Crypto Jackpot

0xeb78dfd268cee4d50cfa3228b4d112aba7c105bc

Smart Contract Review and Security Audit

By

Elite Security (A product of Elitheum)

PASS

Completion Date: 26th January 2022
Language: Solidity

Contents

Audited Project	Crypto Jackpot BEP20 Token Smart Contract
Contract Address	0xEB78DfD268cEE4d50cFA3228b4d112aBa7C105BC
Contract Owner Wallet Address	0x072d5521b3b47e39aecfe434a15ebbdd41a49f34
Contract Creator Wallet address	0xcE036DF77891eA4BfEf72AcAC74b242BeCba5865

Elite Security was commissioned by CRYPTO JACKPOT 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.

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 Elite Security and its affiliates (including holding companies, shareholders, subsidiaries, employees, directors, officers and other representatives) (Elite Security) owe no duty of care towards you or any other person, nor does Elite Security 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 Elite Security 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, Elite Security hereby excludes all liability and responsibility, and neither you nor any other person shall have any claim against Elite Security, 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 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 smart contracts alone. No applications or operations were reviewed for security.

CRYPTO JACKPOT BEP20 TOKEN

Properties

Contract Token name	Crypto Jackpot
Total Supply	2000000000
Decimals	18
Symbol	CJP
Liquidity Fee	3
Marketing Fee	12
Total Fee	15
Max Sell Transaction Amount	10000000
Max Buy Transaction Amount	10000000
Swap Tokens at Amount	100000
Contract Owner Wallet Address	0x072d5521b3b47e39aecfe434a15ebbdd41a49f34
Contract Creator Wallet Address	0xcE036DF77891eA4BfEf72AcAC74b242BeCba5865
Pancake Router	0x10ED43C718714eb63d5aA57B78B54704E256024E
Pancake Pair	0x77AA2061d12D7ECB0E35663ace2Fabaa7E9741E2

Contract Functions

View

i. function allowance(address owner, address spender) public view virtual override returns (uint256)
ii. function balanceOf(address account) public view virtual override returns (uint256) iii. function decimals() public view virtual override returns (uint8)
iv. function isExcludedFromFees(address account) public view returns(bool)
v. function isExcludedFromMaxTx(address account) public view returns(bool)
vi. function name() public view virtual override returns (string memory)
vii. function owner() public view virtual returns (address)
viii. function symbol() public view virtual override returns (string memory)
ix. function totalSupply() public view virtual override returns (uint256)

Executables

i. function approve(address spender, uint256 amount) public virtual override returns (bool)
ii. function decreaseAllowance(address spender, uint256 subtractedValue) public virtual returns (bool)
iii. function excludeFromFees(address account, bool excluded) public onlyOwner
iv. function excludeMultipleAccountsFromFees(address[] calldata accounts, bool excluded) public onlyOwner
v. function increaseAllowance(address spender, uint256 addedValue) public virtual returns (bool) 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)

Owner Executable:

i. function renounceOwnership() public virtual onlyOwner
ii. function setAutomatedMarketMakerPair(address pair, bool value) public onlyOwner
iii. function setExcludeFromMaxTx(address _address, bool value) public onlyOwner
iv. function setFee(uint256 _liquidityFee, uint256 _marketingFee) public onlyOwner
v. function setMarketingWallet(address payable newAddress) public onlyOwner
vi. function setMaxSellTransaction(uint256 _maxSellTxAmount) public onlyOwner vii. function setMaxBuyTransaction(uint256 _maxBuyTxAmount) public onlyOwner viii. function setSwapTokensAtAmount(uint256 newLimit) public onlyOwner ix. function setSwapAndLiquifyEnabled(bool _enabled) public onlyOwner
x. function transferOwnership(address newOwner) public virtual onlyOwner
xi. function updateUniswapV2Router(address newAddress) 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.	Not Checked
Front Running.	Not Checked

Owner privileges

CRYPTO JACKPOT BEP20 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;
}
```

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

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

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. 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.

https://github.com/ethereum/EIPs/blob/master/EIPS/eip-20-token-standard.md recommends that there are no checks for the approval double-spend attack as this should be implemented in user interfaces.

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

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) {
    uint256 currentAllowance = _allowances[_msgSender()][spender];
    require(currentAllowance >= subtractedValue, "ERC20: decreased allowance below zero");
    unchecked {
        _approve(_msgSender(), spender, currentAllowance - subtractedValue);
    }

    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) .

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

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 virtual onlyOwner {
    _setOwner(address(0));
}
```

