

**Mock Audit Report #1**

Version 1.0

*0xOwain*

20/08/25

ERC20 Token Contract Audit

0xOwain

March 7, 2023

Prepared by: 0xOwain

# Table of Contents

* Table of Contents
* Protocol Summary
* Disclaimer
* Risk Classification
* Audit Details
  + Scope
  + Severity Criteria
  + Summary of Findings
  + Tools Used
* High
* Medium
* Low
* Informational
* Gas

**Protocol Summary**

The ERC20 token contract implements a standaed fungible token, supports transfers, approvals, and metadata such as name, symbol, and decimals.

# Disclaimer

0xOwain makes all effort to find as many vulnerabilities in the code in the given time period, but holds no responsibilities for the findings provided in this document. A security audit by the team is not an endorsement of the underlying business or product. The audit was time-boxed and the review of the code was solely on the security aspects of the Solidity implementation of the contracts.

# Risk Classification

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  | Impact |  |  |
|  |  | High | Medium | Low |
|  | High | H | H/M | M |
| Likelihood | Medium | H/M | M | M/L |
|  | Low | M | M/L | L |

We use the [CodeHawks](https://docs.codehawks.com/hawks-auditors/how-to-evaluate-a-finding-severity) severity matrix to determine severity. See the documentation for more details.

# Audit Details

**Methodology**

The review combined **manual inspection** and **automated analysis**. The following areas were considered:

* **Access control**: validated that only intended actors can call privileged functions.
* **State management**: checked for correct and consistent updates of balances, allowances, and total supply.
* **Function logic**: reviewed for correctness, unexpected side effects, and adherence to the ERC20 standard.
* **External interactions**: assessed risk of reentrancy and safety of external calls.
* **Event emissions**: verified that critical state changes emit corresponding events.
* **Tooling**: static analysis was performed using *Slither Analyzer* (solc 0.8.20), with results cross-checked manually.

**Scope**

**\*\*Repo:\*\*** OpenZeppelin Contracts v4.x

**\*\*Commit Hash:\*\*** 4a67c6f3ab2be41487889a020c99e11dedbd6eb4

**\*\*Target:\*\*** ERC20 token implementation

**\*\*Contracts in Scope:\*\***

- ERC20.sol (main implementation, 305 LoC)

- IERC20.sol (interface – referenced only)

- Context.sol (utility – no audit focus)

- IERC20Metadata.sol (metadata interface – low risk)

**\*\*Out of Scope:\*\***

- Extensions (ERC20Burnable, ERC20Capped, ERC20Pausable, etc.)

- Upgradeable variants (ERC20Upgradeable.sol)

- Any other unrelated utilities/helpers

**Severity Criteria**

**\*\*High\*\***

- Direct loss of funds or permanent lock of assets.

- Anyone can exploit (not just privileged roles).

- Breaks core protocol functionality.

**\*\*Medium\*\***

- Causes significant disruption (DoS, griefing, governance failure).

- Exploitable under some conditions or requires privileged roles.

- Financial loss is possible but limited.

**\*\*Low\*\***

- Minor issues: inefficiencies, gas waste, unclear logic, small inconsistencies.

- Doesn't threaten core security or funds.

**\*\*Informational / Non-Critical\*\***

- Code style, readability, missing comments.

- Best practices (naming conventions, event emissions, input validation improvements).

- No security impact.

**Summary of Findings**

**High**

No high severity findings

**Medium**

**\*\*M-01: ERC20 Approval Race\*\***

*\*Summary\**

Changing a non-zero allowance with `approve(spender, new)` can be raced by the spender using `transferFrom()`, allowing use of both the old and new values under tx ordering.

*\*Location\**

`ERC20.approve`, `ERC20.transferFrom`, `ERC20.\_approve`, `ERC20.\_spendAllowance`

*\*Description\**

`approve` overwrites `\_allowances[owner][spender]` without requiring `value` to be zero first.  `transferFrom()` spends whatever allowance is current at the execution (No atomic link between "change" and "spend").

*\*Proof of Concept (PoC)\**

1.  Deploy an ERC20 token.

2.  `owner` approves `spender` for 100 tokens:

    `token.approve(spender, 100);`

3.  Later, `owner` wants to reduce to 20, so broadcasts:

    `token.approve(spender, 20);`

4.  Before that tx is mined, `spender` front-runs with:

    `token.transferFrom(owner, attacker, 100);`

    (uses old allowance).

5.  Then the `approve(20)` tx lands, setting a new allowance.

6.  `spender` can now call:

    `token.transferFrom(owner, attacker, 20);`

    (uses new allowance).

Result: Spender drains 120 tokens instead of the intended 20.

*\*Impact\**

Owner's intended reduction/reset can be bypassed during the race window, therefore effective spend may be "old + new".

*\*Severity\**

Medium: known ERC-20 limitation; the exploit requires a motivated spender + mempool timing but results in unexpected extra spend.

*\*Recommendations\**

- Two-step change: `approve(spender, 0)` -> wait confirmation -> `approve(spender, new)`.

- Consider EIP-2612/Permit2 flows to avoid multi-tx races.

*\*Status\**

Known ERC-20 standard limitation, not an implementation bug.

**Low**

No low severity findings

**Informational**

**\*\*L-01: Inconsistent pragma directives\*\***

*\*Description\**

Files within the ERC20 implementation use different pragma constraints (`>=0.4.16`, `>=0.6.2`, `>=0.8.4`, `>=^0.8.20`).

*\*Impact\**

Low.  No direct vulnerability, but compilation differences could introduce small discrepancies across environments.

*\*Recommendation\**

Standardise pragma directives (e.g., `^0.8.20`) across all contracts).

\*Source\*  
Detected by Slither Analyzer (solc 0.8.20):

A screenshot of a computer program

AI-generated content may be incorrect.

# Gas

No significant gas optimizations identified within the ERC20 implementation.