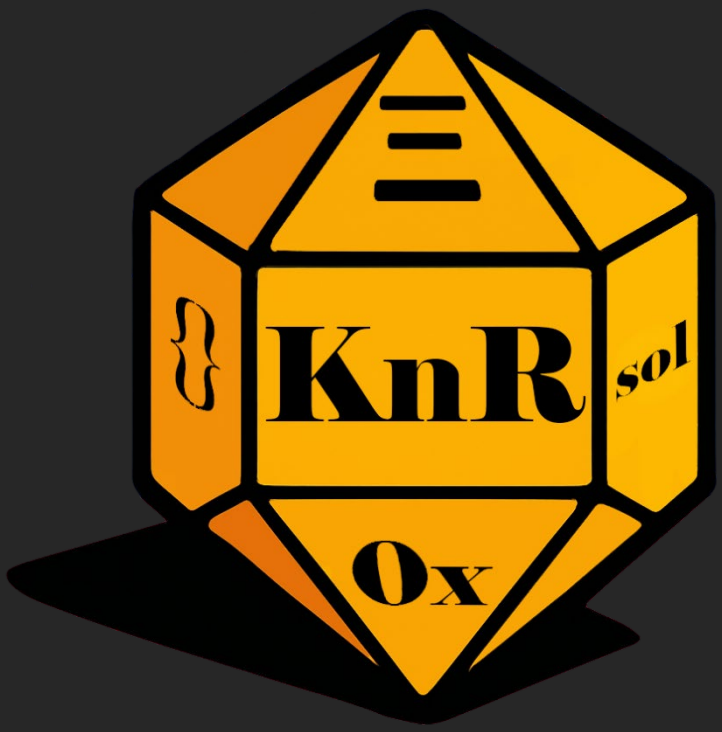

Kode-n-Rolla

Token-0x Security Review

Dec 4th 2025 - Dec 11th 2025

Table of Contents

- **Whoami**.....3
- **Disclaimer**.....3
- **Risk Classification**.....3
- **Protocol Summary**.....4
- **Audit Details**.....4
  - **Scope**.....5
  - **Roles**.....5
- **Executive Summary**.....6
  - **Issue found**.....6
- **Findings**.....6
- **High**.....6
- [H-01] Self-transfer inflates user balance due to stale storage reads in *_transfer*, allowing unlimited minting.....6
- [H-02] Unchecked *burn* underflow lets attackers mint enormous balances and inflate *totalSupply*.....11

Whoami

I am Pavel (aka *kode-n-rolla*), an independent security researcher.

Driven by a passion for decentralization and security, I focus on strengthening blockchain ecosystems.

I believe that security is an ongoing process, and my work contributes to making systems more resilient and reliable.

Disclaimer

This report reflects a time-boxed security review performed as part of a ***CodeHawks*** audit contest.

The analysis focuses exclusively on the security aspects of the reviewed ***Solidity/Yul*** codebase and does not guarantee the discovery of all potential vulnerabilities.

Findings represent the issues identified during the allotted review period and should not be considered an endorsement of the underlying protocol, business model, or operational security practices.

Risk Classification

Severity	Impact: High	Impact: Medium	Impact: Low
Likelihood: High	H	H/M	M
Likelihood: Medium	H/M	M	M/L
Likelihood: Low	M	M/L	L

I used *CodeHawks* severity matrix to determine severity.

Impact refers to the potential harm or consequence to the users or the protocol due to the vulnerability.

- *High Impact*:
    - Funds are directly or nearly directly at risk

    - There's a severe disruption of protocol functionality or availability.
- *Medium Impact*:
    - Funds are indirectly at risk.
    - There's some level of disruption to the protocol's functionality or availability.
- *Low Impact*:
    - Funds are not at risk.
    - However, a function might be incorrect, the state might not be handled appropriately, etc.

Protocol Summary

Token-0x is a custom ERC20 token implementation focused on **gas efficiency** and **minimalism** while remaining compliant with the ERC20 standard.

The protocol provides the full set of required ERC20 functionality, but implements core token logic using a **hybrid approach combining Solidity and low-level Yul (inline assembly)**. This design choice aims to reduce gas costs and execution overhead compared to traditional high-level implementations such as *OpenZeppelin*.

Token-0x is intended to be used as a drop-in ERC20 token, exposing standard ERC20 interfaces for transfers, approvals, and balance tracking, while internally relying on optimized low-level operations to achieve cheaper and more performant execution.

