BLOCK SOLUTIONS

Smart Contract Code Review and Security Analysis Report for ARCHIE Token Staking Smart Contract

Request Date: 2023-05-06
Completion Date: 2023-05-06
Language: Solidity

Contents

Commission

Audited Project	ARCHIE Token Staking Smart Contract
Smart Contract Address	0x9F2FE0CE537dFEe63904dcEFCACC2148ACd8C43E
Contract Deployer	0x5cf547eac02c999d656055cb1076a8658caf3de9
Contract Owner	0xd2C72D1Ef993E4B6600F7D799BbD7CC61fF3a017
Blockchain Network	ArchieChain Mainnet

Block Solutions was commissioned by ARCHIE Token Staking Smart Contract owners to perform an audit of their main smart contract. The purpose of the audit was to achieve the following:

- Ensure that the smart contract functions as intended.
- Identify potential security issues with the smart contract.

The information in this report should be used to understand the risk exposure of the smart contract, and as a guide to improve the security posture of the smart contract by remediating the issues that were identified.

ARCHIE Token Staking Properties

Contract name	ARCHIE Token Staking
Minimum Deposit	100000 ARC
Token Address	0x5e1e437D33A2895AA6bDb20a8810B15cC0D3C569
Deduction Percentage	10%
30 days percentage APY	1%
90 days percentage APY	4.5%
180 days percentage APY	12%
360 days percentage APY	30%

Contract Functions

Executables

i. function changeDeductionPercentage(uint256 amount) public onlyOwner
ii. function changeMinimmumAmount(uint256 amount) external onlyOwner
iii. function changePercentages(uint256 _30dayspercent,uint256 _90dayspercent,uint256 _180dayspercent,uint256 _360dayspercent) external onlyOwner
iv. function changetimeCal(uint256 _time) external onlyOwner
v. function emergencyWithdraw(IARC20 _token ,uint256 _amount) external onlyOwner
vi. function emergencyWithdrawARC(uint256 Amount) external onlyOwner
vii. function farm(uint256 _lockableDays) public payable whenNotPaused
viii. function harvest(uint256 [] memory _index) external whenNotPaused
ix. function pausePool() external onlyOwner
x. function UnpausePool() external onlyOwner
xi. function transferOwnership(address newOwner) public onlyOwner
xii. function addorRemoveSpam(address _Addr,bool _state) external onlyOwner
xiii. function changeToken(IARC20 addr) public onlyOwner

Checklist

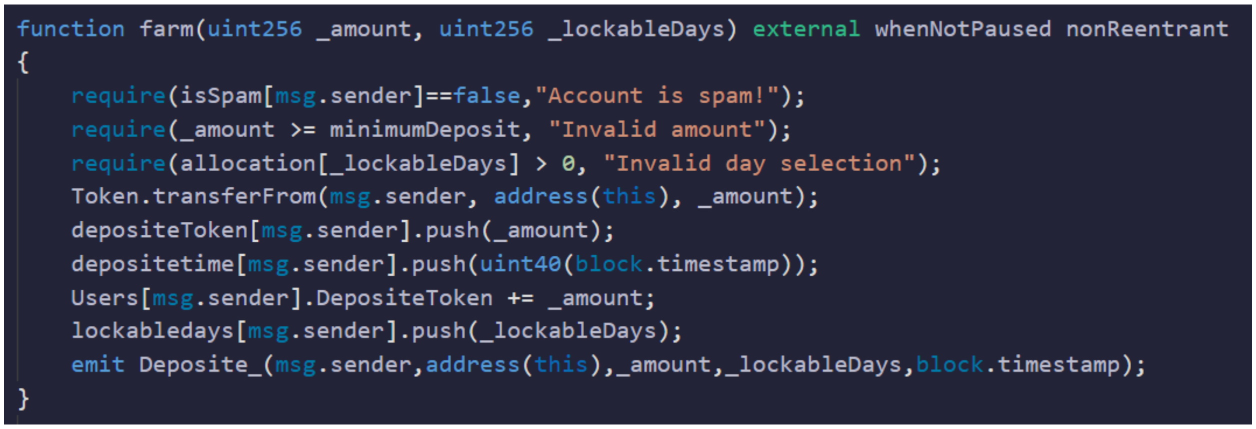
Compiler errors.	Passed
Possible delays in data delivery.	Passed
Timestamp dependence.	Passed
Integer Overflow and Underflow.	Passed
Race Conditions and Reentrancy.	Passed
DoS with Revert.	Passed
DoS with block gas limit.	Passed
Methods execution permissions.	Passed
Economy model of the contract.	Passed
Private user data leaks.	Passed
Malicious Events Log.	Passed
Scoping and Declarations.	Passed
Uninitialized storage pointers.	Passed
Arithmetic accuracy.	Passed
Design Logic.	Passed
Impact of the exchange rate.	Passed
Oracle Calls.	Passed
Cross-function race conditions.	Passed
Fallback function security.	Passed
Safe Open Zeppelin contracts and implementation usage.	Passed