Owner of the contract can exclude the account from fees.

```
function excludeFromFees(address account, bool excluded) public onlyOwner {
    require(_isExcludedFromFees[account] != excluded,
    "CJP: Account is already the value of 'excluded'");
    _isExcludedFromFees[account] = excluded;

    emit ExcludeFromFees(account, excluded);
}
```

Owner of the contract can exclude multiple account from fees.

```
function excludeMultipleAccountsFromFees(address[] calldata accounts, bool excluded)
public onlyOwner {
    for(uint256 i = 0; i < accounts.length; i++) {
        _isExcludedFromFees[accounts[i]] = excluded;
    }

    emit ExcludeMultipleAccountsFromFees(accounts, excluded);
}
```

Owner of the contact can set the automated market pair. The PancakeSwap pair cannot be removed from automatedMarketMakerPairs.

```
function setAutomatedMarketMakerPair(address pair, bool value) public onlyOwner {
    require(pair != uniswapV2Pair,
    "CJP: The PancakeSwap pair cannot be removed from automatedMarketMakerPairs");

    _setAutomatedMarketMakerPair(pair, value);
}
```

Owner of the contract can exclude the address from max transaction limit and from fees.

```
function setExcludeFromAll(address _address) public onlyOwner {
    _isExcludedFromMaxTx[_address] = true;
    _isExcludedFromFees[_address] = true;
}
```

Owner of the contract can exclude the address from maximum transaction limit.

```
function setExcludeFromMaxTx(address _address, bool value) public onlyOwner {
    _isExcludedFromMaxTx[_address] = value;
}
```

Owner of the contract can set the new marketing fee and liquidity fee.

```
function setFee(uint256 _liquidityFee, uint256 _marketingFee) public onlyOwner {
    liquidityFee = _liquidityFee;
    marketingFee = _marketingFee;

    totalFees = liquidityFee.add(marketingFee);
}
```

Owner of the contract can set the marketing the address.

```
function setMarketingWallet(address payable newAddress) public onlyOwner {
    marketingWallet = newAddress;
}
```

Owner of the contract can set the maximum buy transaction limit.

```
function setMaxBuyTransaction(uint256 _maxBuyTxAmount) public onlyOwner {
    maxBuyTransactionAmount = _maxBuyTxAmount * 10**18;
}
```

Owner of the contract can set the maximum sell transaction limit.

```
function setMaxSellTransaction(uint256 _maxSellTxAmount) public onlyOwner {
    maxSellTransactionAmount = _maxSellTxAmount * 10**18;
}
```

Owner of the contract can set new limit for swap token at amount.

```
function setSwapTokensAtAmount(uint256 newLimit) public onlyOwner {
    swapTokensAtAmount = newLimit * 10**18;
}
```

Owner of the contract can set the new router address.

```
function updateUniswapV2Router(address newAddress) public onlyOwner {
    require(newAddress != address(uniswapV2Router), "CJP: The router already has that address");
    emit UpdateUniswapV2Router(newAddress, address(uniswapV2Router));
    uniswapV2Router = IUniswapV2Router02(newAddress);
}
```

Testing Summary

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	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.

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 2 low level issues.

Code Quality

The CRYPTO JACKPOT BEP20 TOKEN Smart Contract protocol consists of one smart contract. It has other inherited contracts like ERC20, Ownable. These are compact and well written contracts. Libraries used in CRYPTO JACKPOT 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), it is assigned a specific address and its properties / methods can be reused many times by other contracts in protocol. The Elite Security 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 CRYPTO JACKPOT BEP20 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

(1) 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.

https://github.com/ethereum/EIPs/blob/master/EIPS/eip-20-token-standard.md recommends that there are no checks for the approval double-spend attack as this should be implemented in user interfaces.

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

(2) 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] + addedValue);
    return true;
}
```

Solution: This issue is acknowledged.

Conclusion

The Smart Contract code passed the audit successfully with some considerations to take. There were two low severity warnings raised meaning that they should be taken into consideration but if the confidence in the owner is good, they can be dismissed. 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 "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.

Disclaimers

Privacy Elite Security Disclaimer

The Elite Security 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.