Ether Authority

JETSWAP PROTOCOL SMART CONTRACT CODE REVIEW AND SECURITY ANALYSIS REPORT

Customer:	JetFuel Team (https://jetfuel.finance)
Prepared on:	23/03/2021
Platform:	Binance Smart Chain
Language:	Solidity
Audit Type:	Extensive

audit@etherauthority.io

Table of contents

THIS IS SECURITY AUDIT REPORT DOCUMENT AND WHICH MAY CONTAIN INFORMATION WHICH IS CONFIDENTIAL. WHICH INCLUDES ANY POTENTIAL VULNERABILITIES AND MALICIOUS CODES WHICH CAN BE USED TO EXPLOIT THE SOFTWARE. THIS MUST BE REFERRED INTERNALLY AND ONLY SHOULD BE MADE AVAILABLE TO PUBLIC AFTER ISSUES ARE RESOLVED.

Project files

Name	Smart Contract Code Review and Security Analysis Report for JETSWAP
Platform	Binance Smart Chain / Solidity
File 1	MasterChef.sol
File 1 MD5 hash	67A621040EA4CD5B851BC8E797D87F8A
File 1 Testnet Contract URL	https://testnet.bscscan.com/address/0xc893573a8528e3c552912eeb934c8e5e7894462a#code
File 2	Multicall.sol
File 2 MD5 hash	C5C1107C4FC647B326284AF5FD0B00EE
File 2 Testnet Contract URL	https://testnet.bscscan.com/address/0xa15fa9d67ed47b35a9e478007d943db1c1286db6#code
File 3	swapV2Factory.sol
File 3 MD5 hash	A6440A04AC2D604CC79A4C62A8C89120
File 3 Testnet Contract URL	https://testnet.bscscan.com/address/0x5659b81b7ca5233bd999073e49fc417e05dc2363#code
File 4	swapV2Router02.sol
File 4 MD5 hash	1140683976AAD1D9A7796FC38957AF94
File 4 Testnet Contract URL	https://testnet.bscscan.com/address/0x27f08ca6ff0d9891a66e4ebc3ce9d46a1873db3a#code
File 5	WingsToken.sol
File 5 MD5 hash	EBCE5069F77A8D0F398133B03F5CD5B8
File 5 Testnet Contract URL	https://testnet.bscscan.com/address/0x34853a9f7f63d8685b7fe32469b5bdb55c212d20#code

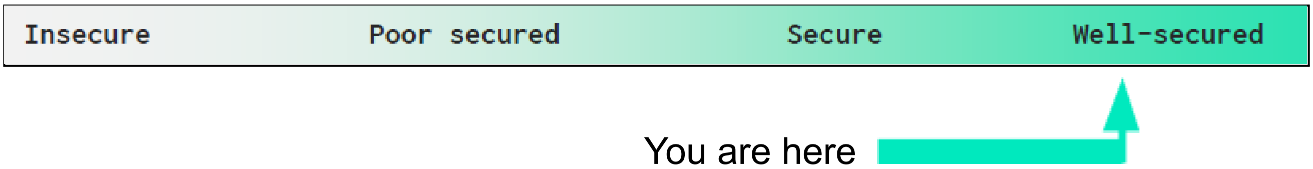
Quick Stats:

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	Moderated
	Function input parameters check bypass	Passed
	Function access control lacks management	Passed
	Critical operation lacks event log	Moderated
	Human/contract checks bypass	Passed
	Random number generation/use vulnerability	N/A
	Fallback function misuse	Passed
	Race condition	Passed
	Logical vulnerability	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	Moderted
	“Short Address” Attack	Passed
	“Double Spend” Attack	Passed

Overall Audit Result: PASSED

Executive Summary

According to the **extensive** audit assessment, Customer`s solidity smart contract is **well secured**.



We used various tools like SmartDec, 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 found issues can be found in the Audit overview section.

We found 0 high, 2 medium and 1 low and some very low level issues.

Code Quality

Jetswap protocol consists of 5 core smart contract files. These smart contracts also contain Libraries, Smart contract inherits and Interfaces. These are compact and well written contracts.

