Token-0x Competitive Audit

github.com/lucasfhope

December 11, 2025

Contents

Competitive Audit Details

This competitive audit is a CodeHawks First Flight.

The findings described in this report correspond to the following commit hash:

```
7f9f55d58a485a36fb56284d8d0e8a415544bf9b
```

Protocol Summary

Token-0x, A secure and cheap base ERC20 implementation, which follows the ERC20 standard. Token-0x has implemented all the necessary functions required to be a compliant ERC20 token but in different way. Token-0x achieves the secure and cheap operations by using a combination of Solidity and Yul in the base implementation.

Scope

```
src/
 --- ERC20.sol
 --- IERC20.sol
 --- helpers/
   ---IERC20Errors.sol
   ---ERC20Internals.sol
```

Roles

Protocols can use this token as a base token for their protocol for rewards, native token for their protocol, etc.

Findings

Severity	Number of valid findings
High	1
Medium	1
Low	1
Total	**3**

[High-1] Inline assembly blocks in `totalSupply_` and `_balanceOf` contains the `return` opcode

Description

Both `totalSupply_` and `_balanceOf` use `assembly { return(...)}` to return values.

In the EVM, a raw `return` opcode exits the entire current call frame, not just the internal helper. When these helpers are used inside another function within the same contract, execution jumps straight out of the calling function and returns to the external caller. Any Solidity statements after the helper call are skipped entirely.

This becomes a problem only inside the ERC20 contract or in contracts that inherit from it. External callers will see the expected return value, because the early return ends only the token's call frame, not theirs. However, if an inheriting contract expects to perform additional computation after calling `_balanceOf` or `totalSupply_`, that logic will never run.

```
1  function totalSupply_() internal view returns (uint256) {
2      assembly {
3          let slot := _totalSupply.slot
4          let supply := sload(slot)
5          mstore(0x00, supply)
6          // @audit returns from call frame
7  @>      return(0x00, 0x20)
8      }
9  }
10
11 function _balanceOf(address owner) internal view returns (uint256) {
12     assembly {
13         if iszero(owner) {
14             revert(0, 0)
15         }
16
```

```
17          let baseSlot := _balances.slot
18          let ptr := mload(0x40)
19          mstore(ptr, owner)
20          mstore(add(ptr, 0x20), baseSlot)
21          let dataSlot := keccak256(ptr, 0x40)
22          let amount := sload(dataSlot)
23          mstore(ptr, amount)
24          mstore(add(ptr, 0x20), 0)
25          // @audit returns from call frame
26  @>      return(ptr, 0x20)
27      }
28  }
```

Risk

Likelihood:

This issue occurs when inheriting contracts call `_balanceOf` or `totalSupply_` inside a function that performs additional logic afterward. In such cases, the early assembly **`return`** causes the caller function to exit prematurely. This is a realistic scenario in extended ERC20 implementations, though inheriting contracts can technically access the storage variables directly instead of using these helpers.

Impact:

The calling function inside the ERC20 or its child contract will silently skip all logic after `_balanceOf` or `totalSupply_`. Therefore, any inheriting contracts that have functions that use `_balanceOf` or `totalSupply_` could have broken calculations, incorrect return values, and skip state updates.

While external integrations such as DeFi protocols calling balanceOf() or totalSupply() are unaffected, any extended logic inside the ERC20 contract itself becomes unsafe.

