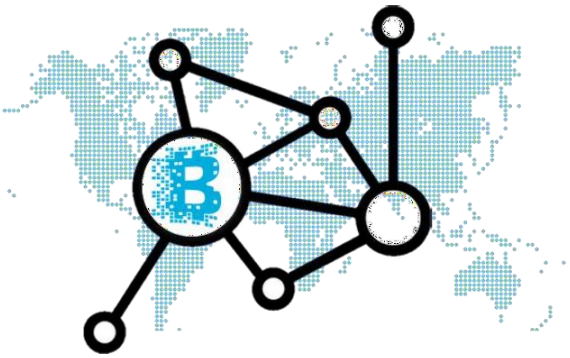
SMART CONTRACT CODE REVIEW AND SECURITY ANALYSIS REPORT FOR Lydian Lion Gold /BEP20 TOKEN SMART CONTRACT

BLOCK SOLUTIONS

Smart Contract Code Review and

Security Analysis Report

For Name: Lydian Lion Gold BEP20 Token



Request Date: 2023-05-20

Completion Date: 2023-05-21

Language: Solidity

SMART CONTRACT CODE REVIEW AND SECURITY ANALYSIS REPORT FOR LYDIAN LION GOLD TOKEN SMART CONTRACT

Audited Project

Block Solutions was commissioned by LYDIAN LION GOLD /BEP20 TOKEN 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.

SMART CONTRACT CODE REVIEW AND SECURITY ANALYSIS REPORT FOR LYDIAN LION GOLD BEP20 TOKEN SMART CONTRACT

Contract Functions

Executable

1. function decimals() external view returns (uint8)
2. function symbol() external view returns (string memory)
3. function name() external view returns (string memory)
4. function totalSupply() external view returns (uint256)
5. function balanceOf(address account) external view returns (uint256)
6. function transfer(address recipient, uint256 amount) public returns (bool)
7. function transferFrom(address sender, address recipient, uint256 amount) public returns (bool)
8. function allowance(address owner, address spender) external view returns (uint256)
9. function approve(address spender, uint256 amount) public returns (bool)
10. function increaseAllowance(address spender, uint256 addedValue) public returns (bool)
11. function decreaseAllowance(address spender, uint256 subtractedValue) public returns (bool)
12. function _transfer(address sender, address recipient, uint256 amount) internal
13. function mint(uint256 amount) public onlyOwner
14. function _approve(address owner, address spender, uint256 amount) internal
15. function _renounceOwnership() public onlyOwner
16. function _transferOwnerShip(address newOwner) public onlyOwner

SMART CONTRACT CODE REVIEW AND SECURITY ANALYSIS REPORT FOR LYDIAN LION GOLD BEP20 TOKEN SMART CONTRACT

Checklist

Compiler Errors	Passed
Possible delays in data delivery	Passed
Timestemp 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 on the exchange rate	Passed
Oracle calls	Passed
Cross-function race conditions	Passed
Fallback function security	Passed
Safe inherited contracts and implementation usage	Passed
Whitepaper-Website-Contract correlation	Not checked
Front running	Not checked

SMART CONTRACT CODE REVIEW AND SECURITY ANALYSIS REPORT FOR LYDIAN LION GOLD BEP20 TOKEN SMART CONTRACT

Executable Functions

BEP20 Token Smart contract

Transfer ownership of the contract to a new account (‘newOwner’).Can only be called by authorized address.

```
function transferOwnership(address newOwner) public onlyOwner {
  _transferOwnership(newOwner);
}
```

Function will transfer token to a specified address recipient is the address to transfer. Only those addresses allowed to call this function that are not blacklisted. ”amount” is the amount to transfer

```
function transfer(address recipient, uint256 amount) external returns (bool) {
  _transfer(_msgSender(), recipient, amount);
  return true;
}
```

Approve the passed address to spend the specified number of token on behalf of msg. sender. “spender” is the address which will spend the funds. ”amount” is the number of tokens to be spent. Only those addresses allowed to call this function that are not blacklisted.

```
function approve(address spender, uint256 amount) external returns (bool) {
  _approve(_msgSender(), spender, amount);
  return true;
}
```

Transfer tokens from the “sender” account to the “recipient” account. The calling account must already have sufficient tokens approved from spending from the “sender” account and “sender” account must have sufficient balance to transfer. Only those addresses allowed to call this function that are not blacklisted.

```
function transferFrom(address sender, address recipient, uint256 amount) external returns (bool) {
  _transfer(sender, recipient, amount);
  _approve(sender, _msgSender(), _allowances[sender][_msgSender()].sub(amount, "BEP20: transfer amount exceeds allowance"));
  return true;
}
```

