BLOCK SOLUTIONS

Smart Contract Code Review and Security Analysis Report for PULSEDOGECOIN ERC20 Token Smart Contract

Request Date: 2022-04-24
Completion Date: 2022-04-26
Language: Solidity

Contents

Commission 3
PULSEDOGECOIN ERC20 TOKEN Properties 4
Contract Functions 5
Executables 5
Checklist 6
Owner privileges 8
PULSEDOGECOIN ERC20 TOKEN Contract 8
Testing Summary 11
Quick Stats: 12
Executive Summary 13
Code Quality 13
Documentation 13
Use of Dependencies 13
Critical 14
High 14
Medium 14
Low 14
Conclusion 15
Our Methodology 15

Commission

Audited Project	PULSEDOGECOIN ERC20 Token Smart Contract
Contract Creator	0xC4e2CA254E8d502AeB9D29C9FE07B26029A814c1
Contract Address	0x34F0915a5f15a66Eba86F6a58bE1A471FB7836A7
Blockchain Platform	Ethereum Mainnet

Block Solutions was commissioned by PULSEDOGECOIN ERC20 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.

PULSEDOGECOIN
ERC20 TOKEN Properties

Contract Token name	PulseDogecoin
Symbol	PLSD
Decimals	12
Total Supply	1701209
Holders	4,419
Transfers	9,466
Number of Claims	5899
Current Day	6
Launch Time	Wednesday, April 20, 2022 2:19:02 PM
Benevolent Address	0x7686640F09123394Cd8Dc3032e9927767aD89344
Hex Origin Address	0x9A6a414D6F3497c05E3b1De90520765fA1E07c03
Merkle_Tree_Root	0x8f4e1c18aa0323d567b9abc6cf64f9626e82ef1b41a404b3f48bfa92eecb9142
Contract Creator	0xC4e2CA254E8d502AeB9D29C9FE07B26029A814c1
Contract Address	0x34F0915a5f15a66Eba86F6a58bE1A471FB7836A7
Blockchain Platform	Ethereum Mainnet

Contract Functions

Executables

i. function approve(address spender, uint256 amount) public virtual override returns (bool)
ii. function claim(address to, uint256 amount, bytes32[] calldata proof) external
iii. function decreaseAllowance(address spender, uint256 subtractedValue) public virtual returns (bool)
iv. function increaseAllowance(address spender, uint256 addedValue) public virtual returns (bool)
v. function mintOaBaTokens() external
vi. function transfer(address recipient, uint256 amount) public virtual override returns (bool)
vii. function transferFrom(address sender,address recipient, uint256 amount) public virtual override returns (bool)

Checklist

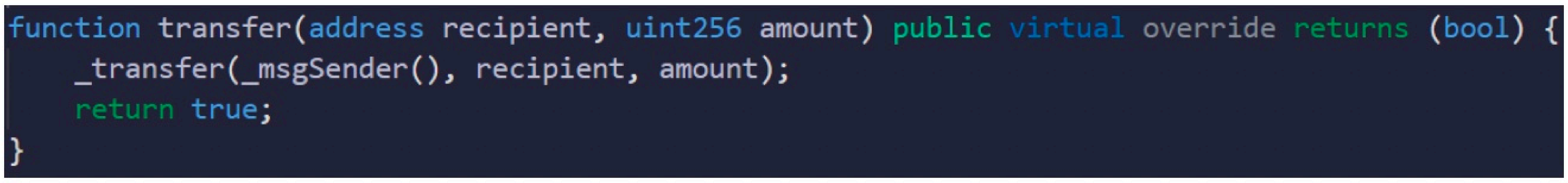
Compiler errors.	Passed
Possible delays in data delivery.	Passed
Timestamp dependence.	Passed
Integer Overflow and Underflow.	Passed
Race Conditions and Reentrancy.	Passed
DoS with Revert.	Passed
DoS with block gas limit.	Passed
Methods execution permissions.	Passed
Economy model of the contract.	Passed
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.	Not-Checked
Front Running.	Not-Checked

