ARBITECH
SOLUTIONS

Bamboo ERC20 Token Smart Contract Code Review and Security Analysis Report

Arbitech Solutions

23 November, 2021

Contents

Commission..3

Disclaimer..4

BAMBOO Properties..5

Contract Functions..6

View..6

Executables..6

Owner Executables..6

Checklist..7

Owner privileges..8

BAMBOO Contract..8

Quick Stats:..12

Executive Summary..13

Code Quality..13

Documentation..14

Use of Dependencies..14

Audit Findings..15

Critical..15

High..15

Medium..15

Low..15

Conclusion..16

Our Methodology..17

Disclaimers..18

Privacy Arbitech Solutions Disclaimer..18

Technical Disclaimer..18

Commission

Audited Project	**BAMBOO Token**
Contract Owner	**0xeda7c5543585900b129887ea1f3596b255275554**
Smart Contract	**0x216D111e913874bD385518458C17B084f68bb500**
Blockchain	**Ropsten Testnet Network**

Arbitech Solutions was commissioned by BAMBOO ERC20 Token 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.

Disclaimer

This is a limited report on our finding 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 you 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 you 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 you to conduct your own independent investigations before making any decisions. We go into more detail on this in the disclaimer below please make sure to read it in full.

DISCLAIMER: By reading this report or any part of it, you agree to the terms of this disclaimer. If you do not agree to the terms, then please immediately cease reading this report, and delete and destroy any and all copies of this report downloaded and/or printed by you. This report is provided for information purposes only and on a non-reliance basis, and does not constitute investment advice. No one shall have any right to rely on the report or its contents, and Arbitech Solutions and its affiliates (including holding companies, shareholders, subsidiaries, employees, directors, officers and other representatives) (Arbitech Solutions) owe no duty of care towards you or any other person, nor does Arbitech Solutions make 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 Arbitech Solutions 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 effect in relation to the report. Except and only to the extent that it is prohibited by law, Arbitech Solutions hereby excludes all liability and responsibility, and neither you nor any other person shall have any claim against Arbitech Solutions, for any amount or kind of loss or damage that may result to you 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 wh in delict, tort (including without limitation negligence), contract, breach of statutory duty, misrepresentation (wh 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 smart contracts alone. No applications or operations were reviewed for security.

BAMBOO Properties

Contract name	BAMBOO Token
Contract address	0x216D111e913874bD385518458C17B084f68bb500
Total supply	3.05
Token ticker	$BAMBOO
Decimals	18
Token holders	1
Transaction’s count	4
Top 100 holder’s dominance	100%
Mintable	Yes
Burnable	Yes
Contract deployer address	0x0f22F0f1C70b0277dEE7F0FF1ac480CB594Ca450
Contract’s current owner address	0xeda7c5543585900b129887ea1f3596b255275554

Contract Functions

View

i. function name() public view returns (string memory)

ii. function symbol() public view returns (string memory)

iii. function decimals() public view returns (uint8)

iv. function totalSupply() public view override returns (uint256)

v. function balanceOf(address account) public view override returns (uint256)

vi. function allowance(address owner, address spender) public view override returns (uint256)

vii. function getHolderClaimableAmountForTokenId(address _holder, uint256 _tokenId) public view returns(uint256)

viii. function getHolderClaimableTotalAmount(address _holder) public view returns(uint256)

Executables

i. function approve(address spender, uint256 amount) public virtual override returns (bool)

ii. function transfer(address recipient, uint256 amount) public virtual override returns (bool)

iii. function transferFrom(address sender, address recipient, uint256 amount) public virtual override returns (bool)

iv. function increaseAllowance(address spender, uint256 addedValue) public virtual returns (bool)

v. function decreaseAllowance(address spender, uint256 subtractedValue) public virtual returns (bool)

vi. function claim() external

Owner Executables

i. function burn(address _from, uint256 _amount) public onlyOwner

ii. function mint(uint256 _amount)public onlyOwner

iii. function setClaimEndTime(uint256 _time) public onlyOwner

iv. function setClaimPeriod(uint256 _period) public onlyOwner

v. function transferOwnership(address newOwner) public onlyOwner

vi. function setClaimUnitPerPeriod(uint256 _amount) public onlyOwner

vii. function setPandaContractAddress(address _panda_contract_address) public onlyOwner

