

AuditBlock



TokenConverter

v0.8.18+commit.87f61d96

✦ Low-Risk

low-risk code

✦ Medium-Risk

medium-risk code

✦ High-Risk

high-risk code

TokenConverter

Disclaimer AUDITBLOCK is not responsible for any financial losses. Nothing in this contract audit is financial advice, please do your own research.

Executive Summary

Contract Name TokenConverter

Overview This contract allows for users to stake in order to yield a substantial reward in the future. The contract inherits 4 basic contracts from the standard Openzeppelin libraries; Ownable and Pancakeswap protocol libraries; IPancakeRouter02, IERC20, IPancakeFactory.sol. These libraries aid in the moderation of access control for some functions, the stake of some activities in contract, and also provide guard against reentrancy attacks.

Method Manual Review, Functional Testing, Automated Testing etc.

Scope of Audit The scope of this audit was to analyze the contract codebase for quality, security, and correctness.
<https://bscscan.com/address/0x49aFBD0A146Afd6D9d5d504fe231a3BA344CEc80#code>



High

Medium

Low

Informational

High

Medium

Low

Informational

Open Issues

0

0

0

0

Acknowledged Issues

1

3

0

0

Partially Resolved Issues

0

0

0

0

Resolved Issues



0

0

0

0

Checked Vulnerabilities

- | | |
|---|--|
|  Re-entrancy |  Tautology or contradiction |
|  Timestamp Dependence |  Missing Zero Address Validation |
|  Gas Limit and Loops |  Return values of low-level calls |
|  Exception Disorder |  Revert/require functions |
|  Gasless Send |  Private modifier |
|  Use of tx.origin |  Using block.timestamp |
|  Compiler version not fixed |  Multiple Sends |
|  Address hardcoded |  Using SHA3 |
|  Divide before multiply |  Using suicide |
|  Integer overflow/underflow |  Using throw |
|  Dangerous strict equalities |  Using inline assembly |

Types of Severities

High

A high-severity issue or vulnerability means that your smart contract can be exploited. Issues on this level are critical to the smart contract's performance or functionality, and we recommend these issues be fixed before moving to a live environment.

Medium

The issues marked as medium severity usually arise because of errors and deficiencies in the smart contract code. Issues on this level could potentially bring problems, and they should still be fixed.

Low

Low-level severity issues can cause minor impact and or are just warnings that can remain unfixed for now. It would be better to fix these issues at some point in the future.

Informational

These are severity issues that indicate an improvement request, a general question, a cosmetic or documentation error, or a request for information. There is low-to-no impact.

Techniques and Methods

The overall quality of code.

- Use of best practices.
- Code documentation and comments match logic and expected behavior.
- Token distribution and calculations are as per the intended behavior mentioned in the whitepaper.
- implementation of ERC-20 token standards.
- Efficient use of gas.
- Code is safe from re-entrance and other vulnerabilities.

The following techniques, methods, and tools were used to review all the smart contracts.

Structural Analysis

In this step, we have analyzed the design patterns and structure of smart contracts. A thorough check was done to ensure the smart contract is structured in a way that will not result in future problems.

Static Analysis

Static analysis of smart contracts was done to identify contract vulnerabilities. In this step, a series of automated tools are used to test the security of smart contracts.

Code Review / Manual Analysis

Manual analysis or review of code was done to identify new vulnerabilities or verify the vulnerabilities found during the static analysis. Contracts were completely manually analyzed, and their logic was checked and compared with the one described in the whitepaper. Besides, the results of the automated analysis were manually verified.

Gas Consumption

In this step, we have checked the behavior of smart contracts in production. Checks were done to know how much gas gets consumed and the possibilities of optimization of code to reduce gas consumption.

Tools and Platforms Used for Audit

Remix IDE, Truffle, Truffle Team, Solhint, Mythril, Slither, Solidity statistic analysis.