Owner privileges

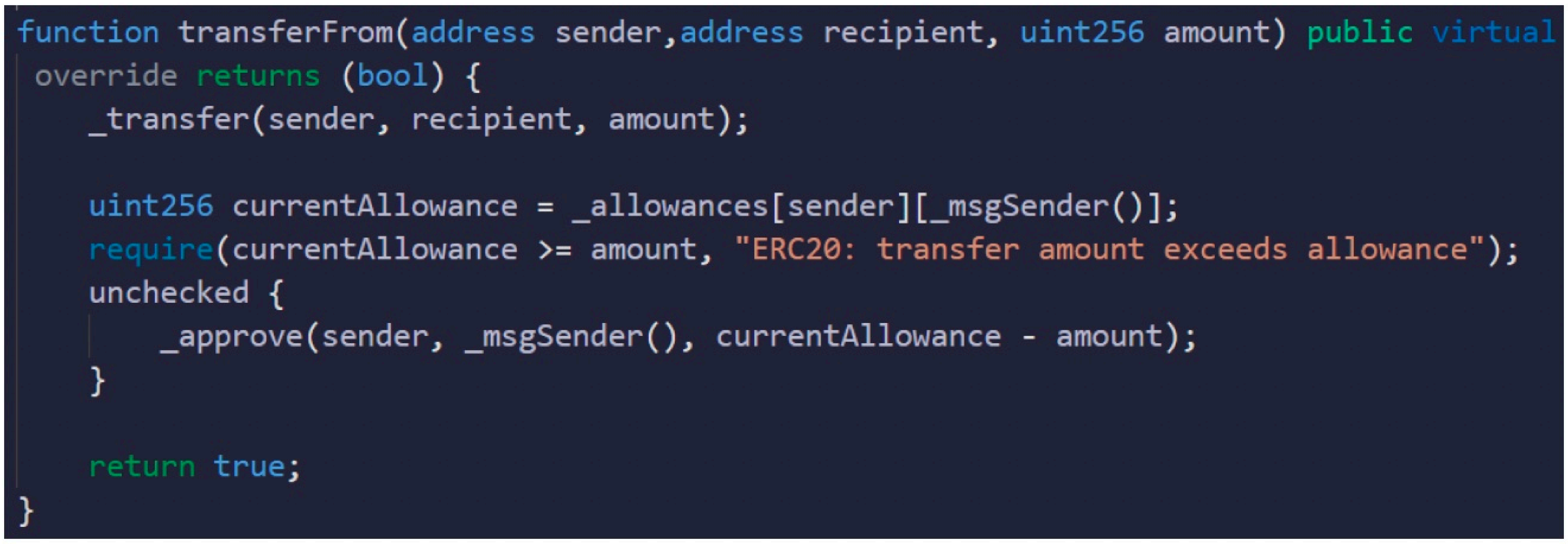
PULSEDOGECOIN ERC20 TOKEN Contract

function will transfer token for a specified address recipient is the address to transfer. “amount” is the amount to be transferred.



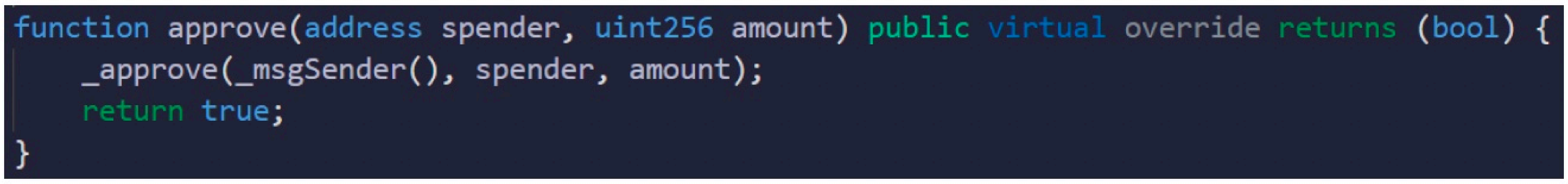
```
function transfer(address recipient, uint256 amount) public virtual override returns (bool) {
    _transfer(_msgSender(), recipient, amount);
    return true;
}
```

Transfer tokens from the “from” account to the “to” account. The calling account must already have sufficient tokens approved for spending from the “from” account and “From” account must have sufficient balance to transfer.” Spender” must have sufficient allowance to transfer.



