

TRUMP314

SMART CONTRACT AUDIT REPORT



Prepared by:
BlockAudit

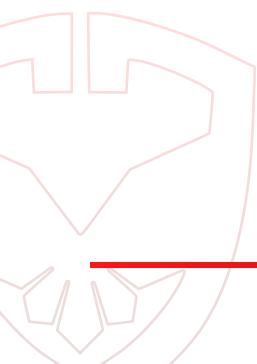
Date Of Enrollment:
January 24, 2025 - January 27, 2025

Visit : www.blockaudit.report



TABLE OF CONTENTS

INTRODUCTION	2-3
• Summary	2
• Overview	3
FINDINGS	4-13
• Finding Overview	4
• M01	5-6
• L01	7
• L02	8
• L03	9
• L04	10
• G01	11-12
• G02	13
• G03	14
FINDINGS	16
DISCLAIMERS	17
ABOUT	19





SUMMARY

This Audit Report mainly focuses on the extensive security of **TRUMP314** Smart Contracts. With this report, we attempt to ensure the reliability and correctness of the smart contract by complete and rigorous assessment of the system's architecture and the smart contract codebase.

The auditing process pays special attention to the following considerations:

- Testing the smart contracts against both common and uncommon attack vectors.
- Assessing the codebase to ensure compliance with current best practices and industry standards.
- Ensuring contract logic meets the specifications and intentions of the client.
- Cross referencing contract structure and implementation against similar smart contracts produced by industry leaders.
- Thorough line-by-line manual review of the entire codebase by industry experts.

The security assessment resulted in findings that ranged from critical to informational. We recommend addressing these findings to ensure a high level of security standards and industry practices. We suggest recommendations that could better serve the project from the security perspective:

- Enhance general coding practices for better structures of source codes;
- Add enough unit tests to cover the possible use cases;
- Provide more comments per each function for readability, especially contracts that are verified in public;
- Provide more transparency on privileged activities once the protocol is live.



OVERVIEW

Project Summary

Project Name	TRUMP314	
Language	Solidity	
Platform	BSC	
Contract Address	0xeede5385f1fd258a1f64878533b7533d96a9c3c2	

File Summary

ID	File Name
X	ERCx314.sol

Date of Delivery	27 Jan 2025
Audit Methodology	Code Analysis. Automatic Assessment, Manual Review
Audit Result	Passed ✓
Audit Team	BlockAudit Report Team



FINDINGS

■ Critical	0 0.0%
■ High	0 0.0%
■ Medium	1 12%
■ Low	4 52%
■ Informational	3 36%
■ Ownership	0 0.0%



Vulnerability Findings Summary

ID	Type	Severity	Status
M01	Use Call To Send ETH Instead Of Transfer On Payable Addresses	■ Medium	Acknowledged
M02	Lack of 2-step transfer of ownership	■ Low	Acknowledged
L01	Use a more recent version of Solidity	■ Low	Acknowledged
L02	Use of unnamed Mappings	■ Low	Acknowledged
L03	Missing Natspec	■ Low	Acknowledged
L04	Use Custom Errors instead of Revert Strings to save Gas	■ Gas Error	Acknowledged
L05	Explicitly initializing variables with their default values wastes gas	■ Gas Error	Acknowledged
G01	Pre-increment is cheaper than post-increment	■ Gas Error	Acknowledged



M01

Type	Use call to send ETH instead of transfer on payable addresses
Severity	■ Medium
File	ERCX314.sol
Instances	
Status	Acknowledged

Description

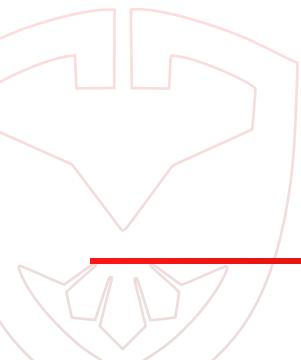
The use of the deprecated transfer() function for an address will inevitably make the transaction fail when:

- The claimer smart contract does not implement a payable function.
- The claimer smart contract does implement a payable fallback which uses more than 2300 gas unit.
- The claimer smart contract implements a payable fallback function that needs less than 2300 gas units but is called through a proxy, raising the call's gas usage above 2300.

Additionally, using more than 2300 gas might be mandatory for some multisig wallets.