Whitepaper-Website-Contract correlation.	Passed
Front Running.	Passed

Contract's Functions

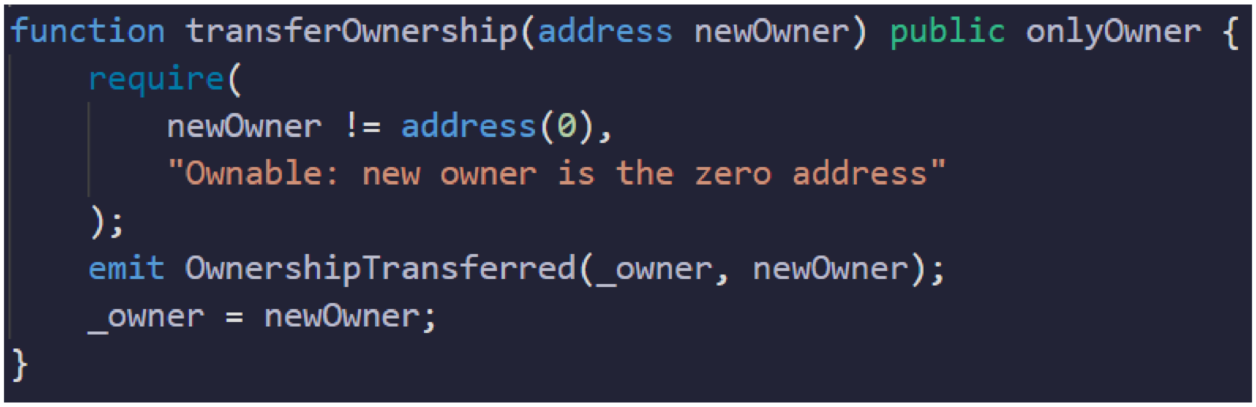
ARCHIE Token Staking Contract

This function is used for staking tokens. User will select the plan of staking. Amount must be greater than minimum deposit amount. Only executable when contract is in not paused. Spam address can't stake.



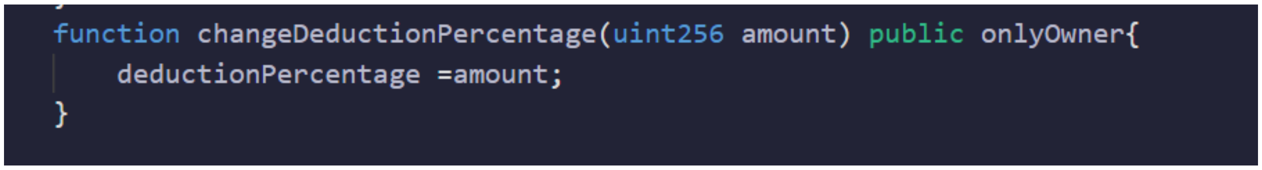
```
function farm(uint256 _amount, uint256 _lockableDays) external whenNotPaused nonReentrant
{
    require(isSpam[msg.sender]==false,"Account is spam!");
    require(_amount >= minimumDeposit, "Invalid amount");
    require(allocation[_lockableDays] > 0, "Invalid day selection");
    Token.transferFrom(msg.sender, address(this), _amount);
    depositeToken[msg.sender].push(_amount);
    depositetime[msg.sender].push(uint40(block.timestamp));
    Users[msg.sender].DepositeToken += _amount;
    lockabledays[msg.sender].push(_lockableDays);
    emit Deposite_(msg.sender,address(this),_amount,_lockableDays,block.timestamp);
}
```

Transfers ownership of the contract to a new account (`newOwner`). Can only be called by the current owner.