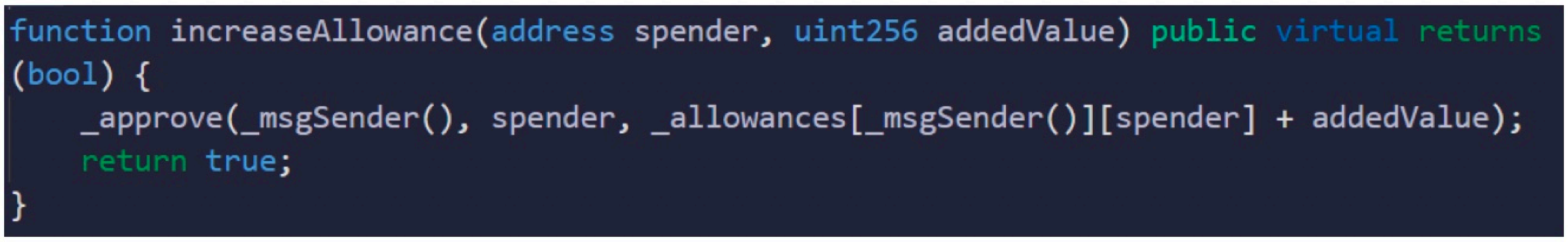
```
function transferFrom(address sender,address recipient, uint256 amount) public virtual
 override returns (bool) {
    _transfer(sender, recipient, amount);

    uint256 currentAllowance = _allowances[sender][_msgSender()];
    require(currentAllowance >= amount, "ERC20: transfer amount exceeds allowance");
    unchecked {
        _approve(sender, _msgSender(), currentAllowance - amount);
    }

    return true;
}
```

Approve the passed address to spend the specified number of tokens on behalf of msg. sender. “spender” is the address which will spend the funds. “tokens” the number of tokens to be spent.



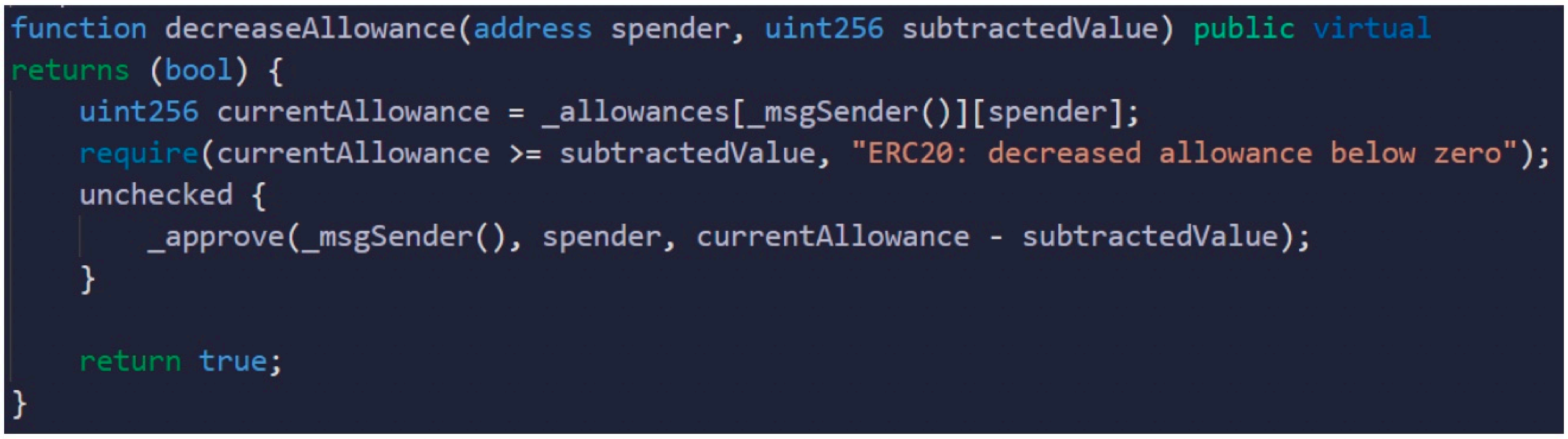
```
function approve(address spender, uint256 amount) public virtual override returns (bool) {
    _approve(_msgSender(), spender, amount);
    return true;
}
```