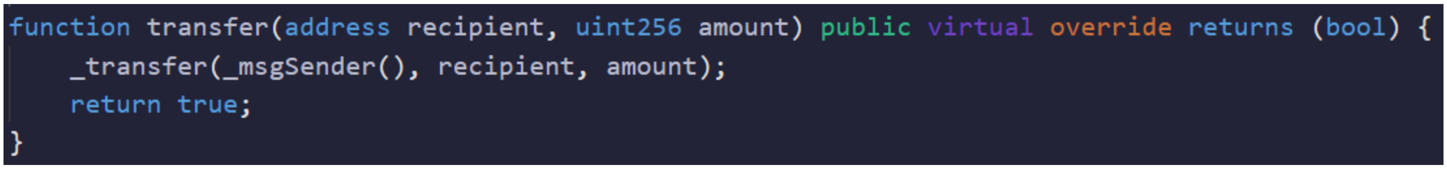
Checklist

Compiler errors.	Passed
Possible delays in data delivery.	Passed
Timestamp dependence.	Low Severity
Integer Overflow and Underflow.	Passed
Race Conditions and Reentrancy.	Passed
DoS with Revert.	Passed
DoS with Arbitech gas limit.	Passed
Methods execution permissions.	Mid Severity
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
Front Running.	Passed
Safe Open Zeppelin contracts and implementation usage.	Passed
Whitepaper-Website-Contract correlation.	Not Checked

Owner privileges

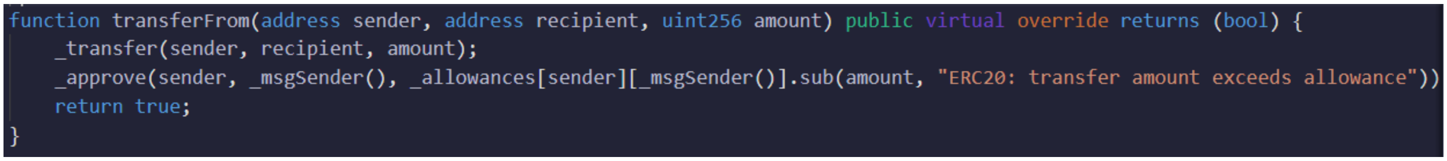
BAMBOO Contract

function will transfer token for a specified address. recipient is the address to transfer' to. amount is the amount to be transferred. Requirements: `recipient` cannot be the zero address. The caller must have a balance of at least `amount`.



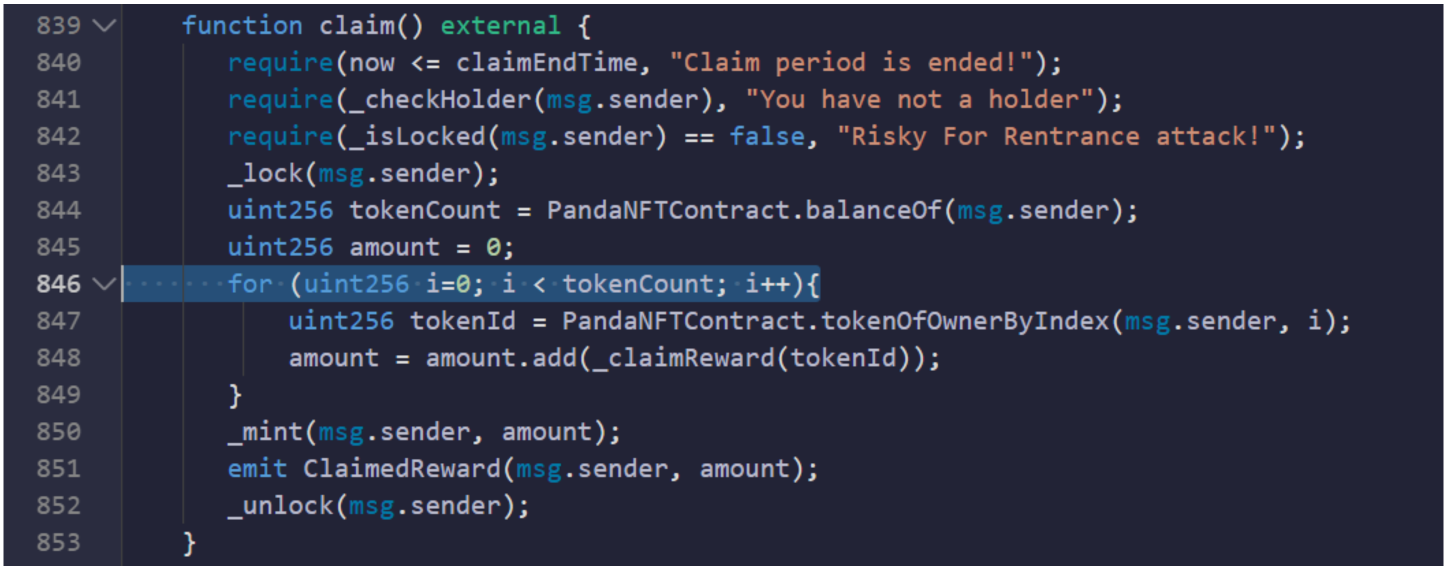
```
function transfer(address recipient, uint256 amount) public virtual override returns (bool) {
    _transfer(_msgSender(), recipient, amount);
    return true;
}
```

