BlockX Finance
Smart Contract Audit Report

November 19, 2020

Introduction 3
About BlockX Finance 3
About ImmuneBytes 3
Documentation Details 3
Audit Process & Methodology 4
Audit Details 4
Audit Goals 5
Security Level References 5
High severity issues 6
Medium severity issues 6
Low severity issues 6
Admin Privileges 8
Unit Tests 9
Coverage Report 9
Slither Tool Result 10
Notes 10
Concluding Remarks 11
Disclaimer 11

Introduction

1. About BlockX Finance

SPX is the flagship token offered by BLOCKX. SPX is an ERC20 token that represents ownership in the stocks of the S&P 500 Index of the United States stock market, which is composed of 500 of the largest and most successful corporations that are publicly traded in the US. SPX accomplishes this goal through direct ownership of the SPDR® S&P 500® ETF Trust.

Visit https://blockx.finance/ 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

BlockX team has provided documentation for the purpose of conducting the audit via GitHub. The documents are:

1. SPX Whitepaper.pdf
2. SPX Token Specification.pdf
3. SPX Crowdsale Specification.pdf
4. SPX Contract deployment information.pdf

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: BlockX Finance
- Languages: Solidity(Smart contract), Javascript(Unit Testing)
- Github commit hash for audit: fd2e4a847f40a4b39da0a12db45367a318267221

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	-	-	6
Closed	-	-	-

High severity issues

No Issues Found

Medium severity issues

No Issue Found

Low severity issues

1. **Redundant initialization of variables to their default value.**
   In the **constructor** of **SPX** smart contract **(Line 122)**, **totalSupply_** is assigned a **zero(0)** value which is the default value for **uint** type variables in Solidity. The same is the case with **exchangeRate uint** variable in **SPXCrowdsale** contract **(Line 24).**

   Recommendation:
   Consider removing the redundant initialization to save contract deployment gas.

2. **SafeMath library not used for an arithmetic operation.**
   In the **contribute**() function of **SPXCrowdsale** contract (**Line 81**), Solidity's **+=** operator is used instead of SafeMath's **add**() function. However, the chances of overflowing of **uint256 totalRaised** are rare but still, it is a standard practice to use SafeMath whenever arithmetic operations are performed.

   Recommendation:
   Consider using SafeMath's **add**() function at the mentioned statement.

3. **Revert reason strings not provided.**
   At **Lines 79, 106, and 164** of **SPXCrowdsale** contract, **require**() statements are used however the revert strings are not provided in these require statements. It is a standard practice to always provide revert reasons in **require** statements.

   Recommendation:
   Consider adding revert strings in the mentioned statements

4. **No visibility specified for variables**
   In the **SPXCrowdsale** contract, most variables are declared without visibility specifier, hence by default, all those variables are marked as **private**. These variables include **saleEndTime**, **exchangeRate**, **totalRaised**, **allowClaiming**, **SPXContractAddress**,

and **mintingFee**. Getters for these variables are also not implemented. It might get difficult for users and off-chain applications (frontends) to read these crucial values.

Recommendation:
Consider marking these mentioned variables as public or implement getter functions for these variables.

5. **Arithmetic operations can be implemented in a gas efficient way.**
   Lines 159 - 161 of **SPXCrowdsale** are implemented in a way that every arithmetic operation is executed in a separate statement and the intermediate results are stored in **userMintTotal** variable. Storing data in memory also costs gas in Ethereum. These statements can be chained together in a single statement.

6. **No mechanism to deal with unforeseen bugs**
   There is no logic implemented in **SPXCrowdsale** smart contract to deal with future unforeseen bugs/upgrades.

   Recommendation:
   Consider adding a mechanism to pause the functioning of the smart contract if any vulnerability is detected in the future. The ability to upgrade the code of smart contracts may also be considered.

Admin Privileges

The **admin/owner** of **SPX** and **SPXCrowdsale** 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. The **setSPXContractAddress**() in **SPXCrowdsale** contract can be called any number of times. Also, any ethereum address can be set as **SPXContractAddress**.

2. The **setSaleEndTime**() in **SPXCrowdsale** contract can be called any number of times by which any **uint** value can be set as **saleEndTime**.

3. In **SPX** token contract **feeController** address has the ability to change the SPX token transfer fee anytime. The fee can range from 0% to 99%.

