QuillAudits

AUDIT REPORT

August 2025

For

Watt Trade

Table of Content

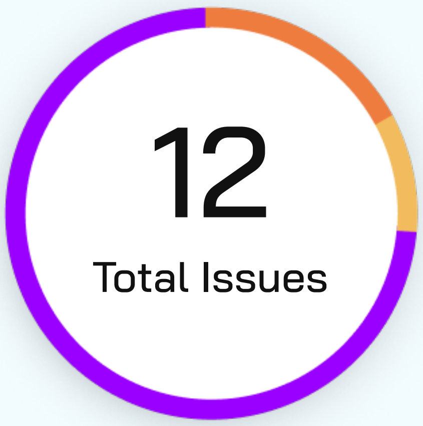
Executive Summary

Project Name	Wattoin
Protocol Type	ERC20 contract.
Project URL	https://www.watt2trade.com/
Overview	Wattoin is ERC20 smart contract that deducts fees on transfer. It uses 0 decimals to make one whole token non divisible by not allowing granularity in amounts.
Audit Scope	The scope of this Audit was to analyze the Wattoin Smart Contracts for quality, security, and correctness.
Source Code link	https://github.com/Jelly-io/dex-wattoin/blob/main/contracts/Wattoin.sol
Branch	main
Contracts in Scope	Wattoin.sol
Commit Hash	112ea0c59e077c160e1e19c6656fa73503db6d3b
Language	Solidity
Method	Manual Analysis, Functional Testing, Automated Testing
Review 1	12th August 2025 - 20th August 2025
Updated Code Received	3rd September 2025
Review 2	8th September 2025
Fixed In	fa5d0168f1838374264244e821ce848a6e1f69bd

Verify the Authenticity of Report on QuillAudits Leaderboard:

https://www.quillaudits.com/leaderboard

Number of Issues per Severity



🟥	Critical	0 (0%)
🟥	High	0 (0%)
🟧	Medium	2 (16.7%)
🟨	Low	1 (8.3%)
🟪	Informational	9 (75.0%)

Severity

Issues	Critical	High	Medium	Low	Informational
Open	0	0	0	0	0
Acknowledged	0	0	**2**	0	**4**
Partially Resolved	0	0	0	0	0
Resolved	0	0	0	**1**	**5**

Summary of Issues

Issue No.	Issue Title	Severity	Status
1	Precision loss in fee calculation, which also allows bypassing fees for small calculations.	Medium	Acknowledged
2	The max amount per wallet restriction can be tricked	Medium	Acknowledged
3	Floating pragma	Low	Resolved
4	Add events according to the requirement	Informational	Acknowledged
5	Redundant 0 value assignment	Informational	Resolved
6	Redundant inheritance and import	Informational	Resolved
7	Add checks for max fee	Informational	Acknowledged
8	Note about fee deduction	Informational	Acknowledged
9	Redundant function overriding	Informational	Resolved
10	Note the intended functionality	Informational	Resolved
11	Note about 0 decimals	Informational	Acknowledged
12	Correct the contract name in the error message	Informational	Resolved

Checked Vulnerabilities

- ✓ Access Management
- ✓ Arbitrary write to storage
- ✓ Centralization of control
- ✓ Ether theft
- ✓ Improper or missing events
- ✓ Logical issues and flaws
- ✓ Arithmetic Computations Correctness
- ✓ Race conditions/front running
- ✓ SWC Registry
- ✓ Re-entrancy
- ✓ Timestamp Dependence
- ✓ Gas Limit and Loops
- ✓ Exception Disorder
- ✓ Gasless Send
- ✓ Use of tx.origin
- ✓ Malicious libraries
- ✓ Compiler version not fixed
- ✓ Address hardcoded
- ✓ Divide before multiply
- ✓ Integer overflow/underflow
- ✓ ERC's conformance
- ✓ Dangerous strict equalities
- ✓ Tautology or contradiction
- ✓ Return values of low-level calls

- ✓ **Missing Zero Address Validation**
- ✓ **Private modifier**
- ✓ **Revert/require functions**
- ✓ **Multiple Sends**
- ✓ **Using suicide**
- ✓ **Using delegatecall**
- ✓ **Upgradeable safety**
- ✓ **Using throw**
- ✓ **Using inline assembly**
- ✓ **Style guide violation**
- ✓ **Unsafe type inference**
- ✓ **Implicit visibility level**

Techniques and Methods

Throughout the audit of smart contracts, care was taken to ensure:

- 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 standards
- Efficient use of gas
- Code is safe from re-entrancy 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

A 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, 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, Foundry, Solhint, Mythril, Slither, Solidity Static Analysis.