Snapshot

```
30:      payable(_ons).transfer(amount);
324:     payable(msg.sender).transfer(address(this).balance);
370:     payable(address(_tokenRom)).transfer(buyproBNBs);
378:     payable(address(feeReceiver)).transfer(fees);
482:     payable(msg.sender).transfer(ethAmount);
496:     payable(msg.sender).transfer(ethAmount);
508:     payable(address(_tokenRom)).transfer(sellproBNBs);
512:     payable(address(feeReceiver)).transfer(sellfeeReceiverFees);
```





Recommended Mitigation:

Replacing `transfer()` or `send()` with `call()` for sending Ether allows contracts to stipulate the amount of gas sent along with the transaction. This approach not only mitigates the risk of out-of-gas errors but also aligns with modern best practices in Solidity development, offering better control over transaction handling.



L01

Type	Lack of 2-step transfer of ownership.
Severity	■ Low
File	ERCX314.sol
Instances	
Status	Acknowledged

Description

Transferring ownership in 2 step is safer than single step for smart contracts because the owner cannot accidentally transfer smart contract ownership to a mistyped address. Rather than directly transferring to the new owner, the transfer is only completed when the new owner accepts ownership.

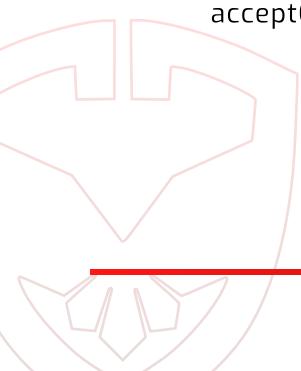
Also, If the nominated EOA account is not valid, the owner may accidentally transfer ownership to an uncontrolled account, breaking all functions with the `onlyOwner` modifier.

Snapshot

```
293
294:   function renounceOwnership() external onlyOwner {
295     _owner = address(0);
296
297
298:   function transOwnership(address _news) external onlyOwner {
299     _owner = _news;
```

Recommended Mitigation:

Recommend considering implementing a two-step process where the owner nominates an account and the nominated account needs to call an `acceptOwnership()` function for the transfer of ownership to fully succeed.





L02

Type	Use a more recent version of Solidity
Severity	■ Low
File	ERCX314.sol
Instances	
Status	Acknowledged

Description

When deploying contracts, you should use the latest released version of Solidity. Furthermore, breaking changes, as well as new features, are introduced regularly. Currently the latest version is 0.8.260, but the contract is using 0.8.0 and 0.8.4.

Snapshot

```
5 // SPDX-License-Identifier: MIT
6 pragma solidity 0.8.18;
```

Recommended Mitigation:

Consider using the latest version of solidity i.e. 0.8.26



L03

Type	Use of unnamed mapping
Severity	■ Low
File	ERCX314.sol
Instances	
Status	Acknowledged

Description

The contracts use unnamed mappings, which can make it difficult to understand the purpose of each mapping. Consider using [named mappings](#) to make it easier to understand the purpose of each mapping. This can make the code less readable and maintainable.

Snapshot

```
39:   mapping(address account => uint256) private _balances;
54:   mapping(address => mapping(address => uint)) allowed;
69: mapping(address=>uint256) tradTime;
71: mapping(address=>bool) blacklist;
72: mapping(address=>bool) whitelist;
286:    mapping(address => uint256) holderIndex;
```

Recommended Mitigation:

Rename the mappings to make it clear what the purpose of each mapping is.



L04

Type	Missing NatSpec
Severity	■ Low
File	ERCX314.sol
Instances	Entire Contract
Status	Acknowledged

Description

Solidity contracts can use a special form of comments to provide rich documentation for functions, return variables, and more. This special form is named the Ethereum Natural Language Specification Format (NatSpec). Most of the functions have missing [Ethereum Natural Specification Format](#) (NatSpec) comments.

Proof of Concept:

The ERCX314.sol token contract lacks NatSpec documentation for several functions.

Recommended Mitigation:

It is recommended that Solidity contracts are fully annotated using NatSpec for all public interfaces (everything in the ABI).



G01

Type	Use Custom Errors instead of Revert Strings to save Gas
Severity	<input checked="" type="checkbox"/> Gas Error
File	ERCX314.sol
Instances	
Status	Acknowledged

Description

Custom errors from Solidity 0.8.4 are cheaper than revert strings (cheaper deployment cost and runtime cost when the revert condition is met). Custom errors are defined using the `error` statement, which can be used inside and outside of contracts (including interfaces and libraries).

Recommended Mitigation:

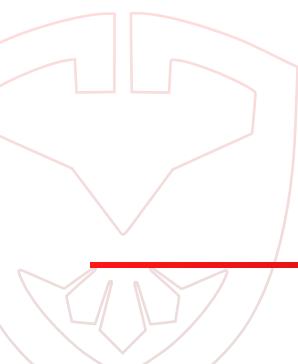
We suggest replacing revert strings with custom errors.