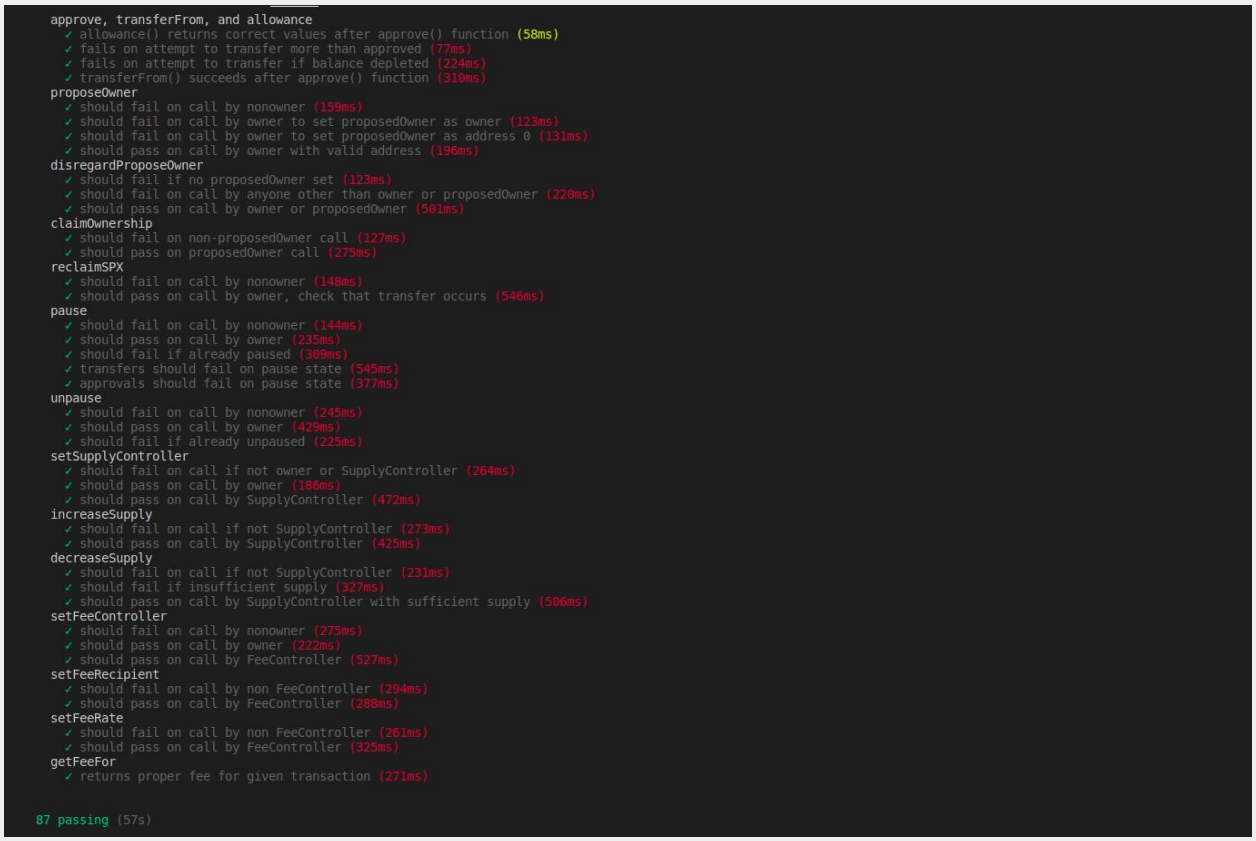
4. The **setMintingFee**() in **SPXCrowdsale** contract can be called any number of times by the admin/owner by which any **uint** value can be set as **mintingFee**. Also, **mintingFee** can be changed at the time of SPX token claiming.

5. The **setExchangeRate**() in **SPXCrowdsale** contract can be called any number of times by which any **uint** value can be set as **exchangeRate.**

Recommendation:
Consider hardcoding a predefined range for variables. Also, consider adding some governance for admin rights for smart contracts or at least use a multi-sig wallet as admin/owner address.

Unit Tests

All unit tests provided by the SPX team are passing without any issues.