Manual Testing

A. Contract - TokenConverter

High Severity Issues

1. The unchecked transfer issue

Description

In contract. The unchecked transfer issue in Solidity is when the return value of an external transfer or transferFrom call is not checked. This means that the contract does not verify whether the call was successful or not. If the call fails, the contract will continue executing, even though the transfer may not have been completed. This can be a security vulnerability, because it allows an attacker to exploit the contract by causing the transfer to fail. For example, the attacker could send a transaction with an invalid value, or they could send the transaction to an address that does not exist. If the contract does not check the return value, it will not be aware that the transfer failed, and it will continue executing as if the transfer was successful.

Remediation

Implement To prevent this issue, it is important to always check the return value of any external transfer or transferFrom call. This can be done using the **required** statement.

Status

open

Medium Severity Issues

1. Divide-before-multiply

Description

In contract. integer division truncates, which means that the decimal part of the result is discarded. This can lead to precision loss if the result is then multiplied by another integer.

Remediation

Implement To prevent this issue Consider ordering multiplication before division.

Status

Acknowledged

2. Reentrancy-vulnerabilities

Description

In contract. A reentrancy vulnerability is a type of security flaw that can occur in Solidity smart contracts. It happens when a contract calls an external contract and then continues executing code before the external contract has finished executing. This can allow an attacker to call the contract recursively and potentially manipulate its state.

Remediation

Implement To prevent this issue. This is a serious security risk in Solidity smart contracts. Developers should take steps to prevent these vulnerabilities by using the `required` and `revert` statements.

