MINNAPAD

SMART CONTRACT AUDIT REPORT

Prepared by:

BlockAudit

Date Of Enrollment:

August 01th, 2023 - August 09th, 2023

Visit : www.blockaudit.report

TABLE OF CONTENTS

SUMMARY

This Audit Report mainly focuses on the extensive security of **MINNAPAD** Smart Contracts. With this report, we attempt to ensure the reliability and correctness of the smart contract by complete and rigorous assessment of the system's architecture and the smart contract codebase.

The auditing process pays special attention to the following considerations:

- Testing the smart contracts against both common and uncommon attack vectors.
- Assessing the codebase to ensure compliance with current best practices and industry standards.
- Ensuring contract logic meets the specifications and intentions of the client.
- Cross referencing contract structure and implementation against similar smart contracts producedby industry leaders.
- Thorough line-by-line manual review of the entire codebase by industry experts.

The security assessment resulted in findings that ranged from critical to informational. We recommend addressing these findings to ensure a high level of security standards and industry practices. We suggest recommendations that could better serve the project from the security perspective:

- Enhance general coding practices for better structures of source codes;
- Add enough unit tests to cover the possible use cases;
- Provide more comments per each function for readability, especially contracts that are verified inpublic;
- Provide more transparency on privileged activities once the protocol is live.

OVERVIEW

Project Summary

Project Name	Minnapad
Logo	
Platform	Polygon (Mumbai)
Language	Solidity
Code Link	https://mumbai.polygonscan.com/address/0x71b1Ae31f57312402675CdbA4f95B98cF8841899 https://mumbai.polygonscan.com/address/0x388018Ce01728c732d7AA92a1f58C9BAc7250abD https://mumbai.polygonscan.com/address/0xEa3E8E5B590F3E2979feeb39979be593531Eb2A1 https://mumbai.polygonscan.com/address/0x751B0AaB748dd9f3255e3AbD4B06E2085b7808fD

File Summary

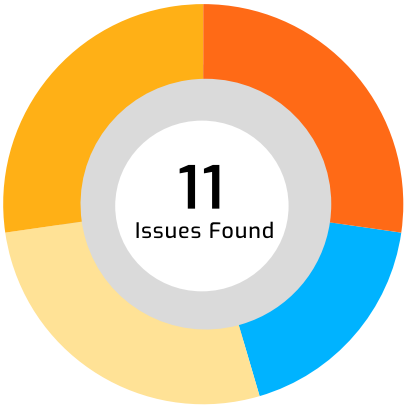
ID	File Name	Audit Status
VOTE	VOTE0-VotingContract.sol	Pass
CTLU	CTLU0-CustomTimeLockUpgradeable.sol	Pass
CC	CC0-ClaimableContract.sol	Pass
LPS	LPS0-LaunchPadStaking.sol	Pass

Audit Summary

Date of Delivery	09 Aug 2023
Audit Methodology	Code Analysis. Automatic Assesment, Manual Review
Audit Result	Passed
Audit Team	BlockAudit Report Team

FINDINGS

Critical	0	0.0%
High	3	27.27%
Medium	2	18.18%
Low	3	27.27%
Informational	3	27.27%
Ownership	0	0.0%
Gas Optimization	0	0.0%



Vulnerability Findings Summary

ID	Type	Line	Severity	Status
VOTE01	Option Validation	73	High	Acknowledged
VOTE02	Membership Token Dependency	53	Medium	Acknowledged
CTLU01	Membership Balance Check	78	Medium	Resolved
CTLU02	Missing Input Validation	243	Low	Acknowledged
CC01	Incorrect Assignment	139	High	Acknowledged
CC02	Missing Array Length Checks	69-103	High	Acknowledged
CC03	Missing Input Validation In Constructor	10 / 13 / 15 / 24/ 26	Low	Acknowledged
CC04	State Variables That Could Be Declared Immutable	29-46	Informational	Acknowledged
LPS01	Missing Input Validation In Constructor	82	Low	Acknowledged
LPS02	State Variables That Could Be Declared Immutable	10	Informational	Acknowledged
LPS03	Unused State Variables	60-62	Informational	Acknowledged

VOTE01

Type	Option Validation
Severity	High
File	VoteContract.sol
Line	73
Status	**Acknowledged**

Description

The function below does not check if there are potential duplicate option hashes within a single question. It does not have specific validation or prevention mechanisms for ensuring unique option hashes within each question.

Remediation

Instead of relying solely on option hashes for validation, use unique identifiers (e.g., integers) assigned to each option in a question. This will ensure that duplicate hashes do not affect the validation process.