Audit Details

This audit was conducted as part of a time-boxed competitive security review hosted on the *CodeHawks* platform. The assessment focused exclusively on the ***Solidity/Yul*** smart contract logic and its adherence to intended game mechanics, security assumptions, and fund safety.

The review did not include off-chain components, frontend integrations, or economic modeling beyond what is explicitly enforced by the contract code.

The review was performed during the "Token-0x" *CodeHawks* audit event.

Scope

The audit scope consisted of the following smart contract:

```
├── src
│   ├── ERC20.sol
│   ├── IERC20.sol
│   └── helpers
│       ├── IERC20Errors.sol
│       └── ERC20Internals.sol
```

Roles

- ***Token Holders***

Any externally owned account (EOA) or smart contract that holds Token-0x balances. Token holders can transfer tokens, approve allowances, and interact with third-party protocols using Token-0x as a standard ERC20 asset.

- ***Integrating Protocols***

DeFi protocols and smart contracts that integrate Token-0x as a base ERC20 token (e.g., for rewards distribution, governance tokens, utility tokens, or internal accounting). These actors rely on strict ERC20 compliance and correct low-level behavior of token operations.

- ***External Callers***

Any contract or system interacting with Token-0x via the ERC20 interface, including DEXes, vaults, bridges, and off-chain services (indexers, explorers).

Executive Summary

The audit identified **two High-severity vulnerabilities** in the Token-0x ERC20 implementation. Both issues allow attackers to **mint arbitrary amounts of tokens**, fully compromising the token's economic integrity and breaking core ERC20 invariants.

The first vulnerability stems from **incorrect handling of self-transfers** in the _transfer logic, where stale storage reads lead to balance inflation when `from == to`. This flaw enables an attacker to repeatedly self-transfer tokens and increase their balance without limit.

The second vulnerability is caused by an **unchecked underflow in the burn logic**, allowing attackers to manipulate internal accounting and mint extremely large balances while also corrupting the *totalSupply*.

Due to these issues, **Token-0x cannot be safely used in production or integrated into DeFi protocols**, as attackers can create unlimited supply, drain protocol funds, manipulate markets, and break downstream assumptions in any system relying on the token's correctness.

Issue found

Severity	Issue
High	Self-transfer inflates user balance due to stale storage reads in *_transfer*, allowing unlimited minting.
High	Unchecked *burn* underflow lets attackers mint enormous balances and inflate *totalSupply*.

Findings

Highs

[H-01] Self-transfer inflates user balance due to stale storage reads in *_transfer*, allowing unlimited minting

Description

In *ERC20Internals.sol::_transfer*, self-transfers (`from == to`) result in **balance inflation** due to incorrect balance update logic.

When from and to are the same address:

- Both *fromAmount* and *toAmount* are loaded from the **same storage slot** (`balances[from]`).
- The function subtracts value from the original balance and stores it.
- It then **adds value to the previously read (stale) balance**, not to the updated value.

As a result, the final balance becomes:

```
finalBalance = oldValue + value
```

while *totalSupply* remains unchanged.

```
...
            }

            sstore(fromSlot, sub(fromAmount, value))
@>          sstore(toSlot, add(toAmount, value)) // balance update
            success := 1
            mstore(ptr, value)
            log3(ptr, 0x20,
0xddf252ad1be2c89b69c2b068fc378daa952ba7f163c4a11628f55a4df523b3ef, from, to)
        }
}
```

This allows an attacker to repeatedly self-transfer tokens and **mint an arbitrary number of tokens**, fully breaking ERC-20 accounting guarantees.

Impact

A malicious token holder can repeatedly perform self-transfers to **inflate their own balance without increasing** ***totalSupply***, effectively minting an unbounded number of tokens and **violating core ERC-20 accounting invariants**.

With an arbitrarily inflated balance, an attacker can:

- Drain liquidity from DEX pools.
- Manipulate lending and collateral-based systems.
- Exploit reward distribution mechanisms.
- Influence or fully control governance systems relying on token balances.
- Cause irreversible loss of funds from integrated protocols and treasuries.

This vulnerability results in a **complete economic compromise of the token** and renders Token-0x **unsafe for any production use or DeFi integration**.

Likelihood

Self-transfers are **valid and commonly permitted ERC-20 operations**.