SMART CONTRACT CODE REVIEW AND SECURITY ANALYSIS REPORT FOR LYDIAN LION GOLD BEP20 TOKEN SMART CONTRACT

Leaves the contract without owner. It will not be possible to call 'onlyOwner` functions anymore. Can only be called by the current owner. Renouncing ownership will leave the contract without an owner, thereby removing any functionality that is only available to the owner.

```
function renounceOwnership() public onlyOwner {
  emit OwnershipTransferred(_owner, address(0));
  _owner = address(0);
}
```

This will increase approval number of tokens to spender address. "_spender" is the address whose allowance will increase and "addedValue" are number of tokens which are going to be added in current allowance. approve should be called when allowed[_spender] == 0. To increment allowed is better to use this function to avoid 2 calls (and wait until the first transaction is mined).

```
function increaseAllowance(address spender, uint256 addedValue) public returns (bool) {
  _approve(_msgSender(), spender, _allowances[_msgSender()][spender].add(addedValue));
  return true;
}
```

This will decrease approval number of tokens to spender address. “_spender" is the address Whose allowance will decrease and "subtractedValue" are number of tokens which are going to be subtracted from current allowance.

```
function decreaseAllowance(address spender, uint256 subtractedValue) public returns (bool) {
  _approve(_msgSender(), spender, _allowances[_msgSender()][spender].sub(subtractedValue, "BEP20: decreased allowance below zero"));
  return true;
}
```

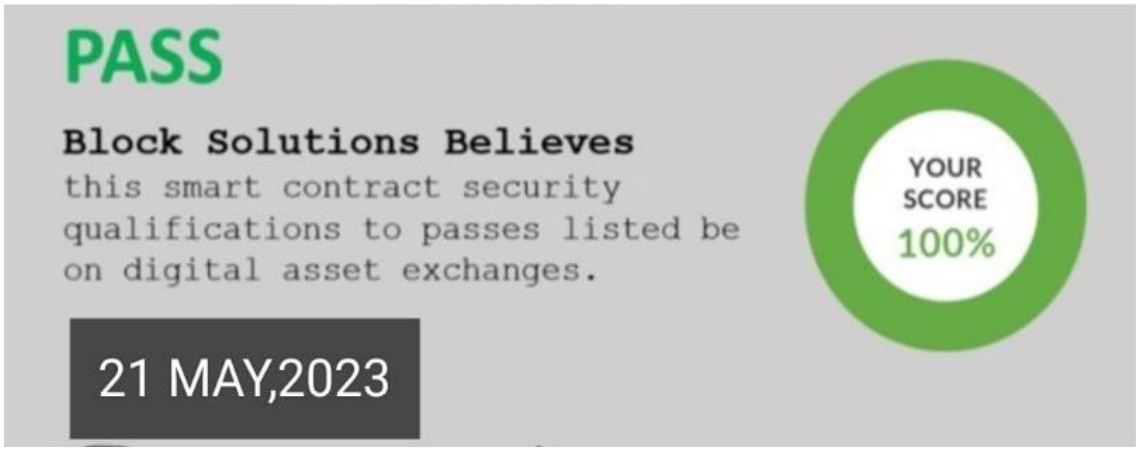
In This function the owner allows to the spender to spend their tokens.

```
function allowance(address owner, address spender) external view returns (uint256) {
  return _allowances[owner][spender];
}
```

In This function the owner can mint the token:

```
 */
function mint(uint256 amount) public onlyOwner returns (bool) {
  _mint(_msgSender(), amount);
  return true;
}
```

Testing Summary



Quick Status

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

SMART CONTRACT CODE REVIEW AND SECURITY ANALYSIS REPORT FOR LYDIAN LION GOLD BEP20 TOKEN SMART CONTRACT

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.

DETECTED VULNERABILITIES

HIGH	MEDIUM	LOW
0	0	0

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 critical, 0 high, 0 medium and 0 low level issues.

SMART CONTRACT CODE REVIEW AND SECURITY ANALYSIS REPORT FOR LYDIAN LION GOLD BEP20 TOKEN SMART CONTRACT

Code Quality:

Bep20 TOKEN Smart Contract protocol consists of one smart contract. It has other inherited contracts like Context, IBEP20, Ownable, Blacklistable. These are compact and well written contracts.
Libraries used in Bep20 TOKEN Smart Contract are part of its logical algorithm. They are smart contracts which contain reusable code. Once deployed on the blockchain (only once), Our 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 LYDIAN LION GOLD Bep20 TOKEN
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. The 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 successfully with some considerations to take. There were no warnings raised. We were given a contract code. And we have used all possible tests based on given objects as files. So, it is good to go for production. 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 was 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.

SMART CONTRACT CODE REVIEW AND SECURITY ANALYSIS REPORT FOR LYDIAN LION GOLD BEP20 TOKEN SMART CONTRACT

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.