Proof of Concept

```
1  import {Test} from "forge-std/Test.sol";
2  import {Token} from "test/Token.sol";
3  import {ERC20} from "src/ERC20.sol";
4
5  contract EarlyReturnTest is Test {
6      ExtendedToken badToken;
7
8      address user1 = makeAddr("user1");
9      address user2 = makeAddr("user2");
10     uint256 amount = 1000;
11
12     function setUp() public {
13         badToken = new ExtendedToken();
```

```
14          badToken.mint(user1, amount);
15          badToken.mint(user2, amount);
16      }
17
18
19      function testTotalSupplyEarlyReturn() public {
20          uint256 otherTokens = badToken.otherTokens(user1);
21          // other tokens should be (totalSupply_ - _balanceOf) -> (
                amount * 2) - amount = amount
22          // but since totalSupply_ returns early, otherTokens ==
                totalSupply_, which is amount * 2
23          assert(otherTokens != (amount * 2) - amount);
24          assert(otherTokens == amount * 2);
25      }
26
27      function testBalanceOfEarlyReturn() public {
28          uint256 doubleBalance = badToken.doubleBalance(user1);
29          // double balance should be _balanceOf * 2 -> amount * 2
30          // but since _balanceOf returns early, doubleBalance ==
                _balanceOf, which is amount
31          assert(doubleBalance != amount * 2);
32          assert(doubleBalance == amount);
33      }
34
35  }
36
37  contract ExtendedToken is ERC20 {
38      constructor() ERC20("ExtendedToken", "EXT") {}
39
40      function mint(address account, uint256 value) public {
41          _mint(account, value);
42      }
43
44      function burn(address account, uint256 value) public {
45          _burn(account, value);
46      }
47
48      function otherTokens(address account) public view returns (uint256)
            {
49          uint256 supply = totalSupply_();
50          uint256 balance = _balanceOf(account);
51          return supply - balance;
52      }
53
54      function doubleBalance(address account) public view returns (
           uint256) {
55          uint256 balance = _balanceOf(account);
56          return balance * 2;
57      }
58  }
```

`ExtendedToken` inherits from the ERC20 contract and uses `totalSupply_` and `_balanceOf` inside functions that perform additional logic. Because these internal helpers perform a raw assembly **`return`**, the parent functions exit early and return only the helper value, not the computed result.

Recommended Mitigation

```
function totalSupply_() internal view returns (uint256) {
    uint256 supply;
    assembly {
        let slot := _totalSupply.slot
-       let supply := sload(slot)
+       supply := sload(slot)
-       mstore(0x00, supply)
-       return(0x00, 0x20)
    }
+       return supply
}

function _balanceOf(address owner) internal view returns (uint256) {
+   uint256 amount;
    assembly {
        if iszero(owner) {
            revert(0, 0)
        }

        let baseSlot := _balances.slot
        let ptr := mload(0x40)
        mstore(ptr, owner)
        mstore(add(ptr, 0x20), baseSlot)
        let dataSlot := keccak256(ptr, 0x40)
-       let amount := sload(dataSlot)
+       amount := sload(dataSlot)
-       mstore(ptr, amount)
-       mstore(add(ptr, 0x20), 0)
-       return(ptr, 0x20)
    }
+       return amount
}
```

This change eliminates the early-exit behavior by removing the assembly **`return`** opcode and allowing Solidity to perform the function return normally.

[Medium-1.1] `ERC20Internals::_totalSupply` is unprotected from overflow and underflow in `_mint` and `_burn`

Description

ERC20 implementations should guard `totalSupply` against overflow or underflow in when minting and burning.

This is left unguarded both`_mint` and `_burn`. Since arithmetic is done in inline assembly, add and sub will wrap on overflow, allowing `totalSupply` to roll from `type(uint256).max` to `0` and from `0` to `type(uint256).max`.

```
function _mint(address account, uint256 value) internal {
    assembly ("memory-safe") {
        if iszero(account) {
            mstore(0x00, shl(224, 0xec442f05))
            mstore(add(0x00, 4), 0x00)
            revert(0x00, 0x24)
        }

        let ptr := mload(0x40)
        let balanceSlot := _balances.slot
        let supplySlot := _totalSupply.slot

        // @audit high - no overflow check on total supply
        let supply := sload(supplySlot)
@>      sstore(supplySlot, add(supply, value))

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

        let accountBalanceSlot := keccak256(ptr, 0x40)
        let accountBalance := sload(accountBalanceSlot)
        sstore(accountBalanceSlot, add(accountBalance, value))
    }
}

 function _burn(address account, uint256 value) internal {
    assembly ("memory-safe") {
        if iszero(account) {
            mstore(0x00, shl(224, 0x96c6fd1e))
            mstore(add(0x00, 4), 0x00)
            revert(0x00, 0x24)
        }

        let ptr := mload(0x40)
        let balanceSlot := _balances.slot
        let supplySlot := _totalSupply.slot