- A self-transfer via *`transfer(to, value)`* occurs whenever a user calls the function with *`to == msg.sender`*. This pattern is frequently triggered by wallets, airdrop scripts, batch tools, and generic ERC-20 integrations that do not explicitly guard against self-transfers.
- A self-transfer via *`transferFrom(from, to, value)`* occurs whenever an approved spender calls the function with *`from == to`*. Since users can grant allowances to themselves or to arbitrary spenders, **any token holder can unilaterally trigger the vulnerability without special privileges**.

Given the lack of access control and the ubiquity of self-transfer paths in ERC-20 usage, the vulnerability is **highly likely to be exploited.**

Proof of Concept

```
    function test_transferFundsToYourself() public {
        address attacker = makeAddr("hacker");
        uint256 mintAmount = 10e18;
        uint256 transferAmount = 1e18;

        // Mint tokens to the attacker
        vm.prank(attacker);
        token.mint(attacker, mintAmount);


        uint256 balanceBefore = token.balanceOf(attacker);
        uint256 totalSupplyBefore = token.totalSupply();

        // --- Phase 1: self-transfer via transfer() ---
        vm.startPrank(attacker);
        token.approve(attacker, mintAmount);
        token.transfer(attacker, transferAmount);
        vm.stopPrank();

        uint256 balanceAfterTransfer = token.balanceOf(attacker);
        uint256 supplyAfterTransfer = token.totalSupply();

        // Balance increased while totalSupply stayed the same
        assertGt(
            balanceAfterTransfer,
            balanceBefore,
            "Self-transfer via transfer() did not increase balance as
expected (inflation bug)"
        );
        assertEq(
            supplyAfterTransfer,
            totalSupplyBefore,
            "totalSupply should remain unchanged after transfer()"
        );

        // --- Phase 2: self-transfer via transferFrom() ---
        vm.prank(attacker);
        token.transferFrom(attacker, attacker, transferAmount);

        uint256 balanceAfterTransferFrom = token.balanceOf(attacker);
        uint256 supplyAfterTransferFrom = token.totalSupply();

        // Balance increased again, still with the same totalSupply
        assertGt(
            balanceAfterTransferFrom,
```

```
            balanceAfterTransfer,
            "Self-transfer via transferFrom() did not further increase
balance as expected"
        );
        assertEq(
            supplyAfterTransferFrom,
            totalSupplyBefore,
            "totalSupply should remain unchanged after transferFrom()"
        );
    }
```

Output result:

```
Ran 1 test for test/Token.t.sol:TokenTest
[PASS] test_transferFundsToYourself() (gas: 100963)
Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 2.88ms (1.04ms
CPU time)
```

Recommended Mitigation

The issue is caused by reading both *fromAmount* and *toAmount* before any storage updates and then writing twice to the same balance slot when *from == to*, which results in inflation due to stale reads.

Prevent the vulnerable execution path by short-circuiting self-transfers **before entering the Yul block**:

```
function _transfer(address from, address to, uint256 value) internal returns
(bool success) {
+   // Mitigate slot collision on self-transfer
+   if (from == to) {
+       // optional: emit Transfer(from, to, 0) if desired
+       return true;
+   }
+
    assembly ("memory-safe") {
```

A high-level early return is preferred because it removes the vulnerable execution path entirely before any low-level logic executes. This keeps the fix minimal, readable, and easy to verify, while avoiding additional branching inside

Yul (which increases complexity and the risk of introducing new low-level bugs).

[H-02] Unchecked *burn* underflow lets attackers mint enormous balances and inflate *totalSupply*

Description

In *ERC20Internals::_burn*, both *totalSupply* and *balances[account]* are updated in inline assembly using *sub(...)* **without any prior balance or supply validation.** Since arithmetic in *Yul* operates modulo *2^256*, if value exceeds either the account's balance or *totalSupply*, the subtraction **underflows and wraps around**, producing extremely large numbers that are then stored.

As a result, an "over-burn" causes:

- *balances[account]* to underflow to a huge value, effectively **minting** tokens to the account.
- *totalSupply* to underflow to a huge value, corrupting global supply accounting.

This completely breaks ERC-20 accounting guarantees and enables an attacker (if the burn path is externally reachable) to inflate their balance to near *type(uint256).max* with a single call.

Issue in *ERC20Internals.sol* at 171 and 178 lines

```
        let supplySlot := _totalSupply.slot

        let supply := sload(supplySlot)
@>      sstore(supplySlot, sub(supply, value))

        mstore(ptr, account)
        mstore(add(ptr, 0x20), balanceSlot)

        let accountBalanceSlot := keccak256(ptr, 0x40)
        let accountBalance := sload(accountBalanceSlot)
@>      sstore(accountBalanceSlot, sub(accountBalance, value))
    }
```

Impact

A malicious user can call the burn entrypoint with an oversized *value* to force both *balances[account]* and *totalSupply* to **underflow and wrap** to values close to *type(uint256).max*, effectively **minting** an enormous amount of tokens in a single transaction instead of burning.