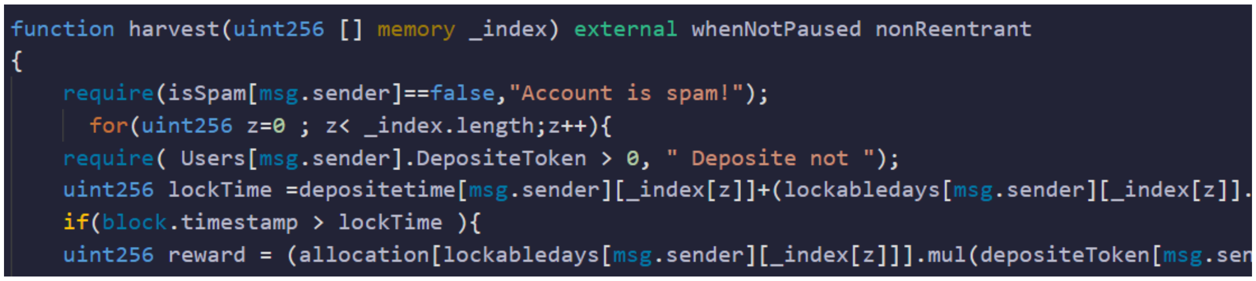
```
function transferOwnership(address newOwner) public onlyOwner {
    require(
        newOwner != address(0),
        "Ownable: new owner is the zero address"
    );
    emit OwnershipTransferred(_owner, newOwner);
    _owner = newOwner;
}
```

This function is used to change deduction percentage. Only owner can call this function.



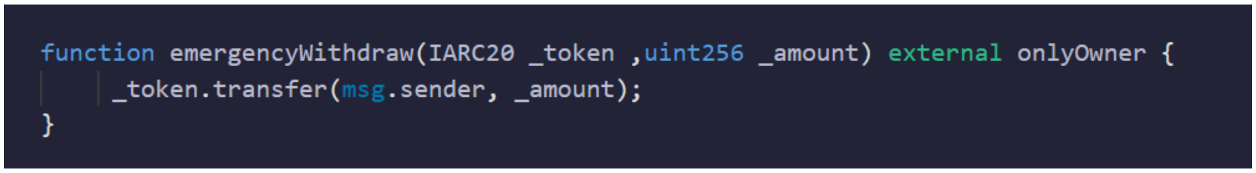
```
function changeDeductionPercentage(uint256 amount) public onlyOwner{
    deductionPercentage =amount;
}
```

This function is used for (harvest) unstake tokens. If user unstake before the unstaking time than 10% will be deducted from its reward. Only executable when contract is in not paused. Spam address can't harvest.



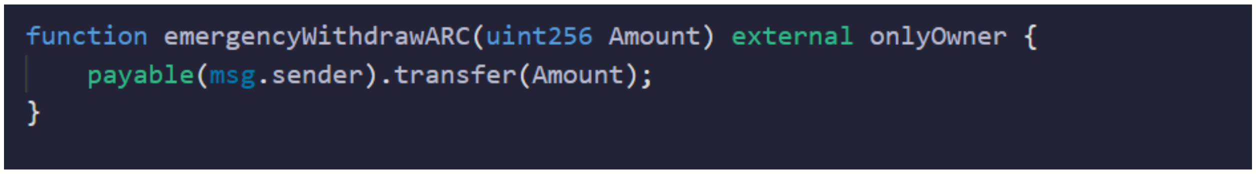
```
function harvest(uint256 [] memory _index) external whenNotPaused nonReentrant
{
    require(isSpam[msg.sender]==false,"Account is spam!");
      for(uint256 z=0 ; z< _index.length;z++){
    require( Users[msg.sender].DepositeToken > 0, " Deposite not ");
    uint256 lockTime =depositetime[msg.sender][_index[z]]+(lockabledays[msg.sender][_index[z]].
    if(block.timestamp > lockTime ){
    uint256 reward = (allocation[lockabledays[msg.sender][_index[z]]].mul(depositeToken[msg.sen
```

This function is used to withdraw tokens from the contract balance. Only owner can call this function.