```

```
38          // @audit high - no underflow check on total supply
39          let supply := sload(supplySlot)
40  @>      sstore(supplySlot, sub(supply, value))
41
42          mstore(ptr, account)
43          mstore(add(ptr, 0x20), balanceSlot)
44
45          let accountBalanceSlot := keccak256(ptr, 0x40)
46          let accountBalance := sload(accountBalanceSlot)
47          sstore(accountBalanceSlot, sub(accountBalance, value))
48      }
49  }
```

Risk

Likelihood:

This issue occurs when an inheriting contract relies on the ERC20 implementation to prevent `totalSupply` overflows and underflows. Inheriting contracts can implement their own checks, but many will assume this is handled internally, as it is in ERC20 implementations from OpenZeppelin, Solady, etc.

Reaching `type(uint256).max` is unlikely in practice, but underflow is much more possible whenever a caller executes `_burn` with `value > totalSupply` and there is no higher-level guard.

Impact:

DeFi protocols rely on the internal accounting of ERC20s. Any overflow or underflow of `totalSupply` could severly affect protocols that utilize governance tokens, interest-bearing tokens, or tokenized vaults (ERC4626), as well as any contract or off-chain indexers that read and use the `totalSupply`.

Proof of Concept

Add the following test to the test suite in `test/Token.t.sol`.

```
1  function testNoOverflowOrUnderflowChecksForTotalSupplyInBurnOrMint()
       public {
2      address user1 = makeAddr("user1");
3      address user2 = makeAddr("user2");
4      token.mint(user1, type(uint256).max);
5      token.mint(user2, 1);
6      // overflowed, type(uint256).max + 1 = 0
7      assert(token.totalSupply() == 0);
8      token.burn(user2, 1);
9      // underflowed, 0 - 1 = type(uint256).max
10     assert(token.totalSupply() == type(uint256).max);
```

```
11 }
```

This test shows that the token's `totalSupply` will overflow back to `0` after reaching `type(uint256).max`, and the token's `totalSupply` will also underflow to `type(uint256).max` after subtracting past `0`.

Recommended Mitigation

Checks should be added to prevent overflows and underflows. Preventing overflows of `totalSupply` is also important to protect individual balances from overflowing.

```
function _mint(address account, uint256 value) internal {
    assembly ("memory-safe") {
        if iszero(account) {
            mstore(0x00, shl(224, 0xec442f05))
            mstore(add(0x00, 4), 0x00)
            revert(0x00, 0x24)
        }

        let ptr := mload(0x40)
        let balanceSlot := _balances.slot
        let supplySlot := _totalSupply.slot

        let supply := sload(supplySlot)
+       let newSupply := add(supply, value)
+       if lt(newSupply, supply) {
+           revert(0,0)
+       }
-       sstore(supplySlot, add(supply, value))
+       sstore(supplySlot, newSupply)

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

        let accountBalanceSlot := keccak256(ptr, 0x40)
        let accountBalance := sload(accountBalanceSlot)
        sstore(accountBalanceSlot, add(accountBalance, value))
    }
}

 function _burn(address account, uint256 value) internal {
    assembly ("memory-safe") {
        if iszero(account) {
            mstore(0x00, shl(224, 0x96c6fd1e))
            mstore(add(0x00, 4), 0x00)
            revert(0x00, 0x24)
        }

```

```
38          let ptr := mload(0x40)
39          let balanceSlot := _balances.slot
40          let supplySlot := _totalSupply.slot
41
42          let supply := sload(supplySlot)
43  +       if lt(supply, value) {
44  +           revert(0,0)
45  +       }
46          sstore(supplySlot, sub(supply, value))
47
48          mstore(ptr, account)
49          mstore(add(ptr, 0x20), balanceSlot)
50
51          let accountBalanceSlot := keccak256(ptr, 0x40)
52          let accountBalance := sload(accountBalanceSlot)
53          sstore(accountBalanceSlot, sub(accountBalance, value))
54      }
55  }
```

Make sure to replace both instances of `revert(0, 0)` with a custom error to align with the rest of the error handling in the contract.