Types of Severity

Every issue in this report has been assigned to a severity level. There are five levels of severity, and each of them has been explained below.

Critical: Immediate and Catastrophic Impact

Critical issues are the ones that an attacker could exploit with relative ease, potentially leading to an immediate and complete loss of user funds, a total takeover of the protocol's functionality, or other catastrophic failures. Critical vulnerabilities are non-negotiable; they absolutely must be fixed.

High (H): Significant Risk of Major Loss or Compromise

High-severity issues represent serious weaknesses that could result in significant financial losses for users, major malfunctions within the protocol, or substantial compromise of its intended operations. While exploiting these vulnerabilities might require specific conditions to be met or a moderate level of technical skill, the potential damage is considerable. These findings are critical and should be addressed and resolved thoroughly before the contract is put into the Mainnet.

Medium (M): Potential for Moderate Harm Under Specific Circumstances

Medium-severity bugs are loopholes in the protocol that could lead to moderate financial losses or partial disruptions of the protocol's intended behavior. However, exploiting these vulnerabilities typically requires more specific and less common conditions to occur, and the overall impact is generally lower compared to high or critical issues. While not as immediately threatening, it's still highly recommended to address these findings to enhance the contract's robustness and prevent potential problems down the line.

Low (L): Minor Imperfections with Limited Repercussions

Low-severity issues are essentially minor imperfections in the smart contract that have a limited impact on user funds or the core functionality of the protocol. Exploiting these would usually require very specific and unlikely scenarios and would yield minimal gain for an attacker. While these findings don't pose an immediate threat, addressing them when feasible can contribute to a more polished and well-maintained codebase.

Informational (I): Opportunities for Improvement, Not Immediate Risks

Informational findings aren't security vulnerabilities in the traditional sense. Instead, they highlight areas related to the clarity and efficiency of the code, gas optimization, the quality of documentation, or adherence to best development practices. These findings don't represent any immediate risk to the security or functionality of the contract but offer valuable insights for improving its overall quality and maintainability. Addressing these is optional but often beneficial for long-term health and clarity.

Types of Issues

Open Security vulnerabilities identified that must be resolved and are currently unresolved.	**Resolved** These are the issues identified in the initial audit and have been successfully fixed.
Acknowledged Vulnerabilities which have been acknowledged but are yet to be resolved.	**Partially Resolved** Considerable efforts have been invested to reduce the risk/impact of the security issue, but are not completely resolved.

Severity Matrix

Likelihood \ Impact	🟥 High	🟧 Medium	🟨 Low
🟥 High	Critical	High	Medium
🟧 Medium	High	Medium	Low
🟨 Low	Medium	Low	Low

Impact

- **High** - leads to a significant material loss of assets in the protocol or significantly harms a group of users.
- **Medium** - only a small amount of funds can be lost (such as leakage of value) or a core functionality of the protocol is affected.
- **Low** - can lead to any kind of unexpected behavior with some of the protocol's functionalities that's not so critical.

Likelihood

- High - attack path is possible with reasonable assumptions that mimic on-chain conditions, and the cost of the attack is relatively low compared to the amount of funds that can be stolen or lost.
- Medium - only a conditionally incentivized attack vector, but still relatively likely.
- Low - has too many or too unlikely assumptions or requires a significant stake by the attacker with little or no incentive.

Medium Severity Issues

Precision loss in fee calculation, which also allows bypassing fees for small calculations.

Acknowledged

Path

Wattoin.sol

Function Name

`_calculateFees()`

Description

For example, the fee would be calculated with this formula
fee = amount.mul(feePercentage).div(100);
Let's say amount = 1 , feePercentage = 5 . So the formula should work like this:

```
fee = (1).mul(5).div(100);
fee = (5).div(100);
```

Here, the answer would be 0.05, but Solidity does not support floating-point numbers. It will only store the 0 as the fee.

(This can also be looked at from the perspective where fees can be bypassed by using smaller values, where the fee will round down to 0, and the receiver will get all the transferred amount.)

Therefore, due to the requirement of non-divisible tokens, the decimals are set to 0. But that can create a problem, as shown in the example above.

Also, it should be noted that because of 0 decimals used and also the denominator (100) used, it won't be able to calculate the fractional fee amounts.

Recommendation

We recommend verifying the mentioned case and also encourage developers to find possible solutions.

Wattoin Team's Response

We are aware of the precision loss in the case of small amounts of coins being transferred - this was a trade-off that was accepted to deliver on the 0 decimals requirement. While it opens up the possibility for abuse to bypass the fee entirely through small transfers, we considered it a way of only taxing big transfers with a fee, not small ones.