The libraries in the Jetswap 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 Jetswap protocol.

The Jetswap 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 well commented. Commenting can provide rich documentation for functions, return variables and more. Ethereum Natural Language Specification Format (NatSpec) is used, which is a good thing.

Documentation

We were given Jetswap smart contracts in the form of solidity files. The hashes of those files and their testnet links are mentioned above in the table.

As mentioned above, most code parts are well commented. so anyone can quickly understand the programming flow as well as complex code logic. Comments are very helpful in understanding the overall architecture of the protocol. It also provided a clear overview of the system components, including helpful details, like the lifetime of the background script.

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, Jetswap smart contracts depend on an inter-connected set of smart contracts.

AS-IS overview

Jetswap protocol is a decentralized exchange running on Binance Smart Chain, with other features like staking, farming, governance tokens, etc. Following are the main components of core smart contracts.

MasterChef.sol

(1) Inherited contracts

(a) Ownable: ownership contract

(2) Usages

(a) using SafeMath for uint256
(b) using SafeBEP20 for IBEP20

(3) Structs

(a) UserInfo: Info about each user
(b) PoolInfo: Info about each pools

(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	Passed	No Issue	Passed
2	updateMultiplier	write	Passed	No Issue	Passed
3	poolLength	read	Passed	No Issue	Passed
4	add	write	Input validation missing	LP Token must not be added twice	Passed with consent
5	set	write	Passed	No Issue	Passed

6	updateStakingPool	internal	Infinite loop possibility	Array length must be limited	Passed with consent
7	getMultiplier	read	Passed	No Issue	Passed
8	pendingWings	read	Passed	No Issue	Passed
9	massUpdatePools	write	Infinite loop possibility	Array length must be 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	enterStaking	write	Passed	No Issue	Passed
15	leaveStaking	write	Passed	No Issue	Passed
16	emergencyWithdraw	write	Passed	No Issue	Passed
17	safeWingsTransfer	write	Passed	No Issue	Passed
18	dev	write	Passed	No Issue	Passed

Multicall.sol

(1) Struct

(a) Call: holds call data and target wallet

(2) Functions

Sl.	Function	Type	Observation	Conclusion	Score
1	aggregate	read	Passed	No Issue	Passed
2	getEthBalance	read	Passed	No Issue	Passed
3	getBlockHash	read	Passed	No Issue	Passed
4	getLastBlockHash	read	Passed	No Issue	Passed
5	getCurrentBlockTimestamp	read	Passed	No Issue	Passed
6	getCurrentBlockDifficulty	read	Passed	No Issue	Passed
7	getCurrentBlockGasLimit	read	Passed	No Issue	Passed
8	getCurrentBlockCoinbase	read	Passed	No Issue	Passed

JetswapFactory.sol

(1) Interfaces

(a) IJetswapFactory
(b) IJetswapPair
(c) IJetswapERC20
(d) IERC20
(e) IJetswapCallee

(2) Inherits

(a) IJetswapFactory

(3) Events

(a) event PairCreated(address indexed token0, address indexed token1, address pair, uint);

(4) Functions

Sl.	Function	Type	Observation	Conclusion	Score
1	constructor	write	Passed	No Issue	Passed
2	allPairsLength	read	Passed	No Issue	Passed
3	createPair	write	Passed	No Issue	Passed
4	setFeeTo	write	Passed	No Issue	Passed
5	setFeeToSetter	write	Passed	No Issue	Passed

JetswapRouter.sol

(1) Interfaces

(a) IJetswapFactory
(b) IJetswapRouter01
(c) IJetswapRouter02
(d) IJetswapPair
(e) IERC20
(f) IWETH