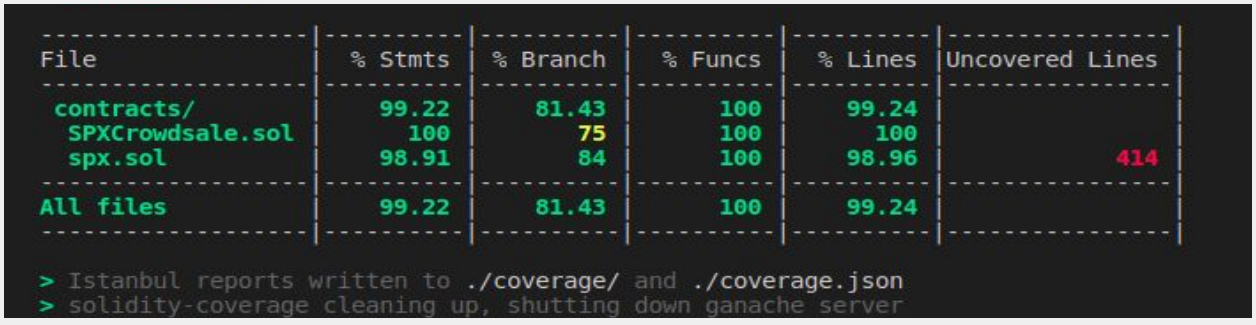


```
  approve, transferFrom, and allowance
    ✓ allowance() returns correct values after approve() function (58ms)
    ✓ fails on attempt to transfer more than approved (77ms)
    ✓ fails on attempt to transfer if balance depleted (224ms)
    ✓ transferFrom() succeeds after approve() function (310ms)
  proposeOwner
    ✓ should fail on call by nonowner (159ms)
    ✓ should fail on call by owner to set proposedOwner as owner (123ms)
    ✓ should fail on call by owner to set proposedOwner as address 0 (131ms)
    ✓ should pass on call by owner with valid address (196ms)
  disregardProposeOwner
    ✓ should fail if no proposedOwner set (123ms)
    ✓ should fail on call by anyone other than owner or proposedOwner (220ms)
    ✓ should pass on call by owner or proposedOwner (501ms)
  claimOwnership
    ✓ should fail on non-proposedOwner call (127ms)
    ✓ should pass on proposedOwner call (275ms)
  reclaimSPX
    ✓ should fail on call by nonowner (148ms)
    ✓ should pass on call by owner, check that transfer occurs (546ms)
  pause
    ✓ should fail on call by nonowner (144ms)
    ✓ should pass on call by owner (235ms)
    ✓ should fail if already paused (309ms)
    ✓ transfers should fail on pause state (545ms)
    ✓ approvals should fail on pause state (377ms)
  unpause
    ✓ should fail on call by nonowner (245ms)
    ✓ should pass on call by owner (429ms)
    ✓ should fail if already unpaused (225ms)
  setSupplyController
    ✓ should fail on call if not owner or SupplyController (264ms)
    ✓ should pass on call by owner (186ms)
    ✓ should pass on call by SupplyController (472ms)
  increaseSupply
    ✓ should fail on call if not SupplyController (273ms)
    ✓ should pass on call by SupplyController (425ms)
  decreaseSupply
    ✓ should fail on call if not SupplyController (231ms)
    ✓ should fail if insufficient supply (327ms)
    ✓ should pass on call by SupplyController with sufficient supply (506ms)
  setFeeController
    ✓ should fail on call by nonowner (275ms)
    ✓ should pass on call by owner (222ms)
    ✓ should pass on call by FeeController (527ms)
  setFeeRecipient
    ✓ should fail on call by non FeeController (294ms)
    ✓ should pass on call by FeeController (288ms)
  setFeeRate
    ✓ should fail on call by non FeeController (261ms)
    ✓ should pass on call by FeeController (325ms)
  getFeeFor
    ✓ returns proper fee for given transaction (271ms)


87 passing (57s)
```

Coverage Report

Test coverage of smart contracts is not 100%.



```
--------------------|----------|----------|----------|----------|----------------|
File                |  % Stmts | % Branch |  % Funcs |  % Lines |Uncovered Lines |
--------------------|----------|----------|----------|----------|----------------|
 contracts/         |    99.22 |    81.43 |      100 |    99.24 |                |
  SPXCrowdsale.sol  |      100 |       75 |      100 |      100 |                |
  spx.sol           |    98.91 |       84 |      100 |    98.96 |            414 |
--------------------|----------|----------|----------|----------|----------------|
All files           |    99.22 |    81.43 |      100 |    99.24 |                |
--------------------|----------|----------|----------|----------|----------------|

> Istanbul reports written to ./coverage/ and ./coverage.json
> solidity-coverage cleaning up, shutting down ganache server
```

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

Slither Tool Result

```
Low level call in SPXCrowdsale.allowClaims() (SPXCrowdsale.sol#136-146):
        - (success,returnData) = SPXContractAddress.call(payload) (SPXCrowdsale.sol#144)
SPXCrowdsale.allowClaims() (SPXCrowdsale.sol#136-146) uses timestamp for comparisons
        Dangerous comparisons:
        - require(bool,string)(now >= saleEndTime,Claims only allowed at end of sale) (SPXCrowdsale.sol#137)
```

```
Low level call in SPXCrowdsale.claim() (SPXCrowdsale.sol#153-166):
        - (success,returnData) = address(SPXContractAddress).call(payload) (SPXCrowdsale.sol#163)
Reentrancy in SPXCrowdsale.claim() (SPXCrowdsale.sol#153-166):
        External calls:
        - (success,returnData) = address(SPXContractAddress).call(payload) (SPXCrowdsale.sol#163)
        Event emitted after the call(s):
        - Claimed(msg.sender,userMintTotal) (SPXCrowdsale.sol#165)
```

Notes

1. Our team at ImmuneBytes confirms that the two low-level calls in **claim**() and **allowClaim**() functions of the **SPXCrowdsale** contract are implemented correctly. The return value of these low-level calls is checked explicitly.

2. The reentrancy warning at **claim**() function can be safely ignored. While updating **ethBalance[msg.sender]** before the external call is the right thing to do, it should also be noted that reentrancy occurs when an external call is made to an **untrusted** contract or when some Ether amount is transferred. As the external call is made to **SPX** token contract which is a trusted contract, there are no chances of reentrancy.

Concluding Remarks

While conducting the audits of BlockX Finance smart contracts, it was observed that the contracts contain only Low severity issues, along with several areas of recommendations.

Our auditors suggest that issues should be resolved by BlockX Finance developers. Resolving the areas of recommendations are up to BlockX Finance'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 BlockX 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.