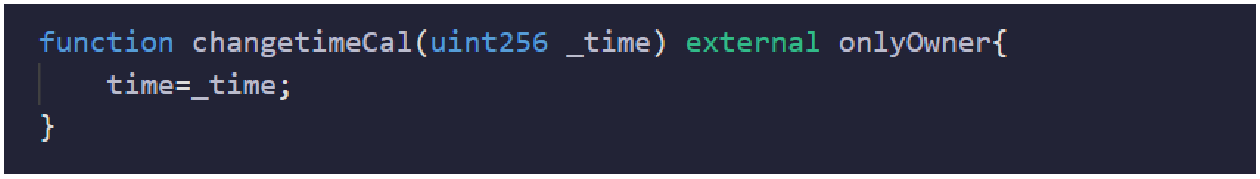
```
function emergencyWithdraw(IARC20 _token ,uint256 _amount) external onlyOwner {
        _token.transfer(msg.sender, _amount);
}
```

This function is used to withdraw ARCHIE Tokens. Only owner can call this function.



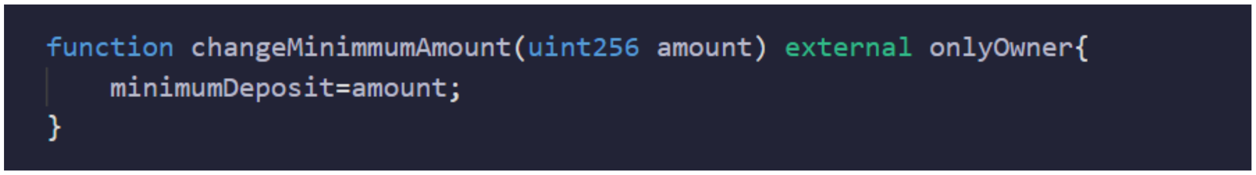
```
function emergencyWithdrawARC(uint256 Amount) external onlyOwner {
    payable(msg.sender).transfer(Amount);
}
```

This function is used to change time. Only owner can call this function.



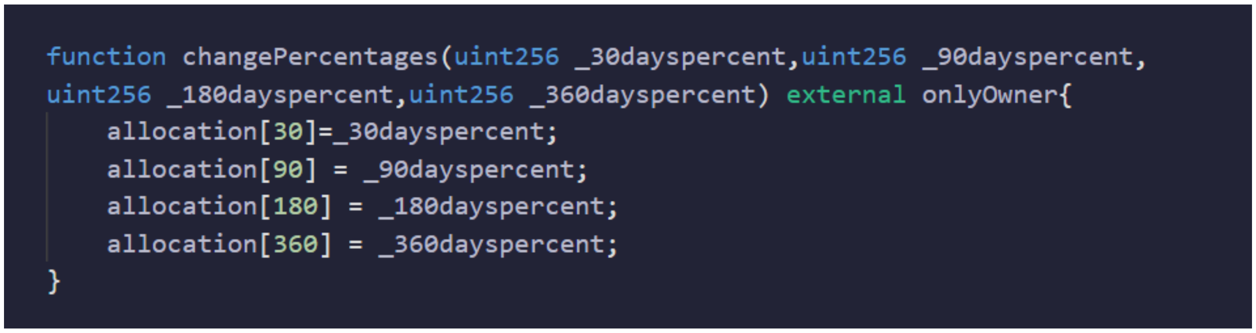
```
function changetimeCal(uint256 _time) external onlyOwner{
    time=_time;
}
```

This function is used to change minimum staking amount. Only owner can call this function.



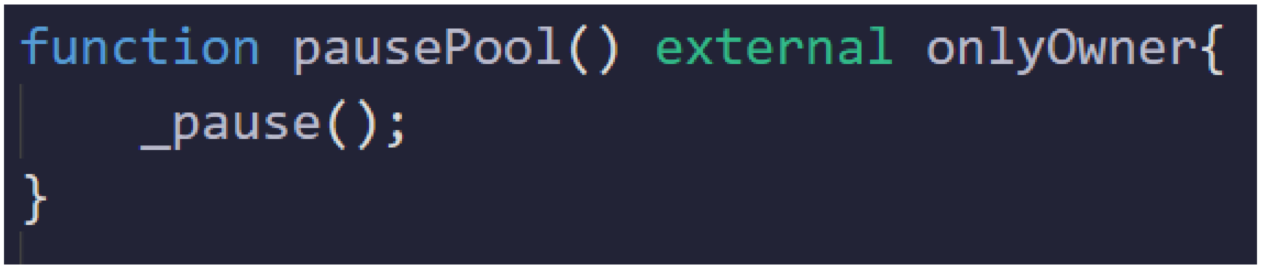
```
function changeMinimmumAmount(uint256 amount) external onlyOwner{
    minimumDeposit=amount;
}
```

This function is used to change percentages of plan wise. Only owner can call this function.