(2) Inherited contracts

(a) IJetswapRouter02

(3) Usages

(a) using SafeMath for uint

(4) Functions

Sl	Function	Type	Observation	Conclusion	Score
1	constructor	write	Passed	No Issue	Passed
2	_addLiquidity	internal	Passed	No Issue	Passed
3	addLiquidity	write	Passed	No Issue	Passed
4	addLiquidityETH	write	Passed	No Issue	Passed
5	removeLiquidity	write	Passed	No Issue	Passed
6	removeLiquidityETH	write	Passed	No Issue	Passed
7	removeLiquidityWithPermit	write	Passed	No Issue	Passed
8	removeLiquidityETHWithPermit	write	Passed	No Issue	Passed
9	removeLiquidityETHSupportingFeeOnTransferTokens	write	Passed	No Issue	Passed
10	removeLiquidityETHWithPermitSupportingFeeOnTransferTokens	internal	Passed	No Issue	Passed
11	_swap	internal	Infinite loop possibility	Keep path limited	Passed with consent
12	swapExactTokensForTokens	write	Passed	No Issue	Passed

13	swapTokensForExactTokens	write	Passed	No Issue	Passed
14	swapExactETHForTokens	write	Passed	No Issue	Passed
15	swapTokensForExactETH	write	Passed	No Issue	Passed
16	swapExactTokensForETH	write	Passed	No Issue	Passed
17	swapETHForExactTokens	write	Passed	No Issue	Passed
18	_swapSupportingFeeOnTransferTokens	internal	Infinite loop possibility	Keep path limited	Passed with consent
19	swapExactTokensForTokensSupportingFeeOnTransferTokens	write	Passed	No Issue	Passed
20	swapExactETHForTokensSupportingFeeOnTransferTokens	write	Passed	No Issue	Passed
21	swapExactTokensForETHSupportingFeeOnTransferTokens	write	Passed	No Issue	Passed
22	quote	read	Passed	No Issue	Passed
23	getAmountOut	read	Passed	No Issue	Passed
24	getAmountIn	read	Passed	No Issue	Passed
25	getAmountsOut	read	Passed	No Issue	Passed
26	getAmountsIn	read	Passed	No Issue	Passed

WingsToken.sol

(1) Interfaces

(a) IBEP20

(2) Inherited contracts

(a) Context: Provides msg.sender and msg.value context

(b) Ownable: Ownership contract

(c) BEP20: Standard contract for BEP20

(3) Events

(a) event DelegateChanged(address indexed delegator, address indexed fromDelegate, address indexed toDelegate);

(b) event DelegateVotesChanged(address indexed delegate, uint256 previousBalance, uint256 newBalance);

(4) Functions

Sl.	Function	Type	Observation	Conclusion	Score
1	mint	write	No max minting set	must be used carefully	Passed with consent
2	delegates	read	Passed	No Issue	Passed
3	delegate	write	Passed	No Issue	Passed
4	delegateBySig	write	Passed	No Issue	Passed
5	getCurrentVotes	read	Passed	No Issue	Passed
6	getPriorVotes	read	Infinite loop possibility	Keep array length limited	Passed with consent
7	_delegate	internal	Passed	No Issue	Passed
8	_moveDelegates	internal	Passed	No Issue	Passed
9	_writeCheckpoint	internal	Passed	No Issue	Passed
10	safe32	read	Passed	No Issue	Passed
11	getChainId	read	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

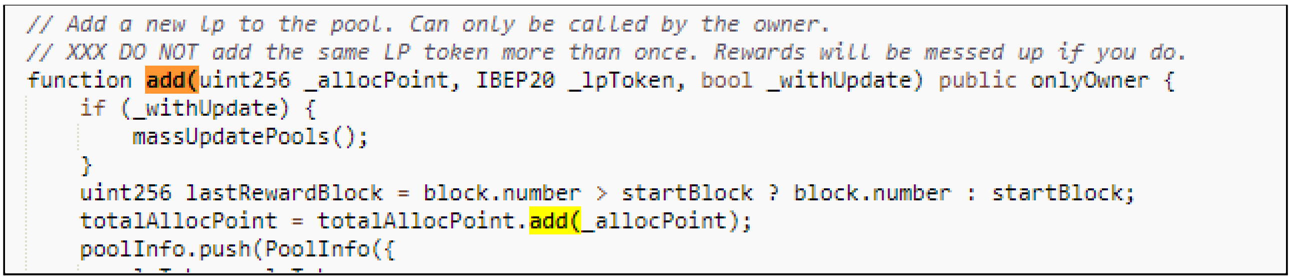
No critical severity vulnerabilities were found.

