BLOCK SOLUTIONS

Smart Contract Code Review and Security Analysis Report for WESENDIT BEP20 Token Smart Contract

Request Date: 2022-08-21
Completion Date: 2022-08-21
Language: Solidity

Contents

Commission

Audited Project	WESENDIT BEP20 Token Smart Contract
Smart Contract Address	0xcBD1C8823FB8010938A717aF6f9263b8a0104901
Owner Address	0x951dacd3dca4a949b90e16859c0d6add43a12549
Creator Address	0x134fe81f2fDB4A558644a415D44a6a642778FadA
Blockchain Platform	Binance Smart Chain Mainnet

Block Solutions was commissioned by WESENDIT 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.

WESENDIT Properties

Contract Token name	WESENDIT
Total supply	1500000000 WSI
Symbol	WSI
Decimals	18
Cap	1500000000
Holder	36 addresses
Transfers	41
Initial Supply	37500000 WSI
Smart Contract Address	0xcBD1C8823FB8010938A717aF6f9263b8a0104901
Owner Address	0x951dacd3dca4a949b90e16859c0d6add43a12549
Creator Address	0x134fe81f2fDB4A558644a415D44a6a642778FadA
Blockchain Platform	Binance Smart Chain Mainnet

Contract Functions

Executables

i. function approve(address spender, uint256 amount) public virtual override returns (bool)
ii. function burn(uint256 amount) public virtual
iii. function burnFrom(address account, uint256 amount) public virtual
iv. function decreaseAllowance(address spender, uint256 subtractedValue) public virtual returns (bool)
v. function distributeSaleToken(address seedSaleWallet,address privateSaleWallet) external onlyOwner
vi. function distributeToken(address teamWallet,address advisorsWallet,address referralsWallet, address developmentWallet,address marketingWallet,address operationsWallet,address exchangeAndLiquidityWallet, address stakingRewardsWallet,address activityRewardsWallet,address airdropWallet) external onlyOwner
vii. function distributeReserveToken(address generalReserveWallet) external onlyOwner
viii. function emergencyWithdraw(address receiver) external onlyOwner
ix. function increaseAllowance(address spender, uint256 addedValue) public virtual returns (bool)
x. function setActivityPoolAddress(address addr) external onlyOwner
xi. function setPancakeRouter(address addr) external onlyOwner
xii. function setReferralPoolAddress(address addr) external onlyOwner
xiii. function setMinTxAmount(uint256 amount) external onlyOwner
xiv. function setStakingPoolAddress(address addr) external onlyOwner
xv. function setFeesEnabled(bool value) external onlyOwner
xvi. function setPauseEnabled(bool value) external onlyOwner
xvii. function setStakingPoolCallbackEnabled(bool value) external onlyOwner
xviii. function transfer(address to, uint256 amount)public virtual override returns (bool)
xix. function transferFrom(address from,address to,uint256 amount) public virtual override returns (bool)
xx. function transferOwnership(address newOwner) public virtual onlyOwner
xxi. function renounceOwnership() public virtual 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

Executable Functions

WESENDIT Contract

function will transfer token for a specified address. “to” is the address to transfer’ to. “amount” is the amount to be transferred. Owner's account must have sufficient balance to transfer.

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

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

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

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) {
    address owner = _msgSender();
    uint256 currentAllowance = allowance(owner, spender);
    require(currentAllowance >= subtractedValue,
    "ERC20: decreased allowance below zero");
    unchecked {
        _approve(owner, spender, currentAllowance - subtractedValue);
    }
    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 from,
    address to,
    uint256 amount
) public virtual override returns (bool) {
    address spender = _msgSender();
    _spendAllowance(from, spender, amount);
    _transfer(from, to, 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) .

```
function increaseAllowance(address spender, uint256 addedValue) public
virtual returns (bool) {
    address owner = _msgSender();
    _approve(owner, spender, allowance(owner, spender) + addedValue);
    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) {
    address owner = _msgSender();
    _approve(owner, spender, amount);
    return true;
}
```

Owner of the tokens can burn own tokens.

```
function burn(uint256 amount) public virtual {
    _burn(_msgSender(), amount);
}
```

Destroys `amount` tokens from `account`, deducting from the caller's allowance.
Requirements:
The caller must have allowance for account's tokens of at least `amount`.

```
function burnFrom(address account, uint256 amount) public virtual {
    _spendAllowance(account, _msgSender(), amount);
    _burn(account, amount);
}
```