[Medium-1.2] Individual user balances are unprotected from underflow in `ERC20Internals::_burn`

Description

Similar to `transfer`, an ERC20 `burn` operation must prevent user balances from going below `0`, since subtracting more than the current balance will underflow and wrap to `type(uint256).max`.

In `ERC20Internals::_burn`, the contract subtracts value from the user's balance in inline assembly without checking that `value <= accountBalance`. Because inline assembly uses unchecked arithmetic, the balance can underflow and wrap to `type(uint256).max`.

```
1  function _burn(address account, uint256 value) internal {
2      assembly ("memory-safe") {
3          if iszero(account) {
4              mstore(0x00, shl(224, 0x96c6fd1e))
5              mstore(add(0x00, 4), 0x00)
6              revert(0x00, 0x24)
7          }
8
9          let ptr := mload(0x40)
10         let balanceSlot := _balances.slot
11         let supplySlot := _totalSupply.slot
12
13         let supply := sload(supplySlot)
```

```
14          sstore(supplySlot, sub(supply, value))
15
16          mstore(ptr, account)
17          mstore(add(ptr, 0x20), balanceSlot)
18
19          // @audit high - no underflow check on account balance
20          let accountBalanceSlot := keccak256(ptr, 0x40)
21          let accountBalance := sload(accountBalanceSlot)
22  @>      sstore(accountBalanceSlot, sub(accountBalance, value))
23      }
24  }
```

Risk

Likelihood:

Underflow occurs whenever `_burn` is called with an amount greater than the user's balance. Because `_burn` does not enforce this check, any misuse by an inheriting contract or future extension can immediately trigger the issue.

Impact:

This breaks the ERC20 accounting, causing incorrect pricing and share accounting. An underflow in a user's balance will inflate the token balance to `type(uint256).max`, which could enable protocol This breaks ERC20 accounting, causing incorrect pricing, share accounting, and governance weight. An underflow in a user's balance inflates it to `type(uint256).max`, which can enable manipulation of any protocol that accepts the token.

Proof of Concept

Add the following test to the test suite in `test/Token.t.sol`.

```
1  function testNoUnderflowCheckForIndividualBalanceInBurn() public {
2      address user1 = makeAddr("user1");
3      address user2 = makeAddr("user2");
4      // just in case the _burn function actually checks totalSupply
           underflow
5      token.mint(user1, 1);
6      token.burn(user2, 1);
7      // underflowed after 0 - 1 = type(uint256).max
8      assert(token.balanceOf(user2) == type(uint256).max);
9  }
```

This test shows that the token balance of `user2` underflowed to `type(uint256).max` after a burn when `user2` had a balance of `0`.

Recommended Mitigation

```
function _burn(address account, uint256 value) internal {
    assembly ("memory-safe") {
        if iszero(account) {
            mstore(0x00, shl(224, 0x96c6fd1e))
            mstore(add(0x00, 4), 0x00)
            revert(0x00, 0x24)
        }

        let ptr := mload(0x40)
        let balanceSlot := _balances.slot
        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)
+       if lt(accountBalance, value) {
+          revert(0,0)
+       }
        sstore(accountBalanceSlot, sub(accountBalance, value))
    }
}
```

Make sure to replace `revert(0, 0)` with a custom error to align with the rest of the error handling in the contract.