Transfer tokens from one address to another. "sender" is the address which you want to send tokens from. "recipient" is the address which you want to transfer to. "amount" is the number of tokens to be transferred. `sender` and `recipient` cannot be the zero address. `sender` must have a balance of at least `amount`. The caller must have allowance for `sender's 'tokens of at least `amount`.



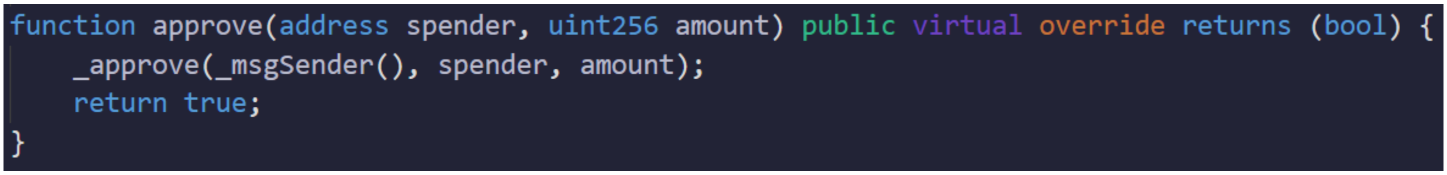
```
function transferFrom(address sender, address recipient, uint256 amount) public virtual override returns (bool) {
    _transfer(sender, recipient, amount);
    _approve(sender, _msgSender(), _allowances[sender][_msgSender()].sub(amount, "ERC20: transfer amount exceeds allowance"))
    return true;
}
```

There are three requirements. One is "Claim period should never ended!" and second one is it checks weither caller is a holder or not. Third requirement is to check weither msg.sender is risky for rentrance attack.This function will check the balance of msg.sender in "PandaNFTContract" and stored it in "tokencount". Then for loop runs and will get all tokenID of msg.sender from "PandaNFTContract" and stored in "tokenID". Then these "tokenID" are passed to "_claimReward" function and reward against each "tokenID" is calculatede and stored in a "amount" variable. Then atlast "_mint" function is called and "amount" is passed to this function and tokens are minted to msg.sender. and "msg.sender" is passed to "_unlock" functions. If a user with more then 50 comes then the for loop gas consumption goes to of 5$ and if user with 1700 NFT comes then this gass fee is more than of 100$. Gas Cost is increasing exponentially with each itteration of loop.