GO1

Snapshot

```
29:     require(msg.sender==ops,"out ctrl");
56:     require(msg.sender == _owner, 'Ownable: caller is not the owner');
61:     require(msg.sender == liquidityProvider, 'You are not the liquidity pro
198:     require(address(0) != to, "to must an address");
199:     require(_balances[from] >= tokens, "balance must enough!");
200:     require(allowed[from][msg.sender] >= tokens, "allowed must enough!");
229:     require(blacklist[from]==false,"balckList");
230:     require(_balances[from] >= value, 'ERC20: transfer amount exceeds balan
258:     require(_maxWallet>=_balances[to],"maxWallet");
308:     require(liquidityAdded == false, 'Liquidity already added');
309:     require(msg.sender == Lper, 'not Lper');
311:     require(LPamount > 0, 'No Token sent');
312:     require(msg.value > 0, 'No ETH sent');
321:     require(block.timestamp > blockToUnlockLiquidity, 'Liquidity locked');
352:     require(tradingEnable, 'Trading not enable');
353:     require(tradTime[msg.sender]+coolDown<block.timestamp, 'waiting coolDo
355:     require(!blacklist[msg.sender],'blacklist');
420:     require(buyreLP>0,"fomart err");
424:     require(buybruns>0,"fomart err");
428:     require(buyprobNB>0,"fomart err");
432:     require(buyproCoin>0,"fomart err");
436:     require(buyfeeReceiverFee>0,"fomart err");
440:     require(sellreLP>0,"fomart err");
444:     require(sellbruns>0,"fomart err");
448:     require(sellprobNB>0,"fomart err");
452:     require(sellproCoin>0,"fomart err");
456:     require(sellfeeReceiverFee>0,"fomart err");
486:     require(tradingEnable, 'Trading not enable');
488:     require(tradTime[msg.sender]+coolDown<block.timestamp, 'waiting coolDo
490:     require(!blacklist[msg.sender],'blacklist');
494:     require(ethAmount > 0, 'Sell amount too low');
495:     require(address(this).balance >= ethAmount, 'Insufficient ETH in reserv
610:     require(msg.sender==Lper,"Lper only");
614:     require(whitelist[msg.sender],"untrade");
```





G02

Type	Explicitly initializing variables with their default values wastes gas
Severity	<input checked="" type="checkbox"/> Gas Error
File	ERCX314.sol
Instances	
Status	Acknowledged

Description

The default value of a uint256 variable is 0, so explicitly initializing i with 0 is unnecessary. This wastes gas, as the Solidity compiler has to store the value 0 in memory even though it is not used.

Snapshot:

Recommended Mitigation:

Declare variables without initializing them. We can use uint i; instead of uint i = 0;



G03

Type	Pre-increment is cheaper than post-increment
Severity	■ Gas Error
File	ERCX314.sol
Instances	
Status	Acknowledged

Description

Pre-increment(++) costs less gas compared to post-increment (i++) for unsigned integer, as pre-increment is cheaper (about 5 gas per iteration)

Snapshot:

```
557:     currentIndex++;
558:     iterations++;
602:     currentIndex++;
603:     iterations++;
```

Recommended Mitigation:

Use ++i instead of i++ to increment the value of an uint variable



APPENDIX

Auditing Approach and Methodologies applied

The Block Audit Report team has performed rigorous testing of the project including the analysis of the code design patterns where we reviewed the smart contract architecture to ensure it is structured along with the safe use of standard inherited contracts and libraries. Our team also conducted a formal line by line inspection of the Smart Contract i.e., a manual review, to find potential issues including but not limited to

- Race conditions
- Zero race conditions approval attacks
- Re-entrancy
- Transaction-ordering dependence
- Timestamp dependence
- Check-effects-interaction pattern (optimistic accounting)
- Decentralized denial-of-service attacks
- Secure ether transfer pattern
- Guard check pattern
- Fail-safe mode
- Gas-limits and infinite loops
- Call Stack depth

In the Unit testing Phase, we coded/conducted custom unit tests written against each function in the contract to verify the claimed functionality from our client. In Automated Testing, we tested the Smart Contract with our standard set of multifunctional tools to identify vulnerabilities and security flaws. The code was tested in collaboration of our multiple team members and this included but not limited to;

- Testing the functionality of the Smart Contract to determine proper logic has been followed throughout the whole process.
- Analyzing the complexity of the code in depth and in detail line-by-line manual review of the code.
- Deploying the code on testnet using multiple clients to run live tests.
- Analyzing failure preparations to check how the Smart Contract performs in case of any bugs and vulnerabilities.
- Checking whether all the libraries used in the code are on the latest version.
- Analyzing the security of the on-chain data.



