply == 0 (SousChef.sol#675)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#dangerous-strict-equalities
INFO:Detectors:
Reentrancy in SousChef.deposit(uint256) (SousChef.sol#688-702):
	External calls:
	- foam.safeTransferFrom(address(msg.sender),address(this),_amount) (SousChef.sol#692)
	State variables written after the call(s):
	- user.rewardPending =
user.amount.mul(poolInfo.accRewardPerShare).div(1e12).sub(user.rewardDebt).add(user.rewardPending)
(SousChef.sol#697)
	- user.amount = user.amount.add(_amount) (SousChef.sol#698)
	- user.rewardDebt = user.amount.mul(poolInfo.accRewardPerShare).div(1e12) (SousChef.sol#699)
Reentrancy in SousChef.emergencyWithdraw() (SousChef.sol#721-728):
	External calls:
	- foam.safeTransfer(address(msg.sender),user.amount) (SousChef.sol#723)
	State variables written after the call(s):
	- user.amount = 0 (SousChef.sol#725)
	- user.rewardDebt = 0 (SousChef.sol#726)
	- user.rewardPending = 0 (SousChef.sol#727)
Reentrancy in SousChef.withdraw(uint256) (SousChef.sol#705-718):
	External calls:
	- foam.safeTransfer(address(msg.sender),_amount) (SousChef.sol#711)
	State variables written after the call(s):
	- user.rewardPending =
user.amount.mul(poolInfo.accRewardPerShare).div(1e12).sub(user.rewardDebt).add(user.rewardPending)
(SousChef.sol#713)
	- user.amount = user.amount.sub(_amount) (SousChef.sol#714)
```

```
        - user.rewardDebt = user.amount.mul(poolInfo.accRewardPerShare).div(1e12) (SousChef.sol#715)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#reentrancy-vulnerabilities-1
INFO:Detectors:
Reentrancy in SousChef.deposit(uint256) (SousChef.sol#688-702):
        External calls:
        - foam.safeTransferFrom(address(msg.sender),address(this),_amount) (SousChef.sol#692)
        State variables written after the call(s):
        - addressList.push(address(msg.sender)) (SousChef.sol#695)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#reentrancy-vulnerabilities-2
INFO:Detectors:
Reentrancy in SousChef.deposit(uint256) (SousChef.sol#688-702):
        External calls:
        - foam.safeTransferFrom(address(msg.sender),address(this),_amount) (SousChef.sol#692)
        Event emitted after the call(s):
        - Deposit(msg.sender,_amount) (SousChef.sol#701)
Reentrancy in SousChef.emergencyWithdraw() (SousChef.sol#721-728):
        External calls:
        - foam.safeTransfer(address(msg.sender),user.amount) (SousChef.sol#723)
        Event emitted after the call(s):
        - EmergencyWithdraw(msg.sender,user.amount) (SousChef.sol#724)
Reentrancy in SousChef.withdraw(uint256) (SousChef.sol#705-718):
        External calls:
        - foam.safeTransfer(address(msg.sender),_amount) (SousChef.sol#711)
        Event emitted after the call(s):
        - Withdraw(msg.sender,_amount) (SousChef.sol#717)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#reentrancy-vulnerabilities-3
INFO:Detectors:
Address.isContract(address) (SousChef.sol#327-338) uses assembly
        - INLINE ASM (SousChef.sol#334-336)
Address._functionCallWithValue(address,bytes,uint256,string) (SousChef.sol#435-461) uses assembly
        - INLINE ASM (SousChef.sol#453-456)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#assembly-usage
INFO:Detectors:
Different versions of Solidity is used in :
        - Version used: ['0.6.12', '>=0.4.0']
        - >=0.4.0 (SousChef.sol#17)
        - 0.6.12 (SousChef.sol#563)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#different-pragma-directives-are-used
INFO:Detectors:
Pragma version>=0.4.0 (SousChef.sol#17) allows old versions
Pragma version0.6.12 (SousChef.sol#563) necessitates a version too recent to be trusted. Consider deploying with
0.6.11
solc-0.6.12 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) (SousChef.sol#356-362):
        - (success) = recipient.call{value: amount}() (SousChef.sol#360)
Low level call in Address._functionCallWithValue(address,bytes,uint256,string) (SousChef.sol#435-461):
        - (success,returndata) = target.call{value: weiValue}(data) (SousChef.sol#444)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#low-level-calls
INFO:Detectors:
Parameter SousChef.getMultiplier(uint256,uint256)._from (SousChef.sol#645) is not in mixedCase
Parameter SousChef.getMultiplier(uint256,uint256)._to (SousChef.sol#645) is not in mixedCase
Parameter SousChef.pendingReward(address)._user (SousChef.sol#656) is not in mixedCase
Parameter SousChef.deposit(uint256)._amount (SousChef.sol#688) is not in mixedCase
Parameter SousChef.withdraw(uint256)._amount (SousChef.sol#705) is not in mixedCase
Reference:
```

```
https://github.com/crytic/slither/wiki/Detector-Documentation#conformance-to-solidity-naming-conventions
INFO:Detectors:
deposit(uint256) should be declared external:
	- SousChef.deposit(uint256) (SousChef.sol#688-702)
withdraw(uint256) should be declared external:
	- SousChef.withdraw(uint256) (SousChef.sol#705-718)
emergencyWithdraw() should be declared external:
	- SousChef.emergencyWithdraw() (SousChef.sol#721-728)
Reference:
https://github.com/crytic/slither/wiki/Detector-Documentation#public-function-that-could-be-declared-external
INFO:Slither:. analyzed (5 contracts with 72 detectors), 24 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 Mozart Finance smart contract(SousChef), it was observed that the contracts contain only a few areas of recommendations.

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 Mozart Finance platform or its product neither this audit is investment advice.
Notes:

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

ImmuneBytes Pvt Ltd.