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 virtual returns
(bool) {
    _approve(_msgSender(), spender, _allowances[_msgSender()][spender] + addedValue);
    return true;
}
```

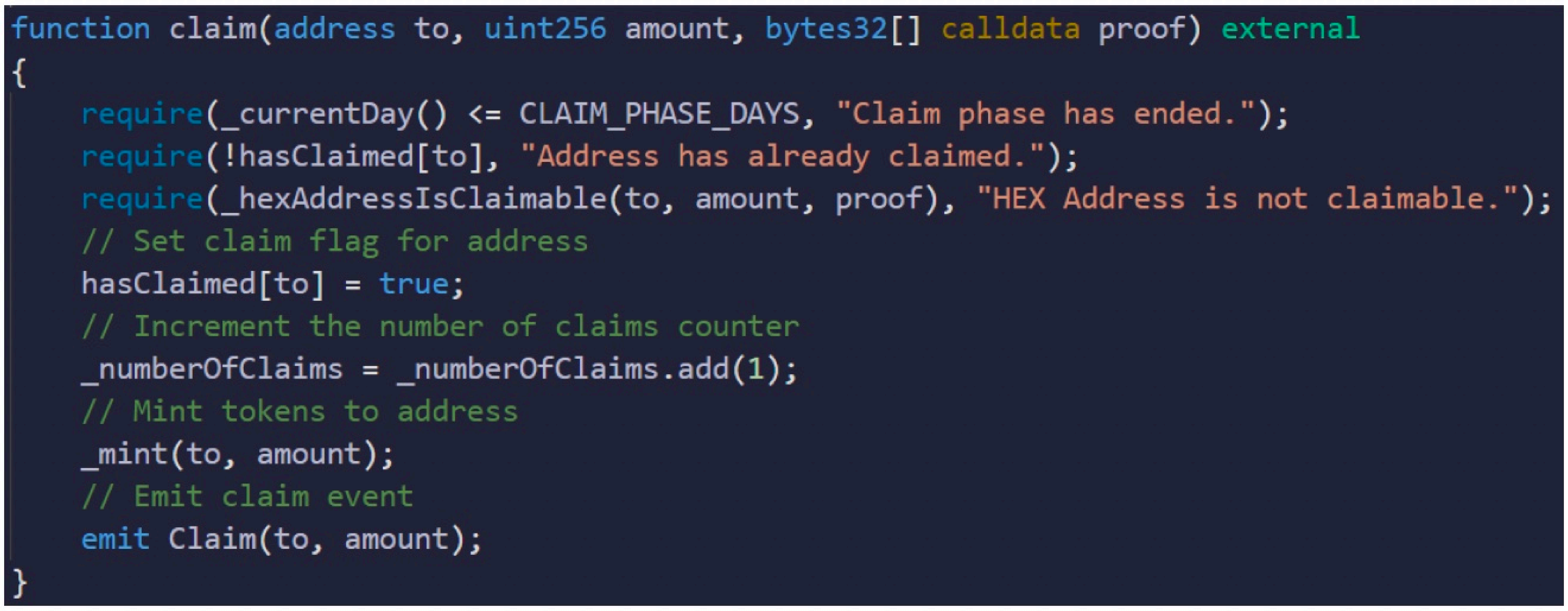
This will decrease approval number of tokens to spender address. "_spender" is the address whose allowance will decrease and "_subtractedValue" are number of tokens which are going to be subtracted from current allowance.