Snapshot

```
function addQuestion(
    bytes32 _questionHash,
    bytes32[] memory _optionHashes,
    uint256 _startTime,
    uint256 _endTime
) external canAddQuestion {
    Question storage newQuestion = questions[_questionHash];
    require(
        newQuestion.questionHash == bytes32(0),
        "Question already exists."
    );
    require(
        _optionHashes.length < type(uint8).max,
        "Option count exceeding"
    );

    newQuestion.questionHash = _questionHash;
    newQuestion.optionHashes = _optionHashes;
    newQuestion.optionCount = uint8(_optionHashes.length);

    questionStartTimes[_questionHash] = _startTime;
    questionEndTimes[_questionHash] = _endTime;

    emit QuestionAdded(_questionHash, _msgSender());
}
```

VOTE02

Type	Membership Token Dependency
Severity	Medium
File	VoteContract.sol
Line	53
Status	**Acknowledged**

Description

The contract uses ERC721 membership tokens as criteria for determining whether an address can vote (_canVote function). It relies on external token contracts without any additional security measures or access control mechanisms such as whitelisting specific membership tokens or verifying their authenticity.

Remediation

Implement additional security measures such as verifying token authenticity through digital signatures or whitelisting specific membership tokens based on their contract addresses and/or specific token properties.

Snapshot

```
constructor(
    address _membershipToken,
    uint8 _voteBalance
) payable Ownable() {
    require(
        _membershipToken != address(0) && _membershipToken != address(this),
        "Invalid Membership token Address"
    );
    voteBalance = _voteBalance;
    membershipToken = _membershipToken;
}

```

CTLU01

Type	Membership Balance Check
Severity	Medium
File	CustomTimeLockUpgradeable.sol
Line	78
Status	**Reported**

Description

The _isMember function checks if an address has a balance greater than membershipBalance in the membership token contract to determine if they are a member. However, this check does not account for any potential errors or exceptions that could occur during the execution of balanceOf.

Remediation

Consider handling exceptions or errors that may occur during calls to external contracts like IERC721.balanceOf. Ensure proper error handling and revert transactions with informative error messages when necessary.

Snapshot

```
function _isMember(address _user) internal virtual returns (bool) {
    return IERC721(membershipToken).balanceOf(_user) > membershipBalance;
}

```

CTLU02

Type	Missing Input Validation
Severity	Low
File	CustomTimeLockUpgradeable.sol
Line	29-46
Status	**Acknowledged**

Description

The initialize function assumes that valid values will be passed as input arguments without additional validation or sanity checks.

Remediation

Implement appropriate input validation in the initialize function to verify that inputs are valid and within acceptable ranges before proceeding with initialization logic.

Snapshot

```
function initialize(
    uint256 minDelay,
    // address[] memory proposers,
    address[] memory executors,
    address admin,
    address _membershipToken,
    uint8 _membershipBalance
) public initializer {
    require(
        _membershipToken != address(0) && _membershipToken != address(this),
        "Invaild Membership Token Address"
    );
    __TimelockController_init(minDelay, new address[](0), executors, admin);
    __Ownable_init();
    __UUPSUpgradeable_init();
    membershipToken = _membershipToken;
    membershipBalance = _membershipBalance;
}
```

CC01

Type	Option Validation
Severity	High
File	ClaimableContract.sol
Line	243
Status	**Acknowledged**

Description

In the buyToken() function, there is an incorrect assignment statement: u.FRtokenBuyed == true;.

Remediation

In the buyToken() function, there is an incorrect assignment statement: u.FRtokenBuyed == true;.

Snapshot

```
UserDetail memory u = userDetails[_msgSender()];
if (block.timestamp <= roundOneEndTime) {
    if (
        u.buyedToken == 0 &&
        !u.FRtokenBuyed
    ) {
        u.remainingTokenToBuy = tierDetails[
            tierId
        ][pool].allocatedAmount;
    }
    require(
        amount <= u.remainingTokenToBuy,
        "amount should be lesser than allocated amount"
    );
    u.remainingTokenToBuy -= amount;
    totalSoldToken += amount;
    u.buyedToken += amount;
    u.tokenToSend += amount;
    if (u.remainingTokenToBuy == 0) {
        u.FRtokenBuyed == true;
    }
    emit TokenBought(_msgSender(), tierId, amount);
    return true;
}
```

CC02

Type	Missing Array Length Checks
Severity	High
File	ClaimableContract.sol
Line	139
Status	**Acknowledged**

Description

In the given function, i is declared as uint8 in the first loop. If l (the length of _allocation) is larger than the range of uint8, it can lead to unexpected behavior or errors. The maximum value that can be stored in a uint8 variable is 255. So if the length of _allocation exceeds 255, it will cause an overflow and wrap around back to zero, potentially causing incorrect iterations or out-of-bounds access.

Remediation

To avoid this issue, you should ensure that _allocation.length does not exceed the maximum value allowed for a uint8. You may consider using a larger integer type such as uint256, which has a much larger range and would accommodate any possible length of _allocation.