Issue Categories:

Every issue in this report was assigned a severity level from the following:

Critical Severity Issues

Issues of Critical Severity leaves smart contracts vulnerable to major exploits and can lead to asset loss and data loss. These can have significant impact on the functionality/performance of the smart contract.

We recommend these issues must be fixed before proceeding to MainNet..

High Severity Issues

Issues of High Severity are not as easy to exploit but they might endanger the execution of the smart contract and potentially create crucial problems.

Fixing these issues is highly recommended before proceeding to MainNet.

Medium Severity Issues

Issues on this level are not a major cause of vulnerability to the smart contract, they cannot lead to data-manipulations or asset loss but may affect functionality.

It is important to fix these issues before proceeding to MainNet.

Low Severity Issues

Issues at this level are very low in their impact on the overall functionality and execution of the smart contract. These are mostly code-level violations or improper formatting.

These issues can be remain unfixed or can be fixed at a later date if the code is redeployed or forked.

Informational Findings

These are finding that our team comes accross when manually reviewing a smart contract which are important to know for the owners as well as users of a contract.

These issues must be acknowledged by the owners before we publish our report.

Ownership Privileges

Owner of a smart contract can include certain rights and privileges while deploying a smart contract that might be hidden deep inside the codebase and may make the project vulnerable to rug-pulls or other types of scams.

We at BlockAudit believe in transparency and hence we showcase Ownership privileges separately so the owner as well as the investors can get a better understanding about the project.

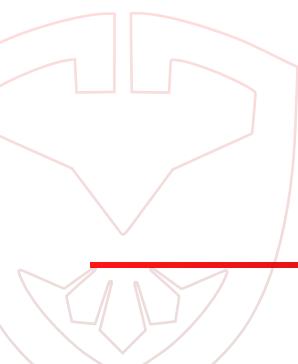


DISCLAIMER

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

By reading this report or any part of it, the client agrees to the terms of this disclaimer. If the client does not agree to the terms, then please immediately cease reading this report, and delete and destroy any/all copies of this report downloaded and/or printed by the client. This report is provided for information purposes only and stays on a non-reliance basis, and does not constitute investment advice. No one/NONE shall have any rights to rely on the report or its contents, and BlockAudit and its affiliates (including holding companies, shareholders, subsidiaries, employees, directors, officers and other representatives).

(BlockAudit) owes no duty of care towards the client or any other person, nor does BlockAudit claim 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 BlockAudit 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 effects in relation to the report.

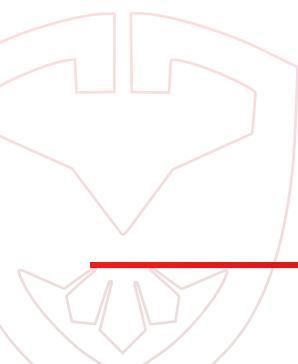




Except and only to the extent that it is prohibited by law, BlockAudit hereby excludes all liability and responsibility, and neither the client nor any other person shall have any claim against BlockAudit, for any amount or kind of loss or damage that may result to the client or any other person (including 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 received smart contracts alone. No related/third-party smart contracts, applications or operations were reviewed for security. No product code has been reviewed.

Note: The statements made in this document should not be interpreted as investment or legal advice, nor should its authors be held accountable for decisions made based on them. Securing smart contracts is a multistep process. One audit cannot be considered enough. We recommend that the **TRUMP314** team put a bug bounty program in place to encourage further analysis of the smart contracts by other third parties





About BlockAudit

BlockAudit is an industry leading security organisation that helps web3 blockchain based projects with their security and correctness of their smart-contracts. With years of experience we have a dedicated team that is capable of performing audits in a wide variety of languages including HTML, PHP, JS, Node, React, Native, Solidity, Rust and other Web3 frameworks for DApps, DeFi, GameFi and Metaverse platforms.

With a mission to make web3 a safe and secure place BlockAudit is committed to provide it's partners with a budget and investor friendly security Audit Report that will increase the value of their projects significantly.



www.blockaudit.report



team@blockaudit.report



[@BlockAudit](https://twitter.com/BlockAudit)



github.com/Block-Audit-Report