```
function decreaseAllowance(address spender, uint256 subtractedValue) public virtual
returns (bool) {
    uint256 currentAllowance = _allowances[_msgSender()][spender];
    require(currentAllowance >= subtractedValue, "ERC20: decreased allowance below zero");
    unchecked {
        _approve(_msgSender(), spender, currentAllowance - subtractedValue);
    }

    return true;
}
```

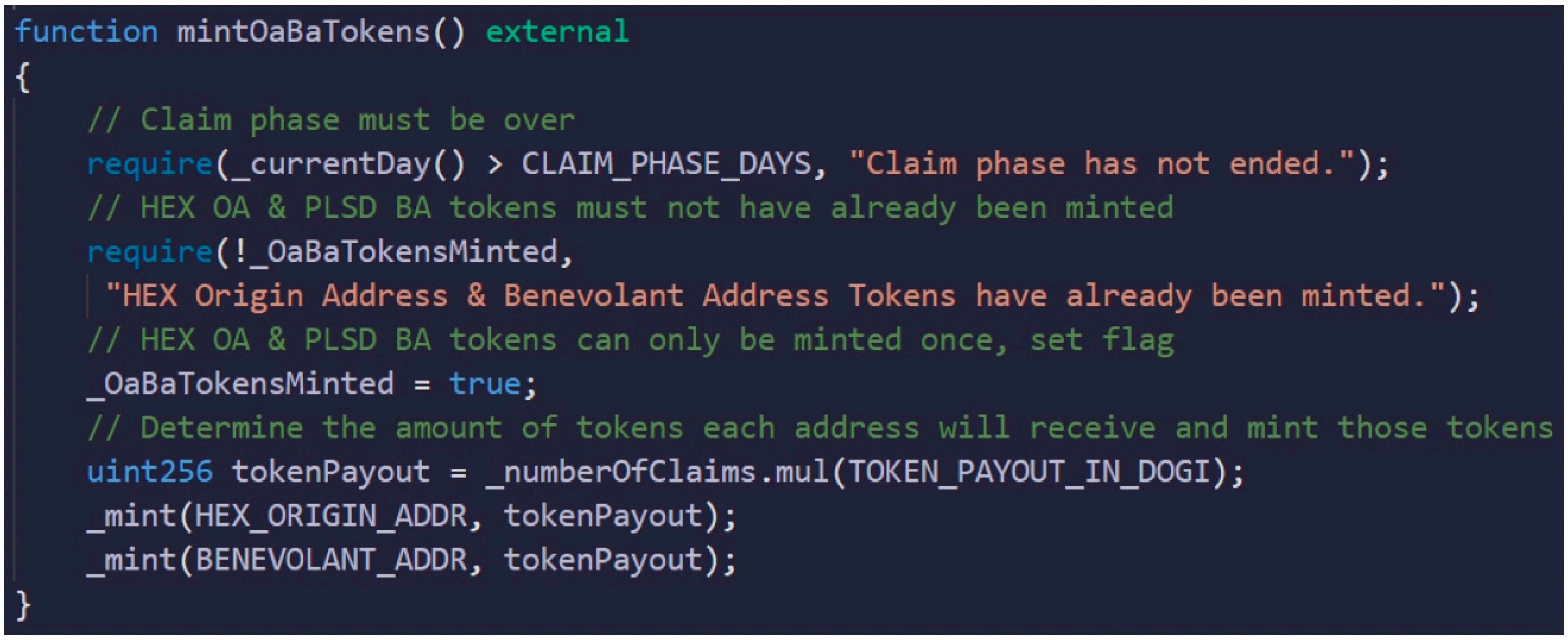
External function to claim airdrop tokens. Must be before the end of the claim phase. Tokens can only be minted once per unique address. The address must be within the airdrop set.

- "to" is the HEX staker address.
- "amount" is the PLSD token amount
- "proof" is the Merkle tree proof



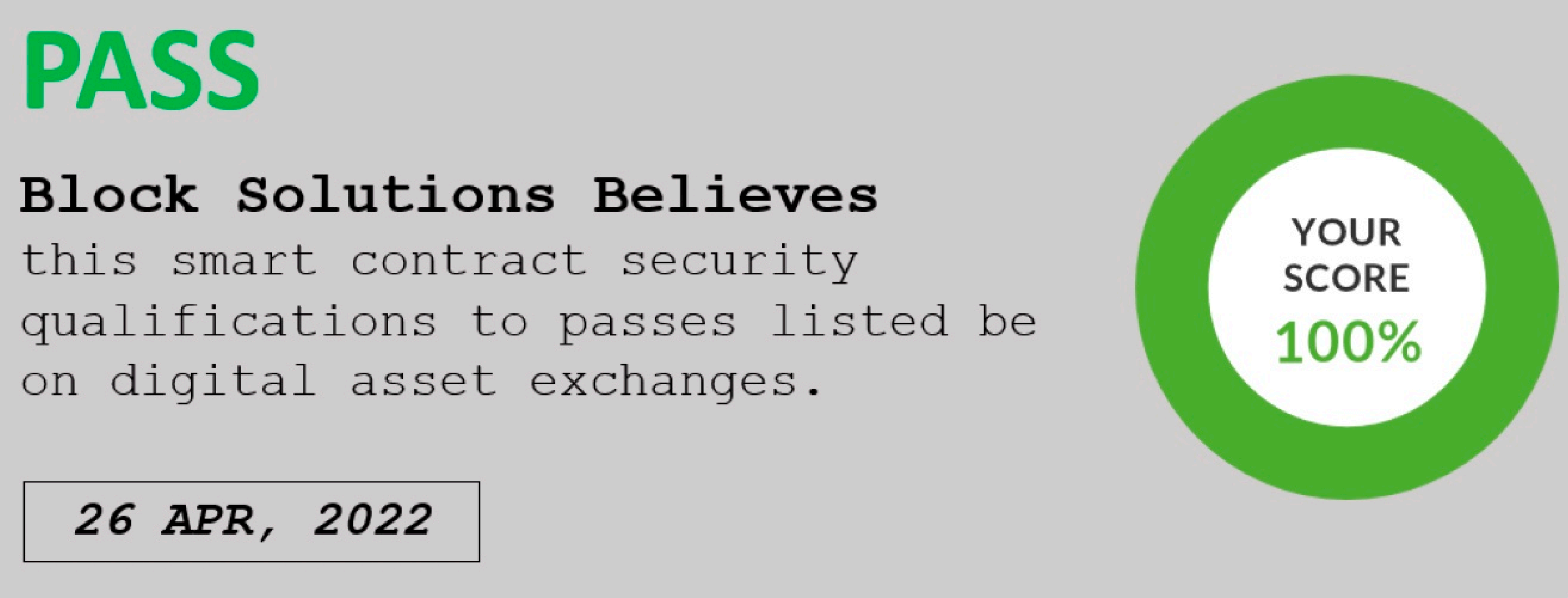
```
function claim(address to, uint256 amount, bytes32[] calldata proof) external
{
    require(_currentDay() <= CLAIM_PHASE_DAYS, "Claim phase has ended.");
    require(!hasClaimed[to], "Address has already claimed.");
    require(_hexAddressIsClaimable(to, amount, proof), "HEX Address is not claimable.");
    // Set claim flag for address
    hasClaimed[to] = true;
    // Increment the number of claims counter
    _numberOfClaims = _numberOfClaims.add(1);
    // Mint tokens to address
    _mint(to, amount);
    // Emit claim event
    emit Claim(to, amount);
}
```