Owner of this contract renounces the ownership. New owner will be the Zero address.

```
function renounceOwnership() public virtual onlyOwner {
    _transferOwnership(address(0));
}
```

Owner of this contract updates the pancake swap router.

```
function setPancakeRouter(address addr) external onlyOwner {
    pancakeRouterAddress = addr;
}
```

Owner of this contract updates the activity pool address.

```
function setActivityPoolAddress(address addr) external onlyOwner {
    activityPoolAddress = addr;
}
```

Owner of this contract updates the referral pools address.

```
function setReferralPoolAddress(address addr) external onlyOwner {
    referralPoolAddress = addr;
}
```

Owner of this contract set the staking pool address.

```
function setStakingPoolAddress(address addr) external onlyOwner {
    stakingPoolAddress = addr;
}
```

Owner of this contract updates the minimum transaction amount.

```
function setMinTxAmount(uint256 amount) external onlyOwner {
    minTxAmount = amount;
}
```

Owner of this contract flips the status of fees deduction.

```
function setFeesEnabled(bool value) external onlyOwner {
    feesEnabled = value;
}
```

Owner of this contract updates the pause status.

```
function setPauseEnabled(bool value) external onlyOwner {
    pauseEnabled = value;
}
```

Owner of this contract updates the staking pool call back status.

```
function setStakingPoolCallbackEnabled(bool value) external onlyOwner {
    stakingPoolCallbackEnabled = value;
}
```

Owner of this contract transfer the contracts WSI tokens as 5 % to seed sale wallet address and 8% for the private sale wallet.

```
function distributeSaleToken(
    address seedSaleWallet,
    address privateSaleWallet
) external onlyOwner {
    _transfer(address(this), seedSaleWallet, 75000000 * 1 ether); // 5%
    _transfer(address(this), privateSaleWallet, 120000000 * 1 ether); // 8%
}
```

Owner of this contract transfer the contracts WSI token to address as listed below.

Team wallet	12%
Advisor wallet	5 %
Referral wallet	5 %
Development wallet	15 %
Marketing wallet	12 %
Operations wallet	10 %
Exchange and liquidity wallet	8 %
Staking rewards wallet	8 %
Activity rewards wallet	3 %
Airdrop wallet	3 %

```
function distributeToken(address teamWallet,address advisorsWallet,address referralsWallet,
    address developmentWallet,address marketingWallet,address operationsWallet,
    address exchangeAndLiquidityWallet,address stakingRewardsWallet,
    address activityRewardsWallet,
    address airdropWallet
) external onlyOwner {
    _transfer(address(this), teamWallet, 180000000 * 1 ether); // 12%
    _transfer(address(this), advisorsWallet, 75000000 * 1 ether); // 5%
    _transfer(address(this), referralsWallet, 75000000 * 1 ether); // 5%
    _transfer(address(this), developmentWallet, 225000000 * 1 ether); // 15%
    _transfer(address(this), marketingWallet, 180000000 * 1 ether); // 12%
    _transfer(address(this), operationsWallet, 150000000 * 1 ether); // 10%
    _transfer(
        address(this),
        exchangeAndLiquidityWallet,
        120000000 * 1 ether
    ); // 8%
    _transfer(address(this), stakingRewardsWallet, 120000000 * 1 ether); // 8%
    _transfer(address(this), activityRewardsWallet, 45000000 * 1 ether); // 3%
    _transfer(address(this), airdropWallet, 45000000 * 1 ether); // 3%
}
```

Owner of this contract the contracts WSI token as 3.5 % to general reserve wallet.

```
function distributeReserveToken(
    address generalReserveWallet
) external onlyOwner {
    _transfer(address(this), generalReserveWallet, 52500000 * 1 ether); // 3.5%
}
```

Owner of this contract withdraws the contracts WSI token balance to the “receiver address”.

```
function emergencyWithdraw(address receiver) external onlyOwner {
    _transfer(address(this), receiver, balanceOf(address(this)));
}
```

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.

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 WESENDIT Smart Contract protocol consists of one smart contract. It has other inherited contracts like ERC20, ERC20Capped, ERC20Burnable, Ownable. These are compact and well written contracts. Libraries used in WESENDIT 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 WESENDIT 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.