With an artificially inflated balance and corrupted supply accounting, the attacker can:

- Drain DEX liquidity pools.
- Break lending/collateral systems and seize assets.
- Manipulate reward distribution and emissions.
- Capture governance power in systems that rely on balances or supply-derived assumptions.

This results in a **complete economic compromise** of the token and can cause catastrophic and unrecoverable losses for any protocol integrating Token-0x.

Likelihood

The vulnerability is triggered whenever *_burn* is reachable via a *public/external* entrypoint (e.g., *burn(uint256)*, *burn(address,uint256)*, or *burnFrom(...)*) that allows a caller to supply a value greater than the caller's balance and/or the current *totalSupply*.

This condition is satisfied in the provided Token test contract, and exposing burn functionality is a **common pattern** for burnable ERC-20 tokens. As a result, exploitation is **permissionless** and **trivial** once a burn entrypoint exists without strict precondition checks.

In addition, any future token or integration that reuses this *_burn* implementation (or copies the assembly block) without adding explicit *balance >= value* and *totalSupply >= value* validation will inherit the same vulnerability, making the issue highly likely to reappear as the codebase evolves.

Proof of Concept

The following Foundry test demonstrates that calling *burn* with a value larger than the caller's balance causes both *totalSupply* and the caller's balance to **increase** due to unchecked underflow in *ERC20Internals::_burn*.

This proof of concept shows that an oversized burn operation results in effective token minting instead of burning.

Steps to reproduce:

Paste the test below into *test/Token.t.sol* and run *forge test*.

```
    function test_underflowViaBurn() public {
        address attacker = makeAddr("hacker");
        uint256 mintAmount = 10e18;

        // Mint some tokens to the attacker
        vm.prank(attacker);
        token.mint(attacker, mintAmount);

        uint256 totalSupplyBefore = token.totalSupply();
        uint256 attackerBalanceBefore = token.balanceOf(attacker);

        // Burn more than the attacker balance
        vm.prank(attacker);
        token.burn(attacker, 100e18);

        uint256 totalSupplyAfter = token.totalSupply();
        uint256 attackerBalanceAfter = token.balanceOf(attacker);

        // Both totalSupply and attacker balance increased due to underflow
        assertGt(
            totalSupplyAfter,
            totalSupplyBefore,
            "Burning more than total supply did not increase totalSupply as
expected (underflow bug)"
        );
        assertGt(
            attackerBalanceAfter,
            attackerBalanceBefore,
            "Burning more than balance did not increase attacker balance as
expected (underflow bug)"
        );
    }
```

Recommended Mitigation

Add explicit **supply** and **balance** validation to *_burn* **before performing any arithmetic in inline assembly**, mirroring the safety pattern already used in *_transfer* and *_spendAllowance*.

The function must revert when value exceeds either *totalSupply* or the account's balance. Only after these preconditions are satisfied should storage be updated. This prevents arithmetic underflow and preserves ERC-20 accounting invariants.

```
-            sstore(supplySlot, sub(supply, value))
+            // Prevent totalSupply underflow: revert when value > supply
+            if lt(supply, value) {
+                // minimal revert; can be replaced with a dedicated error
selector
+                revert(0x00, 0x00)
+            }

             mstore(ptr, account)
             mstore(add(ptr, 0x20), balanceSlot)

             let accountBalanceSlot := keccak256(ptr, 0x40)
             let accountBalance := sload(accountBalanceSlot)

-            sstore(accountBalanceSlot, sub(accountBalance, value))
+            // Prevent balance underflow: revert when value > accountBalance
+            if lt(accountBalance, value) {
+                // reuse the existing ERC20InsufficientBalance_style error
if desired
+                mstore(0x00, shl(224, 0xfc14934b)) // selector placeholder
+                mstore(add(0x00, 4), account)
+                mstore(add(0x00, 0x24), accountBalance)
+                mstore(add(0x00, 0x44), value)
+                revert(0x00, 0x64)
+            }
+
+            // Safe updates (no underflow after the checks)
+            sstore(supplySlot, sub(supply, value))
+            sstore(accountBalanceSlot, sub(accountBalance, value))
        }
    }
```