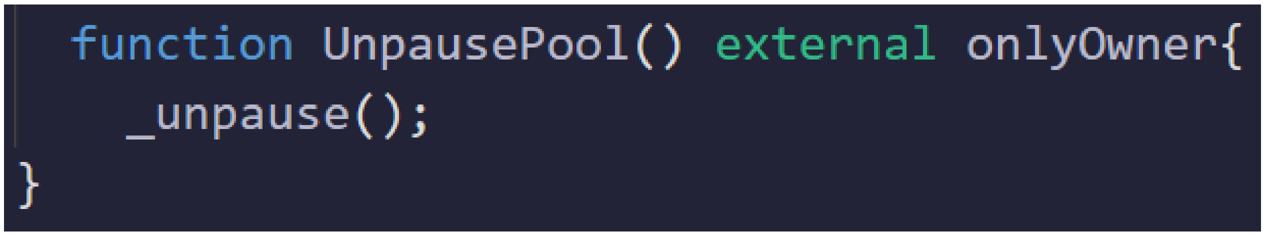
```
function changePercentages(uint256 _30dayspercent,uint256 _90dayspercent,
uint256 _180dayspercent,uint256 _360dayspercent) external onlyOwner{
    allocation[30]=_30dayspercent;
    allocation[90] = _90dayspercent;
    allocation[180] = _180dayspercent;
    allocation[360] = _360dayspercent;
}
```

Owner of this contract can pause the staking pool by executing this function. Only executable by contract owner.



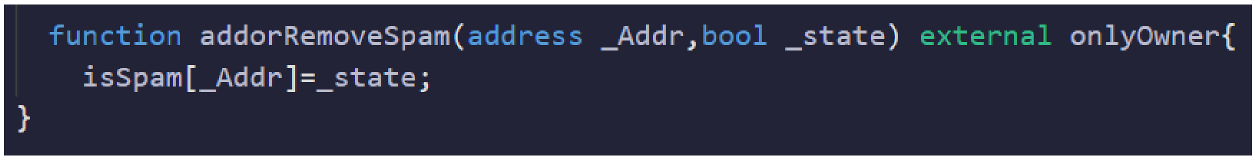
```
function pausePool() external onlyOwner{
    _pause();
}
```

Owner of this contract can un pause the staking pool by executing this function. Only executable by contract owner.



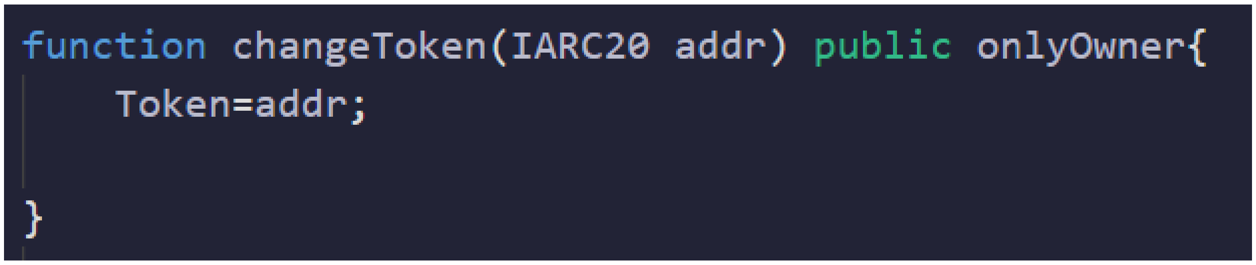
```
  function UnpausePool() external onlyOwner{
     _unpause();
}
```

Owner of this contract can add or remove the address from the spam addresses list.



```
  function addorRemoveSpam(address _Addr,bool _state) external onlyOwner{
    isSpam[_Addr]=_state;
}
```