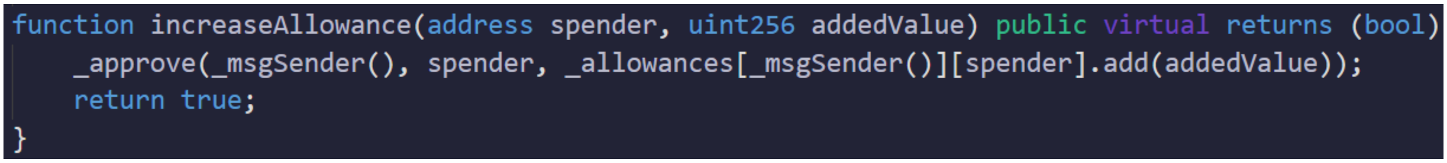
```
    function claim() external {
        require(now <= claimEndTime, "Claim period is ended!");
        require(_checkHolder(msg.sender), "You have not a holder");
        require(_isLocked(msg.sender) == false, "Risky For Rentrance attack!");
        _lock(msg.sender);
        uint256 tokenCount = PandaNFTContract.balanceOf(msg.sender);
        uint256 amount = 0;
        for (uint256 i=0; i < tokenCount; i++){
            uint256 tokenId = PandaNFTContract.tokenOfOwnerByIndex(msg.sender, i);
            amount = amount.add(_claimReward(tokenId));
        }
        _mint(msg.sender, amount);
        emit ClaimedReward(msg.sender, amount);
        _unlock(msg.sender);
    }
```

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. “amount” the number of tokens to be spent.
Beware that changing an allowance with this method brings the risk that someone may use both the old and the new allowance by unfortunate transaction ordering. One possible solution to mitigate this race condition is to first reduce the spender's allowance to 0 and set the desired value afterwards.



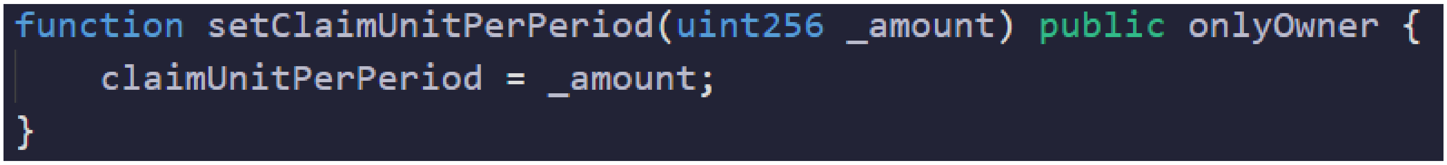
```
function approve(address spender, uint256 amount) public virtual override returns (bool) {
    _approve(_msgSender(), spender, amount);
    return true;
}
```

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 _allowances[spender] == 0. To increment allowed value is better to use this function to avoid 2 calls (and wait until the first transaction is mined) From Bamboo Token. Sol.



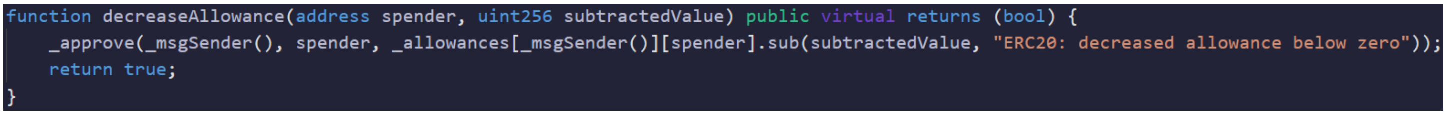
```
function increaseAllowance(address spender, uint256 addedValue) public virtual returns (bool)
    _approve(_msgSender(), spender, _allowances[_msgSender()][spender].add(addedValue));
    return true;
}
```

Owner can only set the claim unit per period amount.



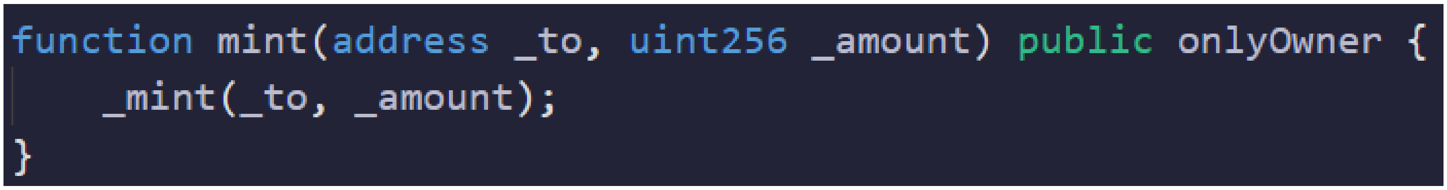
```
function setClaimUnitPerPeriod(uint256 _amount) public onlyOwner {
    claimUnitPerPeriod = _amount;
}
```

Atomically decreases the allowance granted to `spender` by the caller. This is an alternative to {approve} that can be used as a mitigation for problems described in {IERC20-approve}. Emits an {Approval} event indicating the updated allowance.
Requirements: `spender` cannot be the zero address. `spender` must have allowance for the caller of at least `subtractedValue`.