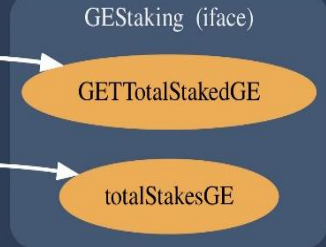
Status

Acknowledged

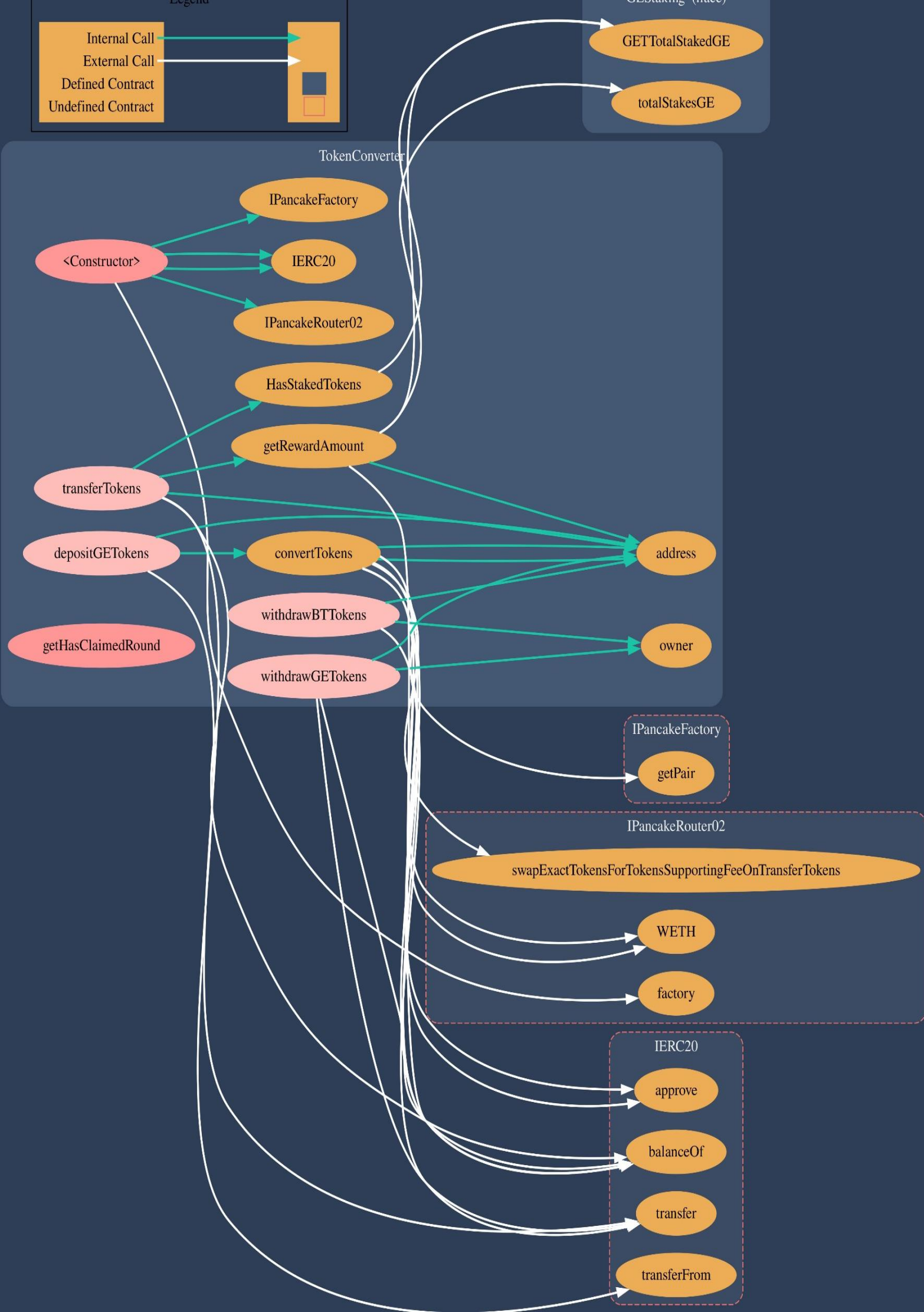
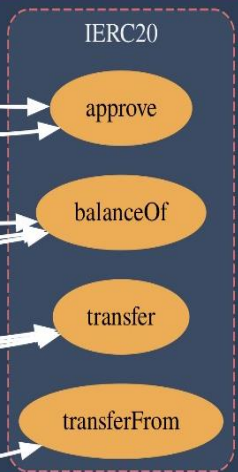