High

No high severity vulnerabilities were found.

Medium

(1) Input validation missing in MasterChef.sol

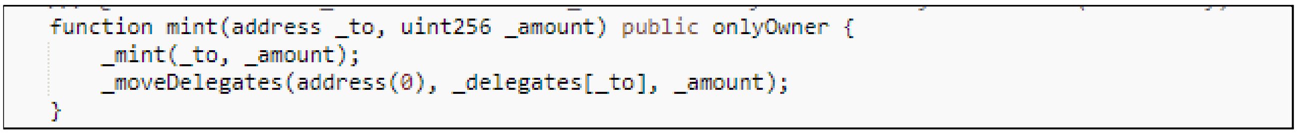


```
// Add a new lp to the pool. Can only be called by the owner.
// XXX DO NOT add the same LP token more than once. Rewards will be messed up if you do.
function add(uint256 _allocPoint, IBEP20 _lpToken, bool _withUpdate) public onlyOwner {
    if (_withUpdate) {
        massUpdatePools();
    }
    uint256 lastRewardBlock = block.number > startBlock ? block.number : startBlock;
    totalAllocPoint = totalAllocPoint.add(_allocPoint);
    poolInfo.push(PoolInfo({
```

As mentioned in the comment, the token must never be added twice. So, there must be a condition to prevent that happening by mistake.

Resolution: we got confirmation from the Jetswap team as this will be taken extra care as this is the owner function.

(2) Minting can be unlimited by owner in WingsToken.sol



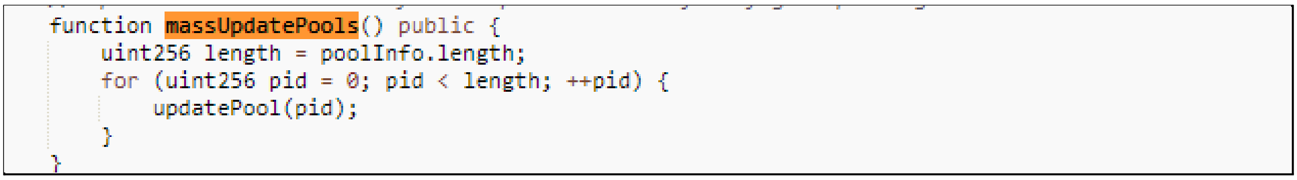
```
function mint(address _to, uint256 _amount) public onlyOwner {
    _mint(_to, _amount);
    _moveDelegates(address(0), _delegates[_to], _amount);
}
```

Unlimited minting is considered a bad practice for tokenomics and hence it should be discouraged.

Resolution: Jetswap team confirmed that this minting would be triggered by masterChef contract only.

Low

(1) Infinite loops possibility at multiple places:



```
function massUpdatePools() public {
    uint256 length = poolInfo.length;
    for (uint256 pid = 0; pid < length; ++pid) {
        updatePool(pid);
    }
}
```

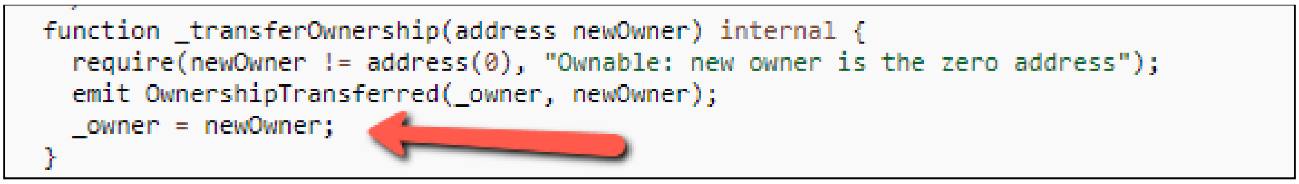
As seen in the AS-IS section, there are several places in the smart contracts, where array.length is used directly in the loops. It is recommended to put some kind of limits, so it does not go wild and create any scenario where it can hit the block gas limit.