Snapshot

```
uint l = _allocation.length;
for (uint8 i = 0; i < l; ++i) {
    require(_allocation[i] != address(0), "Zero Address Specified");
    participants[_allocation[i]] = true;
}
```

CC03

Type	Missing Input Validation in Constructor
Severity	Low
File	ClaimableContract.sol
Line	69-103
Status	**Acknowledged**

Description

Constructor does not check if inputs are zero addresses or if other inputs are zeros, So if Bob inputs zero addresses or values. funds can be lost.

Remediation

Check for zero address validation and add checks on values. Also, you may consider adding a way to accept only the whitelisted/approved token/staking contract.

Snapshot

```
constructor(
    IStakeable _stakingContract,
    IERC20Metadata _rewardToken,
    uint _totalsupply,
    uint[] memory tierWeights,
    uint _listingTime,
    uint _claimSlots,
    uint _vestingTime,
    uint _roundOneStartTime,
    uint _roundOneEndTime,
    uint _FCFSStartTime,
    uint _FCFSEndTime
) Ownable() ReentrancyGuard() {
    stakingContract = _stakingContract;
    rewardToken = _rewardToken;
    totalSupply = _totalsupply * 10 ** rewardToken.decimals();
    uint k;
    require(tierWeights.length == 18, "6 x 3 Tier weights needed");
    for (uint i = 0; i < 6; ++i) {
        for (uint j = 0; j < 3; ++j) {
            tierDetails[i][j].tierLevel = i;
            tierDetails[i][j].poolLevel = j;
            tierDetails[i][j].poolWeight = tierWeights[k];
            k++;
        }
    }

```

CC04

Type	State variables that could be declared immutable
Severity	Informational
File	ClaimableContract.sol
Line	10 / 13 / 15 / 24/ 26
Status	**Acknowledged**

Description

State variables that are not updated following deployment should be declared immutable to save gas

Remediation

Add the immutable attribute to state variables that never change or are set only in the constructor.

Snapshot

LPS01

Type	Missing Input Validation in Constructor
Severity	Low
File	LaunchPadStaking.sol
Line	82
Status	**Acknowledged**

Description

Constructor does not check if stakingToken is zero address, So if Bob inputs zero address or another contract that can cause unexpected behavior. Funds can be lost.

Remediation

Check for zero address validation. Also, you may consider adding a way to accept only the whitelisted/approved staking contract.

Snapshot

```
constructor(IERC20Metadata _stakingToken) Ownable() ReentrancyGuard() {
    stakingToken = _stakingToken;
    signer = _msgSender();

    pools[1] = poolDetail(1, 15, 5 seconds);

    pools[2] = poolDetail(2, 50, 180 seconds);

    pools[3] = poolDetail(3, 100, 360 seconds);
}
```

LPS02

Type	State variables that could be declared immutable
Severity	Informational
File	LaunchPadStaking.sol
Line	10
Status	**Reported**

Description

State variables that are not updated following deployment should be declared immutable to save gas.

Remediation

Add the immutable attribute to state variables that never change or are set only in the constructor.

Snapshot

LPS03

Type	Unused State Variable
Severity	Informational
File	LaunchPadStaking.sol
Line	60-62
Status	**Acknowledged**

Description

the state variables below are declared but they are never used across the contract so they are a waste of gas.

Remediation

Remove unused state variables.

Snapshot

```
1  mapping(address => bool) private isWhitelist;
2  address[] private whiteList;
3  uint256 private whitelistCount;
4
```

APPENDIX

Auditing Approach and Methodologies applied

The Block Audit Report team has performed rigorous testing of the project including the analysis of the code design patterns where we reviewed the smart contract architecture to ensure it is structured along with the safe use of standard inherited contracts and libraries. Our team also conducted a formal line by line inspection of the Smart Contract i.e., a manual review, to find potential issues including but not limited to

- Race conditions
- Zero race conditions approval attacks
- Re-entrancy
- Transaction-ordering dependence
- Timestamp dependence
- Check-effects-interaction pattern (optimistic accounting)
- Decentralized denial-of-service attacks
- Secure ether transfer pattern
- Guard check pattern
- Fail-safe mode
- Gas-limits and infinite loops
- Call Stack depth

In the Unit testing Phase, we coded/conducted custom unit tests written against each function in the contract to verify the claimed functionality from our client. In Automated Testing, we tested the Smart Contract with our standard set of multifunctional tools to identify vulnerabilities and security flaws. The code was tested in collaboration of our multiple team members and this included but not limited to;

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

Issue Categories:

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

Critical Severity Issues

Issues of Critical Severity leaves smart contracts vulnerable to major exploits and can lead to asset loss and data loss. These can have significant impact on the functionality/performance of the smart contract.
We recommend these issues must be fixed before proceeding to MainNet..