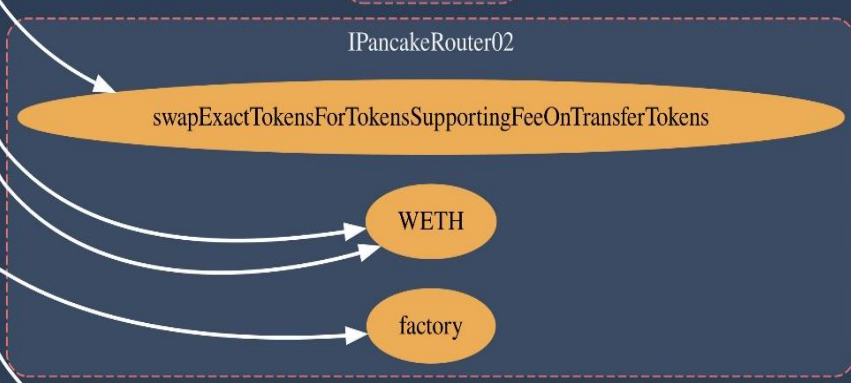
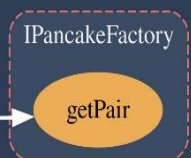
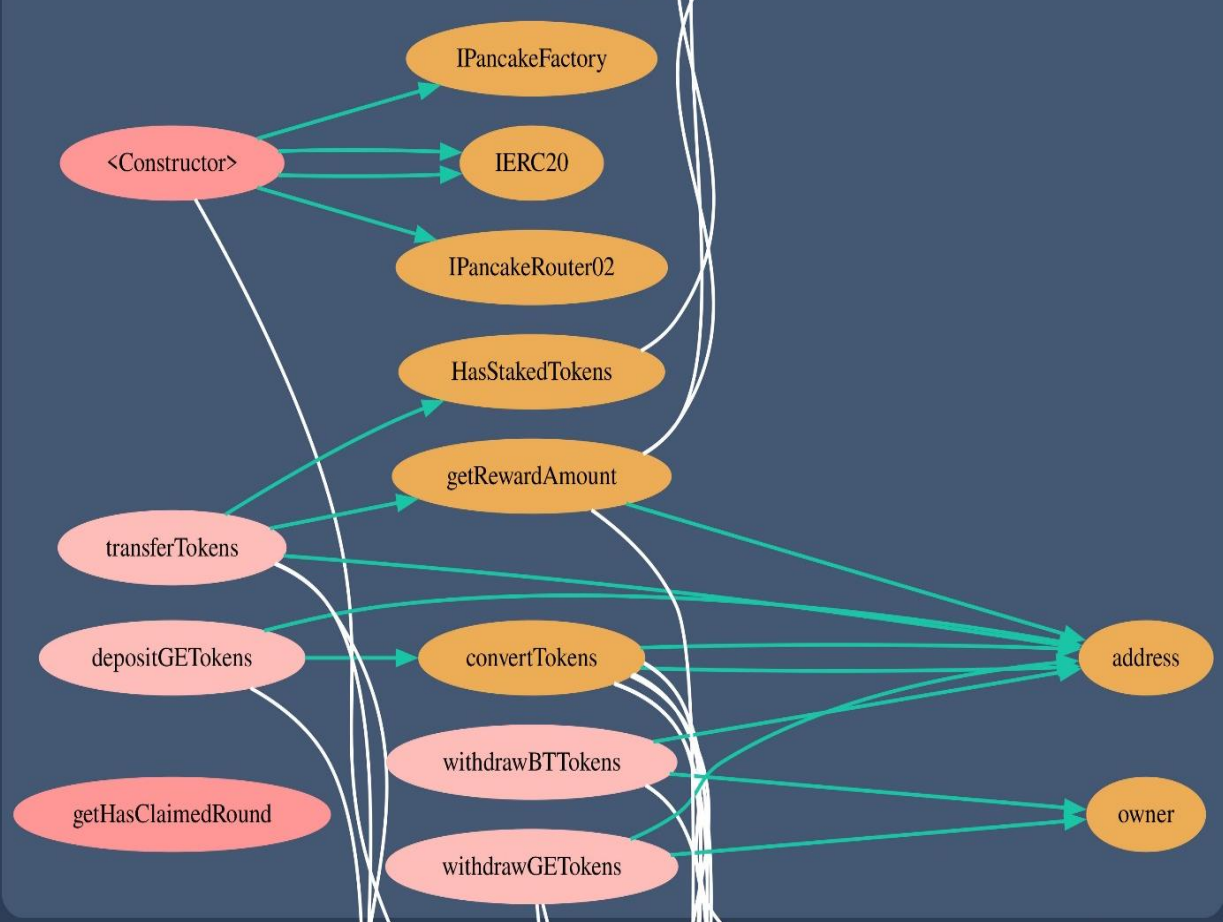
Low Severity Issues