The max amount per wallet restriction can be tricked

Acknowledged

Path
Wattoin.sol

Function Name
`_beforeTokenTransfer(`

Description
The _beforeTokenTransfer() checks that if the transferred amount makes the user's balance more than maxAmountPerWallet.

The way it checks it is by checking the current balance of to address and adding the amount to transfer to it. And comparing it with maxAmountPerWallet.

Before receiving any amount, the user can transfer the current amount to another address, making their balance less or zero. So while comparing it with maxAmountPerWallet. It will always pass.

Example: The maxAmountPerWallet is assumed to be 100 in this example.

1. User A has 100 tokens.
2. User A will be receiving more tokens now (e.g, 50 more). But since the maxAmountPerWallet is 100, the transfer() will revert with ERC20: transfer amount exceeds max wallet amount.
3. So, to receive more than 100 tokens the User A sends his tokens to another address, so his current balance becomes 0.
4. Now, whenever more tokens are transferred to his address. It won't revert as the balance of User A is 0 while comparing, so when more tokens are sent to him, the balance still stays less than maxAmountPerWallet.

Additionally, users can create multiple addresses to receive a larger amount per address.

Recommendation
Consider checking that the team is aware of these scenarios.

Wattoin Team's Response
We are aware of the inherent limitations of token volume restrictions and their enforcement. While it can easily be circumvented with multiple wallet addresses, we still wanted to do what was in our hands to make it difficult for hostile investors to amass large amounts of the token under their name.

Low Severity Issues

Floating pragma

Resolved

Path

Wattoin.sol

Description

Contract is using version ^0.8.24 with a floating pragma instead of locking to a specific version. Floating pragmas allow the contract to be compiled with any version greater than or equal to the specified version for that major version. If the contract wasn’t thoroughly tested with that version, this can introduce possible bugs.

Recommendation

Consider using a fixed solidity version whenever possible.

Informational Issues

Add events according to the requirement

Acknowledged

Path
Wattoin.sol

Description
The contract doesn't include any events. Events can be added if required.

Recommendation
Add events according to the requirement.

Wattoin Team's Response
only default ERC20 events are required, so we're all good

Redundant 0 value assignment

Resolved

Path
Wattoin.sol

Function Name
Constructor

Description
The constructor assigns a 0 value to some variables, like maxAmountPerWallet, feePercentage, firePitPercentage.

The default values of these variables are already 0. So the 0 value assignment for these variables is not necessary in the constructor.

Recommendation
Remove the 0 value assignments for the mentioned variables.

Redundant inheritance and import.

Resolved

Path

Wattoin.sol

Description

Context is inherited by Wattoin. But Wattoin also inherits the ERC20Burnable, which already imports Context.

ERC20Burnable is inherited by Wattoin. But since the ERC20Burnable inherits ERC20, the Wattoin contract doesn't need to import the ERC20 as it's unused..

Recommendation

Remove redundant inheritance and imports.

1. Remove the import statement and inheritance of Context.
2. Remove the import of ERC20.

Add checks for max fee

Acknowledged

Path
Wattoin.sol

Function Name
setFeePercentage(), setFirePitPercentage()

Description
setFeePercentage(), setFirePitPercentage() are used to set the fee percentage. Currently, these functions allow setting the fee equal to 100%.

The restriction can be added to set the fee up to the maximum predefined amount, which won't be 100, but less than it.

For example, the functions will only allow setting the percentage amount up to 50(This value should be decided by the project team).

This will make things more transparent for the users and will increase the trust amongst them that the fee can't go more than a specific percentage.

Recommendation
Consider adding max fee checks in both functions. The same check present can be updated. The updated percentage check can look like this: require(percentage <= 50, "ERC20: invalid fire pit percentage");
The actual max amount can be decided by the project team.

Wattoin Team's Response
We decided against this approach, as it would transform the check into a policy-enforcing mechanism rather than serving its intended purpose as a straightforward programmatic safety measure.

Note about fee deduction

Acknowledged

Path

Wattoin.sol

Function Name

_calculateFees()

Description

The _calculateFees() calculates the fee to burn and the fee to send to the treasury. The if{} checks for !feeExemptList(sender) && !feeExemptList(recipient), in this case, if any of the address from sender or recipient is exempted.

For example, if the sender is exempted, then while sending tokens to any recipient, it won't deduct the fee.

Recommendation

Consider verifying if it is according to the business logic and the team doesn't want it to be (!feeExemptList(sender) || !feeExemptList(recipient)) so that even if any one address will be exempted, in this case sender, it will still take the fee.

Redundant function overriding

Resolved

Path

Wattoin.sol

Function Name

increaseAllowance(), decreaseAllowance().

Description

Currently, the Openzeppelin library version used is v4.9.0 . In this version ERC20 implementation already has the increaseAllowance() and decreaseAllowance().