High Severity Issues

Issues of High Severity are not as easy to exploit but they might endanger the execution of the smart contract and potentially create crucial problems.
Fixing these issues is highly recommended before proceeding to MainNet.

Medium Severity Issues

Issues on this level are not a major cause of vulnerability to the smart contract, they cannot lead to data-manipulations or asset loss but may affect funtionality.
It is important to fix these issues before proceeding to MainNet.

Low Severity Issues

Issues at this level are very low in their impact on the overall functionality and execution of the smart contract. These are mostly code-level violations or improper formatting.
These issues can be remain unfixed or can be fixed at a later date if the code is redeployed or forked.

Informational Findings

These are finding that our team comes accross when manually reviewing a smart contract which are important to know for the owners as well as users of a contract.
These issues must be acknowledged by the owners before we publish our report.

Ownership Privileges

Owner of a smart contract can include certain rights and Privileges while deploying a smart contract that might be hidden deep inside the codebase and may make the project vulnerable to rug-pulls or other types of scams.
We at BlockAudit believe in transparency and hence we showcase Ownership Privileges separately so the owner as well as the investors can get a better understanding about the project.

Gas Optimization

Solidity gas optimization is the process of lowering the cost of operating your Solidity smart code. The term "gas" refers to the level of processing power required to perform specific tasks on the Ethereum network.
Each Ethereum transaction costs a fee since it requires the use of computer resources. It will deduct a fee anytime any function in the smart contract is invoked by the contract's owner or users.

DISCLAIMER

This is a limited report on our findings based on our analysis, in accordance with good industry practice as at the date of this report, in relation to cybersecurity vulnerabilities and issues in the framework and algorithms based on smart contracts, the details of which are set out in this report. In order to get a full view of our analysis, it is crucial for the client to read the full report. While we have done our best in conducting our analysis and producing this report, it is important to note that the client should not rely on this report and cannot claim against us on the basis of what it says or doesn't say, or how we produced it, and it is important for the client to conduct the client's own independent investigations before making any decisions. We go into more detail on this in the below disclaimer below – please make sure to read it in full.

By reading this report or any part of it, the client agrees to the terms of this disclaimer. If the client does not agree to the terms, then please immediately cease reading this report, and delete and destroy any/all copies of this report downloaded and/or printed by the client. This report is provided for information purposes only and stays on a non-reliance basis, and does not constitute investment advice. No one/ NONE shall have any rights to rely on the report or its contents, and BlockAudit and its affiliates (including holding companies, shareholders, subsidiaries, employees, directors, officers and other representatives).

(BlockAudit) owes no duty of care towards the client or any other person, nor does BlockAudit claim any warranty or representation to any person on the accuracy or completeness of the report. The report is provided "as is", without any conditions, warranties or other terms of any kind except as set out in this disclaimer, and BlockAudit hereby excludes all representations, warranties, conditions and other terms (including, without limitation, the warranties implied by law of satisfactory quality, fitness for purpose and the use of reasonable care and skill) which, but for this clause, might have effects in relation to the report.

Except and only to the extent that it is prohibited by law, BlockAudit hereby excludes all liability and responsibility, and neither the client nor any other person shall have any claim against BlockAudit, for any amount or kind of loss or damage that may result to the client or any other person (including without limitation, any direct, indirect, special, punitive, consequential or pure economic loss or damages, or any loss of income, profits, goodwill, data, contracts, use of money, or business interruption, and whether in delict, tort (including without limitation negligence), contract, breach of statutory duty, misrepresentation (whether innocent or negligent) or otherwise under any claim of any nature whatsoever in any jurisdiction) in any way arising from or connected with this report and the use, inability to use or the results of use of this report, and any reliance on this report.

The analysis of the security is purely based on the received smart contracts alone. No related/third-party smart contracts, applications or operations were reviewed for security. No product code has been reviewed.

Note: The statements made in this document should not be interpreted as investment or legal advice, nor should its authors be held accountable for decisions made based on them. Securing smart contracts is a multistep process. One audit cannot be considered enough. We recommend that the **MINNAPAD** team put a bug bounty program in place to encourage further analysis of the smart contracts by other third parties

About BlockAudit

BlockAudit is an industry leading security organisation that helps web3 blockchain based projects with their security and correctness of their smart-contracts. With years of experience we have a dedicated team that is capable of performing audits in a wide variety of languages including HTML, PHP, JS, Node, React, Native, Solidity, Rust and other Web3 frameworks for DApps, DeFi, GameFi and Metaverse platforms.

With a mission to make web3 a safe and secure place BlockAudit is commited to provide it's partners with a budget and investor friendly security Audit Report that will increase the value of their projects significantly.

 *www.blockaudit.report*

 *team@blockaudit.report*

 *@BlockAudit*

 *github.com/Block-Audit-Report*