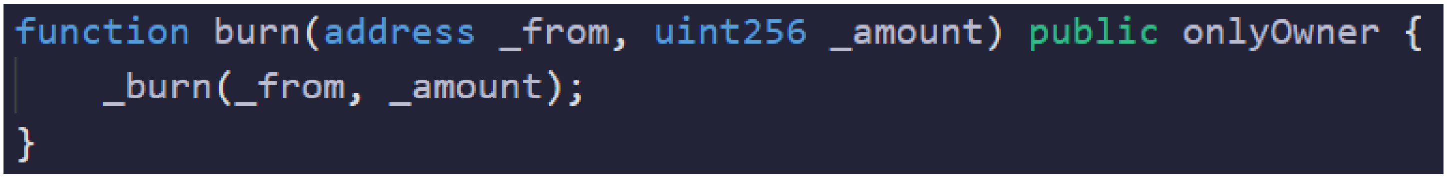
```
function decreaseAllowance(address spender, uint256 subtractedValue) public virtual returns (bool) {
    _approve(_msgSender(), spender, _allowances[_msgSender()][spender].sub(subtractedValue, "ERC20: decreased allowance below zero"));
    return true;
}
```

Creates `_amount` tokens and assigns them to ‘msg.sender `, increasing the total supply. onlyOwner can call this function.



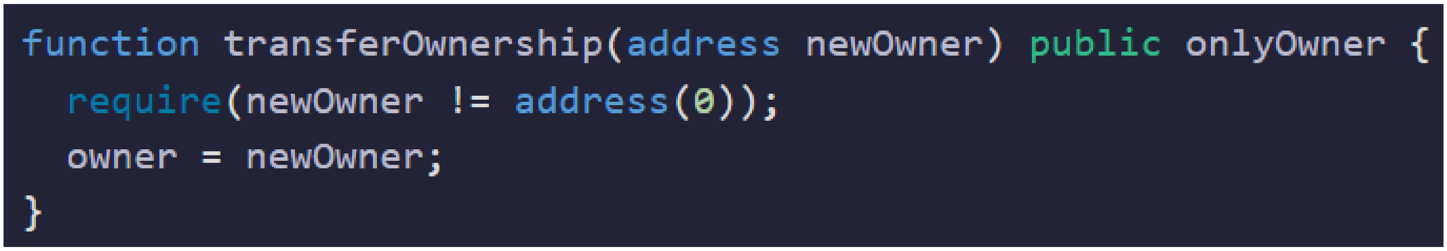
```
function mint(address _to, uint256 _amount) public onlyOwner {
    _mint(_to, _amount);
}
```

Destroys `amount` tokens from `account`, reducing the total supply. `msg. sender` must have at least `amount` tokens. OnlyOwner can call this function an can burn his own tokens.



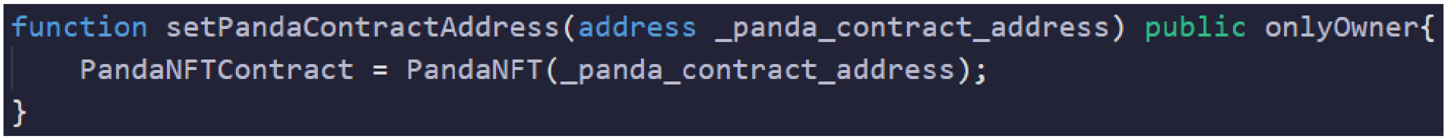
```
function burn(address _from, uint256 _amount) public onlyOwner {
    _burn(_from, _amount);
}
```

Allows the current owner to transfer control of the contract to a newOwner. “newOwner” The address to transfer ownership to.