Currently, the Wattoin contract inherits the ERC20 implementation but still overrides both increaseAllowance() and decreaseAllowance().

This redundant function overriding can be removed.

Recommendation

Remove both increaseAllowance() and decreaseAllowance() overridden functions from the Wattoin contract.

Note the intended functionality.

Resolved

Path

Wattoin.sol

Function Name

_beforeTokenTransfer()

Description

_beforeTokenTransfer() requires balanceOf(to).add(amount) <= maxAmountPerWallet to be true. In this case, it uses “<=” and not “<”. It should be checked that it was intended to use “<=”.

Recommendation

Consider checking that it’s intended.

Note about 0 decimals

Acknowledged

Path

Wattoin.sol

Description

The Wattoin is an ERC20 contract that uses 0 decimals. This means one token would be represented by 1. which would not be divisible. Hence, no decimals were considered in the implementation.

Recommendation

This note can be acknowledged.

Correct the contract name in the error message

Resolved

Path

Wattoin.sol

Function Name

setFirePitPercentage(), setFeePercentage()

Description

Functions like setFirePitPercentage() and setFeePercentage() revert in some cases.

The error messages used are ERC20: invalid fee percentage and ERC20: invalid fire pit percentage. In these error messages, the contract name used is ERC20 says the error is from ERC20 contract implementation.

Since currently both functions setFirePitPercentage() and setFeePercentage() are in the Wattoin contract, and also it’s not directly related to the ERC20 implementation, the error messages can be changed to Wattoin: invalid fee percentage and Wattoin: invalid fire pit percentage.

Recommendation

Consider correcting the contract name in the error message.

Functional Tests

Some of the tests performed are mentioned below:

- ✓ Should be able to blacklist the account.
- ✓ Should be able to remove blacklisted the account.
- ✓ Should be able to exempt the account from fees.
- ✓ Should be able to remove the account from the fee exemption list.
- ✓ Should be able to set the max percentage per wallet.
- ✓ Should be able to set the fee and fire pit fee percentage.
- ✓ Should be able to toggle the fee and the fire pit fee.
- ✓ Should be able to set the treasury address.
- ✓ Should be able to transfer the tokens.
- ✓ Should be able to transfer the approved tokens.
- ✓ Should be able to deduct fees while transferring.
- ✗ Should be able to deduct fees for a smaller amount while transferring tokens.

Automated Tests

No major issues were found. Some false positive errors were reported by the tools. All the other issues have been categorized above according to their level of severity.

Threat Model

Contract	Function	Threats
Wattoin	constructor	Failures in variable initialization
	burn	Failure to burn tokens, Access control failure
	burnFrom	Failure to burn approved tokens, Access control failure
	addToBlacklist	Failure to add the address to the blacklist, Access control failure
	removeFromBlacklist	Failure to remove the added address from the blacklist, Access control failure
	addToFeeExemptList	Failure to add the address to the exemption list, Access control failure
	removeFromFeeExemptList	Failure to remove the added address from the exemption list, Access control failure
	setMaxAmountPerWallet	Failure to set the maximum amount allowed per wallet, Access control failure
	toggleFee	Failure to toggle the fee, Access control failure
	toggleFirePit	Failure to toggle the fire pit fee, Access control failure
	setFeePercentage	Failure to set the fee percentage, Access control failure
	setFirePitPercentage	Failure to set the fire pit fee percentage, Access control failure

Contract	Function	Threats
	setTreasury	Failure to set the treasury address, Access control failure
	transfer	Failure to transfer the tokens, failure to calculate fees, precision loss
	transferFrom	Failure to transfer the approved tokens, failure to calculate fees, precision loss

Closing Summary

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

Issues of medium, low, and informational severity were found. Wattoin Team resolved few issues and acknowledged others

Disclaimer

About QuillAudits

QuillAudits is a leading name in Web3 security, offering top-notch solutions to safeguard projects across DeFi, GameFi, NFT gaming, and all blockchain layers. With seven years of expertise, we've secured over 1400 projects globally, averting over $3 billion in losses. Our specialists rigorously audit smart contracts and ensure DApp safety on major platforms like Ethereum, BSC, Arbitrum, Algorand, Tron, Polygon, Polkadot, Fantom, NEAR, Solana, and others, guaranteeing your project's security with cutting-edge practices.

QuillAudits

7+ Years of Expertise	**1M+** Lines of Code Audited
50+ Chains Supported	**1400+** Projects Secured

Follow Our Journey

AUDIT REPORT

August 2025

For

Watt Trade

QuillAudits

Canada, India, Singapore, UAE, UK

www.quillaudits.com audits@quillaudits.com