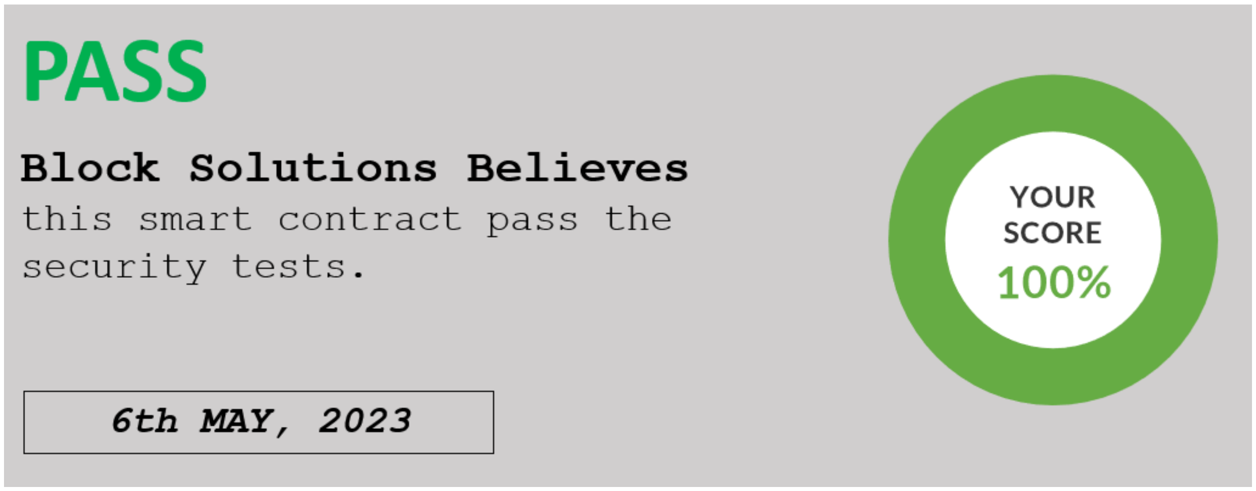
Owner of this contract can change the staking token.



```
function changeToken(IARC20 addr) public onlyOwner{
    Token=addr;

}
```

Testing Summary



PASS

Block Solutions Believes
this smart contract pass the security tests.

6th MAY, 2023

YOUR SCORE 100%

Quick Stats:

Main Category	Subcategory	Result
Contract Programming	Solidity version not specified	Passed
	Solidity version too old	Passed
	Integer overflow/underflow	Passed
	Function input parameters lack of check	Passed
	Function input parameters check bypass	Passed
	Function access control lacks management	Passed
	Critical operation lacks event log	Passed
	Human/contract checks bypass	Passed
	Random number generation/use vulnerability	Passed
	Fallback function misuse	Passed
	Race condition	Passed
	Logical vulnerability	Passed
	Other programming issues	Passed
Code Specification	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	Passed
	High consumption ‘storage’ storage	Passed
	“Out of Gas” Attack	Passed
Business Risk	The maximum limit for mintage not set	Passed
	“Short Address” Attack	Passed
	“Double Spend” Attack	Passed

Overall Audit Result: Passed

Executive Summary

According to the standard audit assessment, Customer`s solidity smart contract is Well-secured. Again, it is recommended to perform an Extensive audit assessment to bring a more assured conclusion.

Insecure
Poor secured
Secure
Well-secured
you are here

We used various tools like 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 Quick Stat section.
We found 0 critical, 0 high, 0 medium and 0 low level issues.

Code Quality

The ARCHIE TOKEN STAKING Smart Contract protocol consists of one smart contract. It has other inherited contracts. These are compact and well written contracts. Libraries used in ARCHIE TOKEN STAKING Smart Contract are part of its logical algorithm. They are smart contracts which contain 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 protocol. The BLOCKSOLUTIONS team has **not** provided scenario and unit test scripts, which would help to determine the integrity of the code in an automated way.
Overall, the code is not commented. Commenting can provide rich documentation for functions, return variables and more.

Documentation

As mentioned above, it’s recommended to write comments in the smart contract code, so anyone can quickly understand the programming flow as well as complex code logic. We were given a ARCHIE TOKEN STAKING Smart Contract smart contract code in the form of File.

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 even core code blocks are written well and systematically. This smart contract does not interact with other external smart contracts.

Risk Level	Description
Critical	Critical vulnerabilities are usually straightforward to exploit and can lead to token 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

No Medium severity vulnerabilities were found.

Low

No Low severity vulnerabilities were found.

Conclusion

The Smart Contract code passed the audit. We were given a contract code. And we have used all possible tests based on given objects as files. Since possible test cases can be unlimited for such extensive smart contract protocol, hence 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 Quick Stat section of the report. Audit report contains all found security vulnerabilities and other issues in the reviewed code.
Security state of the reviewed contract 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.