No issues found

Legend



TokenConverter



Automated Test

TokenConverter.depositGETokens(uint256) (contracts/contract.sol#32-36) ignores return value by GEToken.transferFrom(msg.sender,address(this),amount * 10 ** 18) (contracts/contract.sol#34)

TokenConverter.transferTokens() (contracts/contract.sol#60-68) ignores return value by BitTorrentToken.transfer(msg.sender,amount) (contracts/contract.sol#66)

TokenConverter.withdrawBTokens() (contracts/contract.sol#70-74) ignores return value by BitTorrentToken.transfer(owner(),contractTokenBalance) (contracts/contract.sol#73)

TokenConverter.withdrawGETokens() (contracts/contract.sol#76-80) ignores return value by GEToken.transfer(owner(),contractTokenBalance) (contracts/contract.sol#79)

Reference: <https://github.com/crytic/slither/wiki/Detector-Documentation#unchecked-transfer>

TokenConverter.getRewardAmount(address) (contracts/contract.sol#85-93) performs a multiplication on the result of a division:

- percentageReward = (tokensStakedByUser * 10 ** 12) /

totalStakedTokens (contracts/contract.sol#88)

- amount = (percentageReward * totalReward) / 10 ** 12

(contracts/contract.sol#91)

Reference: <https://github.com/crytic/slither/wiki/Detector-Documentation#divide-before-multiply>

Reentrancy in TokenConverter.transferTokens() (contracts/contract.sol#60-68) External calls:

- BitTorrentToken.transfer(msg.sender,amount)

(contracts/contract.sol#66)

State variables written after the call(s):

- hasClaimedRound[currentRound][msg.sender] = true

(contracts/contract.sol#67)

TokenConverter.hasClaimedRound (contracts/contract.sol#13) can be used in cross function reentrancies:

- TokenConverter.getHasClaimedRound(uint256)

(contracts/contract.sol#82-84)

- TokenConverter.hasClaimedRound (contracts/contract.sol#13)

- TokenConverter.transferTokens() (contracts/contract.sol#60-68)

Reference: <https://github.com/crytic/slither/wiki/Detector-Documentation#reentrancy-vulnerabilities-1>

Functional Testing

Some of the tests performed are mentioned below:

- ✗ Should revert when non-owner calls the start function
- ✓ Should be able to start contract activities after the contract owner successfully calls the start function
- ✓ Should revert when users call reward when the contract has not started
- ✓ Should revert when users attempt to deposit less than the required stake amount Should revert
- ✓ when users set themselves as stakeholders
- ✓ Should be able to deposit successfully and continuously.
- ✓ Should revert if the user attempts to reinvest with zero
- ✓ Should revert if the user attempts to reinvest when they have not yielded enough Should
- ✓ be able to reinvest when yielded funds in the contract are enough
- ✗ Should revert when users call to withdrawBTTToken with the condition required
- ✗ Should be able to withdrawGETTokens with conditional requirement
- ✓ Should be able to remove users by contract owner when leave function is called Owner should be able to
- ✓ withdraw funds from the contracts without any hindrance.

SOLIDITY UNIT TESTING

Progress: Starting

PASS ✓ Tested

- ✓ Check winning proposal
- ✓ Check the winning proposal with value
- ✓ Before all
- ✓ Check success
- ✓ Check success2
- ✓ Check sender and value

Result for tests Passed:

Time Taken: 0.34s

Closing Summary

In this report, we have considered the security of the Token TokenConverter. We performed our audit according to the procedure described above.

Some issues of Medium, Low, and Informational severity were found, Some suggestions and best practices are also provided in order to improve the code quality and security posture. In the end, TokenConverter. Team Acknowledged all Issues.

Disclaimer

AuditBlock smart contract audit is not a security warranty, investment advice, or an endorsement of this Platform. This audit does not provide a security or correctness guarantee for the audited smart contracts.

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 multi-step process. One audit cannot be considered enough. We recommend that the contract Team put in place a bug bounty program to encourage further analysis of the smart contract by other third parties.

About AudiTBlock

Contractsaudit is a secure smart contracts audit platform designed by Auditblock

We are a team of dedicated blockchain security experts and smart contract auditors determined to ensure that Smart Contract-based Web3 projects can avail the latest and best security solutions to operate in a trustworthy and risk-free ecosystem.



300+

Audits Completed



\$3B

Secured



300K

Lines of Code Audited

Audit Report,
2 September 2023



For
0x49aFB.....4CEc80



auditblock@gmail.com



AudiTBlock