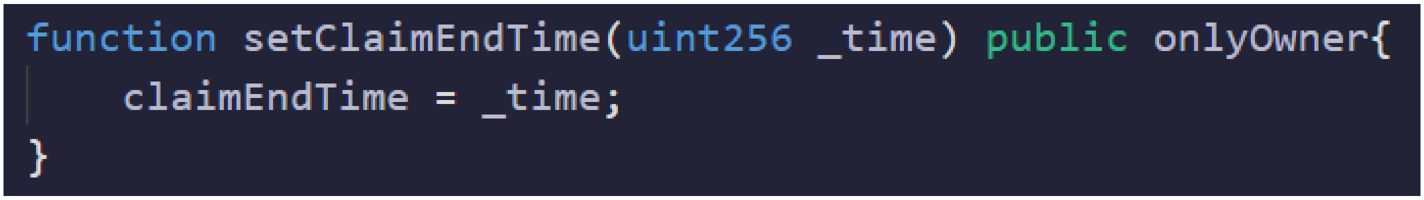
```
function transferOwnership(address newOwner) public onlyOwner {
  require(newOwner != address(0));
  owner = newOwner;
}
```

Owner can set the panda contract address in constructor once smart contract is deployed and later owner of this contract can set the new panda NFT contract address.



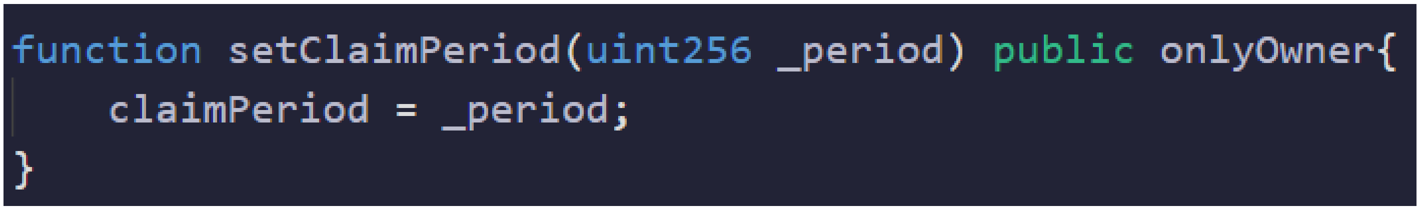
```
function setPandaContractAddress(address _panda_contract_address) public onlyOwner{
    PandaNFTContract = PandaNFT(_panda_contract_address);
}
```

Owner can set the claim end time and can change this at any time.



```
function setClaimEndTime(uint256 _time) public onlyOwner{
    claimEndTime = _time;
}
```

Owner can set the new time period as claim period.



```
function setClaimPeriod(uint256 _period) public onlyOwner{
    claimPeriod = _period;
}
```

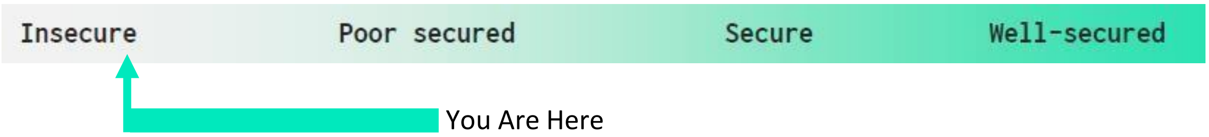
Quick Stats:

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	N/A
	Fallback function misuse	N/A
	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	Fail
	High consumption ‘for/while’ loop	Fail
	High consumption ‘storage’ storage	Fail
	“Out of Gas” Attack	Fail
Business Risk	The maximum limit for mintage not set	Passed
	“Short Address” Attack	Passed
	“Double Spend” Attack	Passed

Overall Audit Result: Failed

Executive Summary

According to the standard audit assessment, Customer`s solidity smart contract is Insecure. Again, it is recommended to perform an Extensive audit assessment to bring a more assured conclusion. **Gas Cost is increasing exponentially with each itteration of loop**



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 1 critical, 0 high, 0 medium and 3 low level issues.