[Low-1] Calls to `_balanceOf` and `_allowance` with `address(0)` revert instead of returning `0`

Description

Under standard ERC20 behavior, querying the balance of an account or the allowance between two addresses is a harmless view operation that never reverts. When `address(0)` is used, the expected behavior is simply to return `0`:

- `balanceOf(address(0))` → `0`
- `allowance(address(0), spender)` → `0`
- `allowance(owner, address(0))` → `0`

In this implementation, both `_balanceOf` and `_allowance` revert when given `address(0)`:

- `_balanceOf` reverts if `owner == address(0)`
- `_allowance` reverts if `owner == address(0)` or `spender == address(0)`

Calls will revert in cases where a normal ERC20 would safely return `0`. This breaks expected ERC20 read semantics and can affect integrations or tooling that rely on zero-address queries being safe.

```
function _balanceOf(address owner) internal view returns (uint256) {
    assembly {
        // @audit low - shouldn't revert
        if iszero(owner) {
@>          revert(0, 0)
        }

        let baseSlot := _balances.slot
        let ptr := mload(0x40)
        mstore(ptr, owner)
        mstore(add(ptr, 0x20), baseSlot)
        let dataSlot := keccak256(ptr, 0x40)
        let amount := sload(dataSlot)
        mstore(ptr, amount)
        mstore(add(ptr, 0x20), 0)
        return(ptr, 0x20)
    }
}

function _allowance(address owner, address spender) internal view
   returns (uint256 remaining) {
    assembly {
        // @audit low - shouldn't revert
        if or(iszero(owner), iszero(spender)) {
@>          revert(0, 0)
        }

        let ptr := mload(0x40)
        let baseSlot := _allowances.slot

        mstore(ptr, owner)
        mstore(add(ptr, 0x20), baseSlot)
        let initialHash := keccak256(ptr, 0x40)
        mstore(ptr, spender)
        mstore(add(ptr, 0x20), initialHash)

        let allowanceSlot := keccak256(ptr, 0x40)
        remaining := sload(allowanceSlot)
    }
}
```

Risk

Likelihood:

Zero-address balance and allowance reads occur in both on-chain validation paths and off-chain tooling. Because ERC20s normally return `0` for these cases, unexpected reverts can to appear during real integrations.

Impact:

The issue does not threaten funds or core token functionality, but it does cause unnecessary reverts in systems that assume ERC20-standard behavior. This reduces compatibility and may require special-case handling for this token.

Recommended Mitigation

Remove the zero-address checks in `_balanceOf` and `_allowance`.

The `_balances` mapping is never updated for `address(0)` in `_burn` and `_transfer` reverts if it is `to` or `from` `address(0)`, so it will always return `0`.

Similarly, `_approve` reverts when `owner` or `spender` is `address(0)`, so `_allowance` can safely rely on the default mapping value and will return `0` when either is `address(0)`.

```
function _balanceOf(address owner) internal view returns (uint256) {
    assembly {
-       if iszero(owner) {
-           revert(0, 0)
-       }

        let baseSlot := _balances.slot
        let ptr := mload(0x40)
        mstore(ptr, owner)
        mstore(add(ptr, 0x20), baseSlot)
        let dataSlot := keccak256(ptr, 0x40)
        let amount := sload(dataSlot)
        mstore(ptr, amount)
        mstore(add(ptr, 0x20), 0)
        return(ptr, 0x20)
    }
}

function _allowance(address owner, address spender) internal view
   returns (uint256 remaining) {
    assembly {
-       if or(iszero(owner), iszero(spender)) {
-           revert(0, 0)
-       }
```

```
24
25          let ptr := mload(0x40)
26          let baseSlot := _allowances.slot
27
28          mstore(ptr, owner)
29          mstore(add(ptr, 0x20), baseSlot)
30          let initialHash := keccak256(ptr, 0x40)
31          mstore(ptr, spender)
32          mstore(add(ptr, 0x20), initialHash)
33
34          let allowanceSlot := keccak256(ptr, 0x40)
35          remaining := sload(allowanceSlot)
36      }
37  }
```