Resolution: We got confirmation from the Jetswap team that the array will be provided as limited length. And this will be taken care of from the client side.

Very Low

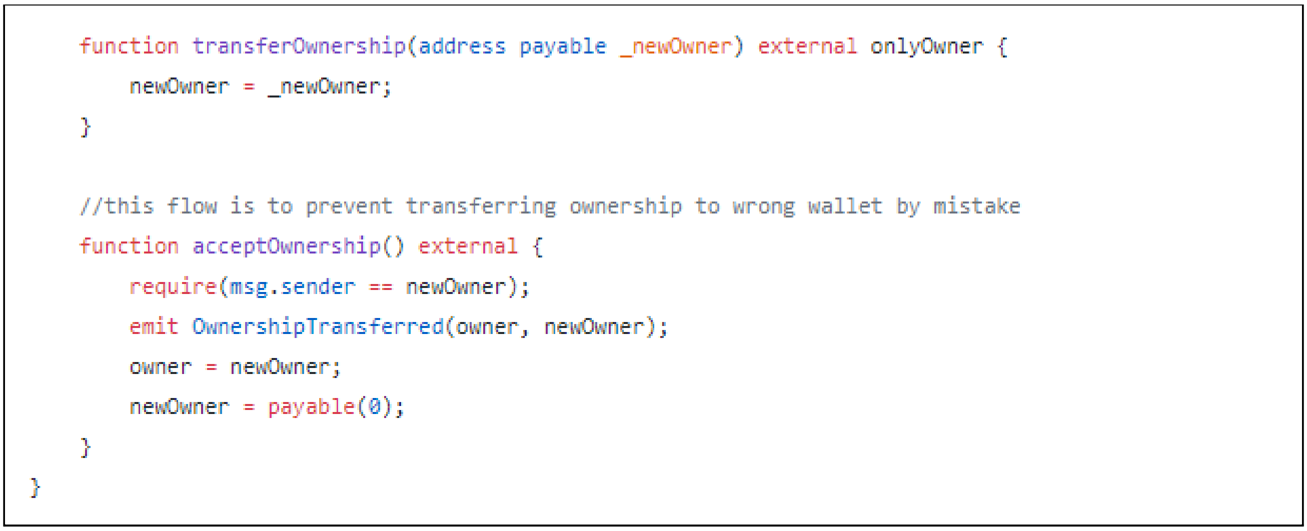
(1) Ownership transfer function:

Ownable.sol smart contract has active ownership transfer. This will be troublesome if the ownership was sent to an incorrect address by human error.



```
function _transferOwnership(address newOwner) internal {
  require(newOwner != address(0), "Ownable: new owner is the zero address");
  emit OwnershipTransferred(_owner, newOwner);
  _owner = newOwner;
}
```

so, it is a good practice to implement acceptOwnership style to prevent it. Code flow similar to below:



```
    function transferOwnership(address payable _newOwner) external onlyOwner {
        newOwner = _newOwner;
    }

    //this flow is to prevent transferring ownership to wrong wallet by mistake
    function acceptOwnership() external {
        require(msg.sender == newOwner);
        emit OwnershipTransferred(owner, newOwner);
        owner = newOwner;
        newOwner = payable(0);
    }
}
```

Resolution: Jetswap team acknowledged this, as this should be taken care of from admin side.

(2) Use the latest solidity version while contract deployment to prevent any compiler version level bugs.

Resolution: This issue is acknowledged.

(3) Event log must be fired in place where the stats are being changed. for example:

- setFeeTo function in JetswapFactory.sol
- setFeeToSetter function in JetswapFactory.sol
- initialize function in JetswapFactory.sol

Resolution: This issue is acknowledged.

Conclusion

We were given 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. **So it is good to go for the production.**

Since possible test cases can be unlimited for such extensive smart contract protocol, so 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 contract, based on extensive 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.

Email: audit@EtherAuthority.io