Code Quality

The BAMBOO ERC20 Token protocol consists of one smart contract. It has other inherited contracts like ERC20 and Ownable. These are compact and well written contracts. Libraries used in BAMBOO ERC20 Token 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 ARBITECH SOLUTIONS 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 BAMBOO ERC20 Token smart contract code in the form of File.

Use of Dependencies

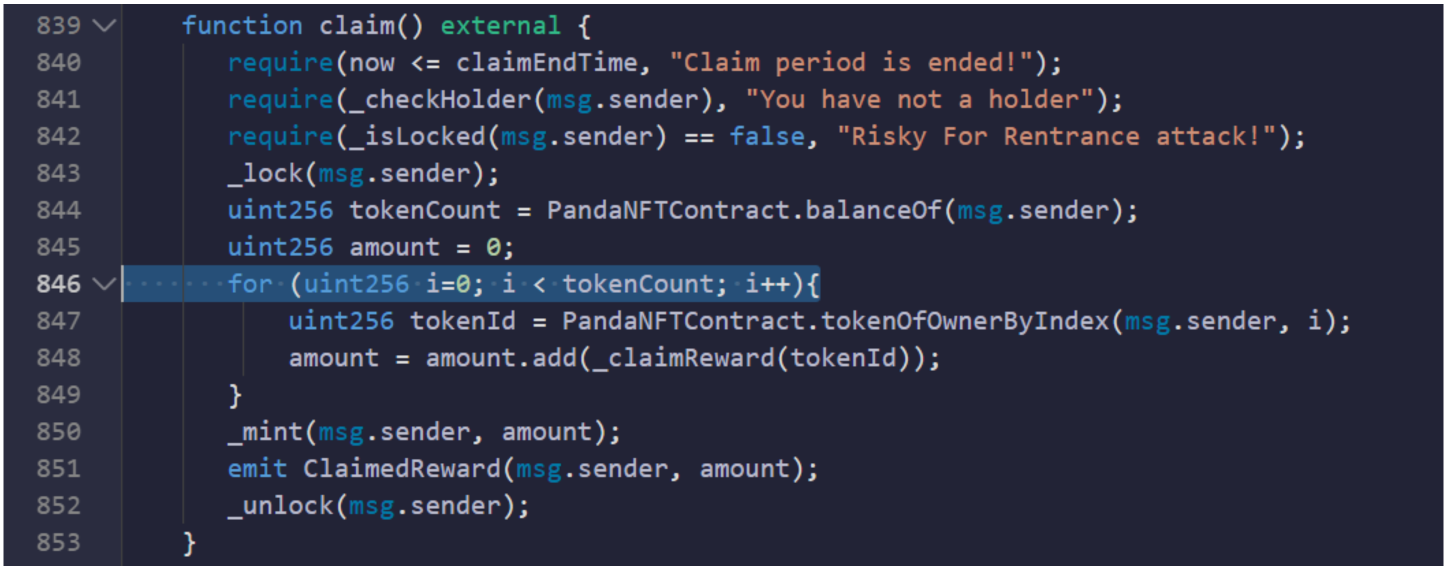
As per our observation, the libraries are used in this smart contract infrastructure that are based on well-known industry standard open-source projects. And even core code is written well and systematically. This smart contract does not interact with other external smart contracts.

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

Gas Cost is increasing exponentially with each iteration of loop. If a user with more than 50 comes then the for-loop gas consumption goes to ether of 5$ and if user with 1700 NFT comes then this gas fee is more than ether of 100$.



```
function claim() external {
    require(now <= claimEndTime, "Claim period is ended!");
    require(_checkHolder(msg.sender), "You have not a holder");
    require(_isLocked(msg.sender) == false, "Risky For Rentrance attack!");
    _lock(msg.sender);
    uint256 tokenCount = PandaNFTContract.balanceOf(msg.sender);
    uint256 amount = 0;
    for (uint256 i=0; i < tokenCount; i++){
        uint256 tokenId = PandaNFTContract.tokenOfOwnerByIndex(msg.sender, i);
        amount = amount.add(_claimReward(tokenId));
    }
    _mint(msg.sender, amount);
    emit ClaimedReward(msg.sender, amount);
    _unlock(msg.sender);
}
```

Solution:

We can overcome this issue by using mapping rather than for loop. Which will leads to decrease in Gas Cost.