Mint token on HEX Origin & PLSD Benevolent Address. Must be after claim phase has ended. Tokens can only be minted once. Determine the number of tokens each address will receive and mint those tokens.



```
function mintOaBaTokens() external
{
    // Claim phase must be over
    require(_currentDay() > CLAIM_PHASE_DAYS, "Claim phase has not ended.");
    // HEX OA & PLSD BA tokens must not have already been minted
    require(!_OaBaTokensMinted,
     "HEX Origin Address & Benevolant Address Tokens have already been minted.");
    // HEX OA & PLSD BA tokens can only be minted once, set flag
    _OaBaTokensMinted = true;
    // Determine the amount of tokens each address will receive and mint those tokens
    uint256 tokenPayout = _numberOfClaims.mul(TOKEN_PAYOUT_IN_DOGI);
    _mint(HEX_ORIGIN_ADDR, tokenPayout);
    _mint(BENEVOLANT_ADDR, tokenPayout);
}
```

Testing Summary



PASS

Block Solutions Believes
this smart contract security
qualifications to passes listed be
on digital asset exchanges.

26 APR, 2022

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 PULSEDOGECOIN ERC20 TOKEN Smart Contract protocol consists of one smart contract. 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 PULSEDOGECOIN ERC20 TOKEN 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 successfully.
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 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.