High

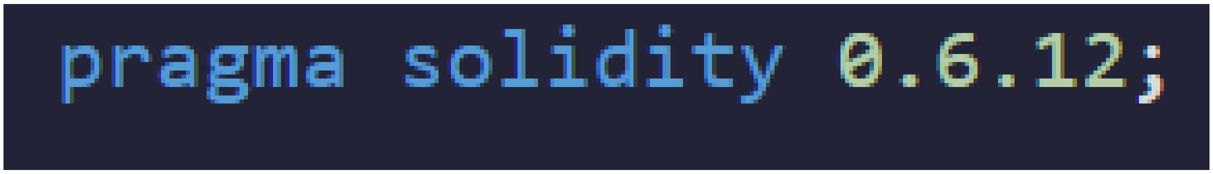
No high severity vulnerabilities were found.

Medium

No Medium severity vulnerabilities were found.

Low

(1) Compiler version can be upgraded.

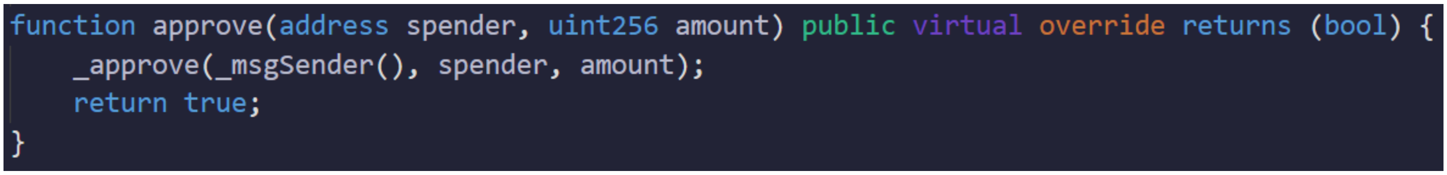


```
pragma solidity 0.6.12;
```

Although this does not raise any security vulnerability, using the latest compiler version can help to prevent any compiler level bugs.

(2) Approve ()

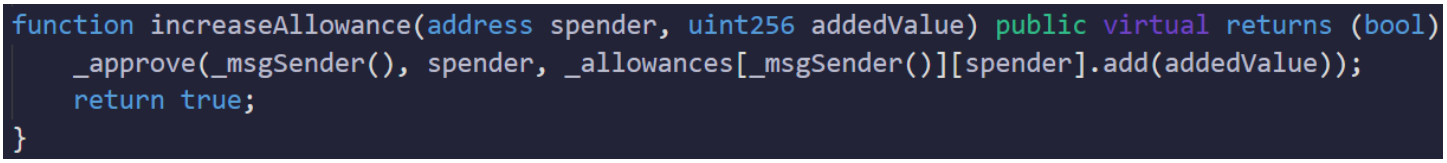
Approve the passed address to spend the specified number of tokens on behalf of msg. sender. “spender” is the address which will spend the funds. “amount” the number of tokens to be spent.
Beware that changing an allowance with this method brings the risk that someone may use both the old and the new allowance by unfortunate transaction ordering. One possible solution to mitigate this race condition is to first reduce the spender's allowance to 0 and set the desired value afterwards.



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

(3) IncreaseAllowance ()

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 _allowances[spender] == 0. To increment allowed value 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].add(addedValue));
    return true;
}
```

Solution: This issue is acknowledged.

Conclusion

The Smart Contract code passed the audit successfully on the Ropsten Testnet Network with some considerations to take. There were one critical and three low severity warnings raised meaning that they should be taken into consideration. The last change is advisable in order to provide more security to new holders. Nonetheless this is not necessary if the holders and/or investors feel confident with the contract owners. 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 "Insecure".

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

Privacy Arbitech Solutions Disclaimer

Arbitech Solutions 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, 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, bug free 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 conducting a bug bounty program to confirm the high level of security of this smart contract.

Technical Disclaimer

Smart contracts are deployed and executed on the 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.